National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

EVALUATING ERROR RECOVERY MECHANISMS IN THE XPRESS TRANSFER PROTOCOL

AYUB ASAMBA

A Major Report
in
The Department
of
Computer Science

Presented in Partial Fulfilment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec

March, 1994

©AYUB ASAMBA,1994

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

THE AUTHOR HAS GRANTED AN IRREVOCABLE NON-EXCLUSIVE LICENCE ALLOWING THE NATIONAL LIBRARY OF CANADA TO REPRODUCE, LOAN, DISTRIBUTE OR SELL COPIES OF HIS/HER THESIS BY ANY MEANS AND IN ANY FORM OR FORMAT, MAKING THIS THESIS AVAILABLE TO INTERESTED PERSONS.

L'AUTEUR A ACCORDE UNE LICENCE IRREVOCABLE ET NON EXCLUSIVE PERMETTANT A LA BIBLIOTHEQUE NATIONALE DU CANADA DE REPRODUIRE, PRETER, DISTRIBUER OU VENDRE DES COPIES DE SA THESE DE QUELQUE MANIERE ET SOUS QUELQUE FORME QUE CE SOIT POUR METTRE DES EXEMPLAIRES DE CETTE THESE A LA DISPOSITION DES PERSONNE INTERESSEES.

THE AUTHOR RETAINS OWNERSHIP OF THE COPYRIGHT IN HIS/HER THESIS. NEITHER THE THESIS NOR SUBSTANTIAL EXTRACTS FROM IT MAY BE PRINTED OR OTHERWISE REPRODUCED WITHOUT HIS/HER PERMISSION.

L'AUTEUR CONSERVE LA PROPRIETE DU DROIT D'AUTEUR QUI PROTEGE SA THESE. NI LA THESE NI DES EXTRAITS SUBSTANTIELS DE CELLE-CI NE DOIVENT ETRE IMPRIMES OU AUTREMENT REPRODUITS SANS SON AUTORISATION.

ISBN 0-612-01380-4

Canada

Abstract

EVALUATING ERROR RECOVERY MECHANISMS IN THE XPRESS TRANSFER PROTOCOL

AYUB ASAMBA

The objective of this project is to use a real time environment to verify the error recovery mechanism of the Xpress Transfer Protocol (XTP) version 3.6, which has been specified in the Estelle specification language.

Two modules were built to interact with the XTP Estelle specification module. At the bottom of the XTP module, is a medium module that makes use of the Internet Protocol to provide an unreliable transmission medium. The loose source routing strategy is implemented to guarantee the unreliabilty of the transmission medium. On top of the XTP module is the user module, which uses the services of XTP to transfer and receive data reliably across the unreliable underlying medium.

ACKNOWLEDGEMENT

I wish to express my sincere gratitude to my supervisor, Dr. J.W. Atwood for his professional guidance throughout the course of the project.

I also acknowledge the technical support that I got from the system administrator. Special thanks go to the team members of the High Speed Protocol Laboratory (HSPL), for their encouragement and helpful discussion.

Financial assistance from the Kenya Government in conjunction with the Canadian International Development Agency (CIDA) is greatly appreciated.

Finally, i gratefully acknowledge the moral support that i got from my family.

Contents

List of Figures

1 INTRODUCTION

The transport layer protocol in the OSI model provides a reliable data delivery service to the application layer. It defines error recovery mechanisms that ensure the data being delivered is in sequence and without duplication.

The design of some of the extant transport protocols, such as the Transmission Control Protocol (TCP), followed a pessimistic approach, based on the fact that the underlying transmission network was slow and had a high bit error rate. With the advent of high speed networks with low bit error rates, a more optimistic design approach to the error recovery mechanism is necessary in order to maximize the throughput of such high bandwidth. The Xpress Transfer Protocol (XTP) [6] is a new protocol designed to meet such high efficiency. This new design approach must also be shown to work in the same more traditional environment.

The Institut National des Télécommunications (INT) of France, carried out an experiment to validate the error recovery mechanisms of XTP. The experiment was based on a specification of XTP, version 3.6 that had been written in the Estelle specification language [7]. In Estelle, XTP is represented as an XTP module. The internal structures of the XTP module are also represented as modules. The major XTP internal modules are the context module and the router module. The context module represents the functionalities of the transport layer, while the router module represents the functionalities of the network layer (routing packets through the network). These modules exchange interactions through interaction points, which define the endpoints of a communication channel. Each interaction point is associated with a queue that stores and delivers data in a FIFO fashion.

The experiment involved two context modules, which communicated with each other via a medium module (representing the transmission medium). Since the simulation was carried out within a single process, it was not feasible to have a scenario where a packet being transmitted from one context to the other got lost or reordered by the medium module.

To provide an approximation of an unreliable transmission medium in such a closed environment, the INT specification manipulated the queues associated with the medium module's interaction points, by dequeueing (packet loss) and reordering the packets.

In the current project, the error recovery mechanism of XTP is validated by using a real environment transmission medium. The test configuration consisted of two XTP specification modules placed on two distinct Sun workstations and made to run on top of the Internet Protocol (IP). The IP layer provides a connectionless and unreliable delivery system. It is connectionless because it treats each IP datagram independently of all others. It is unreliable because it does not guarantee that the IP datagram will ever get delivered or be delivered correctly [5].

Since the two Sun workstations are located in the same room, the probability of packets being lost or reordered is negligible. To guarantee an unreliable transmission path between the two workstations, the Loose Source Routing option of IP was used. This option increases the number of intermediate routers that the packets have to traverse. For example, two machines A and D placed in the same locality, can use this routing strategy to provide the following path for the packets

$A(Montreal) \longrightarrow B(Toronto) \longrightarrow C(California) \longrightarrow D(Montreal)$
and
$A(Montreal) \longleftarrow B(Toronto) \longleftarrow C(California) \longleftarrow D(Montreal)$
instead of
$A(Montreal) \longleftrightarrow D(Montreal)$.

The structure of the XTP packets in an Estelle specification is represented in logical form. The transmission of an XTP packet through a physical medium, by encapsulating it in an IP datagram, requires the packet to be in a particular physical representation, as defined by the Protocol Definition document [11]. An interface module was therefore built between the

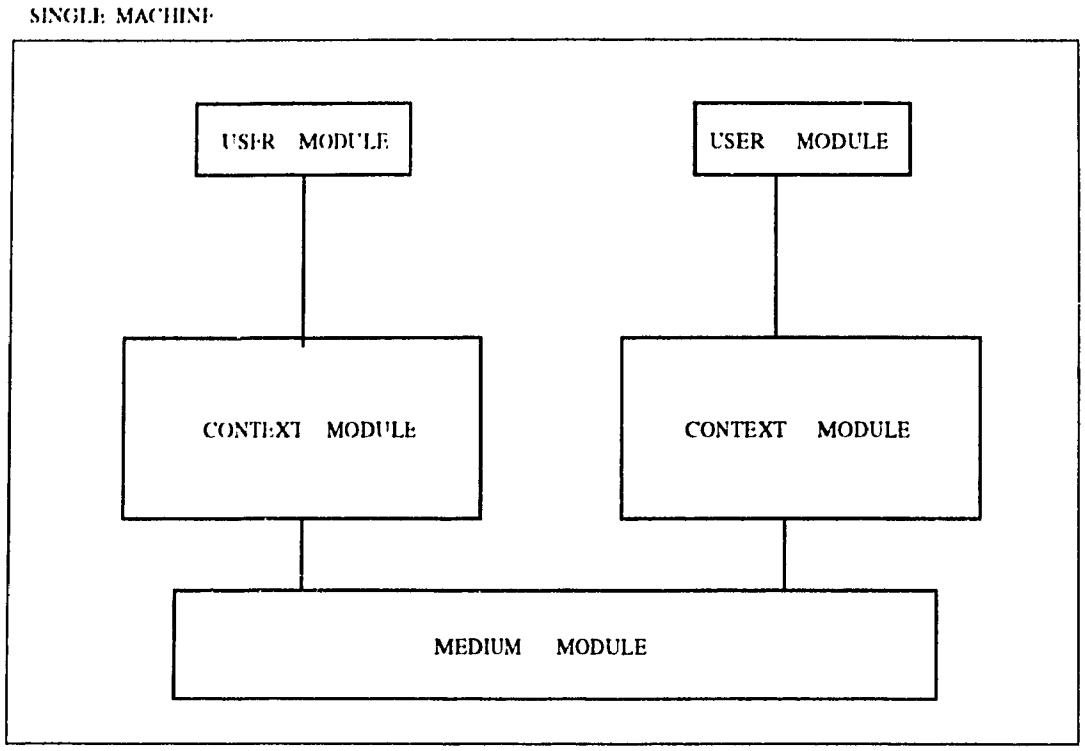


Figure 1: INT TEST CONFIGURATION

XTP module and the Internet Protocol (see page 62). This interface module consists of a library of C-code routines (see page 64) that perform the function of transforming packets designated for transmission from the logical to the physical representation. Packets received from a physical medium through an IP socket are transformed into logical form before being channelled to the XTP module via an interaction point. Figures 1 and 2 show the disparity in test configuration between that used by INT and that used by this project.

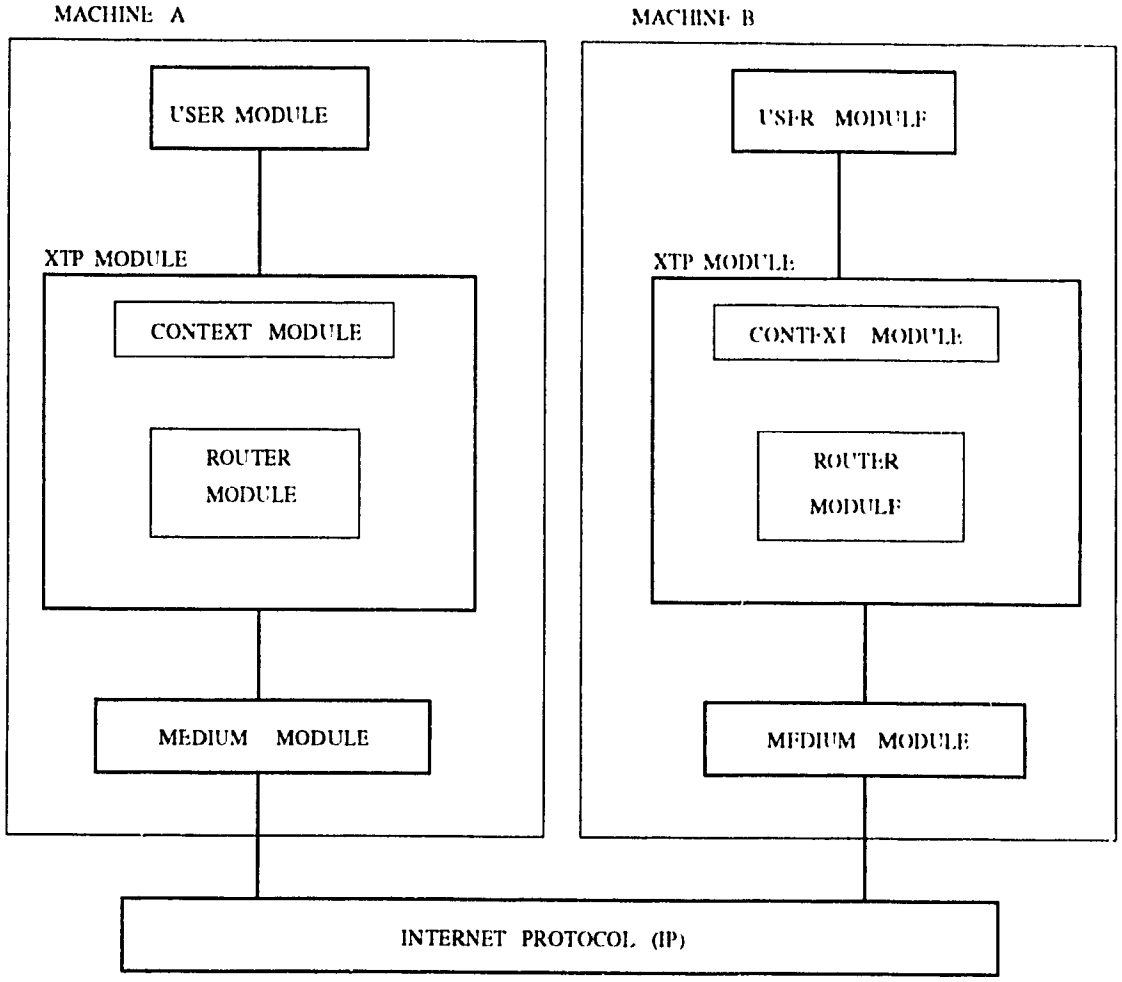


Figure 2: THIS PROJECT TEST CONFIGURATION

2 OVERVIEW OF XTP AND ESTELLE

2.1 XTP OVERVIEW

XTP is a high performance protocol, whose design was based on the mechanisms and concepts of extant protocols such as TCP, DELTA-T, VTMP, etc.

The design of this new protocol was motivated by the fact that extant protocols could not meet the demands that were taking place both from above and below the transport layer in the OSI model. From above, new application domains such as distributed systems and multimedia workstations demanded new services from the communication subsystem. For example, the shift towards distributed computation and demand paging requires communication services that rely heavily upon request/response interaction [11]. The 3-way handshake required by TCP to set up a reliable connection is too costly and unacceptable for the transaction communication that is required for request/response interaction.

From below, the transport layer must respond to the drastic advances in network hardware that is moving towards high bandwidth with lower bit error rate medium, from 10Mbit/s ethernet to 100Mbit/s FDDI (fiber distributed data interface) and later 1Gbit/s.

As it became apparent that the network software cannot keep pace with such speeds, it was necessary to incorporate the functionality of the network layer in designing the new protocol that can easily be implemented in VLSI. Without such drastic changes in the design, mere improvement on the existing protocols would not be able to catch up with the network media and the bandwidth available to the application would remain essentially the same.

Experiments have shown that protocols require more processing per packet at the receiver than at the sender and hence the receiver is the bottleneck in protocol processing [11]. In high speed networks and with the XTP goal of processing packets at the speed of the underlying high speed media, the receiver must not be overburdened with the amount

of processing it has to do. XTP reduces the burden on the receiver by making it operate as the slave of the transmitter. The receiver is required to issue a control packet only when its transmitter sends a status requesting command in a packet header. This master/slave relationship has the significant advantage of enabling the sender to request control packets at synchronization points independent of patterns determined by data acknowledgement.

TCP adopts a positive acknowledgement retransmission (PAR) scheme using the *go back n* retransmission algorithm. Under PAR, the receiver only acknowledges data received error free and in sequence. Retransmissions are triggered by timeout at the sender for unacknowledged data. *Go back n* retransmits all data starting from the point of last acknowledgement. In high bandwidth networks where the number of packets in transit can be very large, using the *go back n* retransmission algorithm can be very inefficient.

In XTP, the sender requests a status report from the receiver, either periodically, or when the flow control window has closed, and then bases its retransmissions on the information contained in this report. The report is informative enough to allow *selective retransmission.*

Other useful features that give XTP an edge over the existing protocols are mentioned in the section that dwells on the functionalities of the relevant fields in the XTP packets.

2.2 ESTELLE OVERVIEW

Estelle is a formal specification language that can be used for describing distributed concurrent process systems [12]. In particular, Estelle is used for describing the services and protocols of the open system interconnection (OSI) architecture defined in the ISO 7498 standard.

An Estelle specification describes a collection of components. Each component is an instance of a module defined within the Estelle specification by a module definition. The

behavior of the module and its internal structure are specified by the set of transitions that the module may perform, and by the definition of submodules, if any, together with their interconnection.

A module may have a number of input/output access points (communication interfaces) called interaction points. A channel associated with the two interaction points defines the set of interactions that can be sent or received or both. The exchange of messages is called interaction.

A module instance can send an interaction to another module instance through a previously established communication link between the two interaction points. An interaction received by a module instance is appended to an unbounded FIFO queue associated with this interaction point. A FIFO queue could exclusively belong to a single interaction point or could be shared with some other interaction points. For the former, the queue is referred to as an individual queue and for the latter, as a common queue. The modules communicate in asynchronous mode. In other words, they are non blocking, they can always send an interaction.

2.2.1 ESTELLE COMPILER

An estelle compiler translates an Estelle specification into C - language source code. The compiler consists of the translator and the C code generator tool.

The *translator* takes an Estelle specification as its input, and perform a complete static anaysis (lexical, syntactical and semantic) of the specification. If no errors are found, it generates a specification representation in the so-called intermediate form (IF).

The *C-code generator* takes the intermediate form as its input and returns the C-code of the specification.

```
> ec filename.stl
```

2.2.2 SIMULATOR/DEBUGGER (EDB)

EDB is an Estelle symbolic and interactive simulator/debugger [12]. Its purpose is to help the user in discovering and processing errors that occur during the execution of an Estelle specification. It enables the user to concentrate on the properties that he wants to verify. The user of the EDB may conduct the simulation in two different ways.

1. STEP BY STEP SIMULATION

   In this mode, after a single transition has fired, EDB issues a prompt for the user to enter the next command. The user has the option of choosing the next transition to fire from among the fireable transitions or examining all the information that constitutes the global state of the specification under simulation. The user may also modify the values of objects such as local variables and the content of the FIFO queues associated with one end of a channel. This last facility may be used to simulate reordering or loss of packets by an unreliable medium.

   To select the next transition to fire, the following command is issued

   edb> $se

   where se stands for select executable transition.

   A list of fireable transitions will be displayed, each prefixed by a number. These numbers are used to select the desired transition. Once the transition has been selected, the following command is issued to fire the transition

   edb> c

   where c stands for continue.

   The following commands are used to display the content of the interaction processed by the last transition to fire.

- To display the content of the output.

edb> d$ltro

where ltro stands for last transition output.

- To diplay the content of the input

edb> d$ltri

where ltri stands for last transition input.

- To identify the transition that fired last.

edb> d$ltrid

where ltrid stands for last transition identification.

- To display the hierarchical structure of the specification modules.

edb> d$h

where h stands for hierarchy.

- To display the interactions in a queue of a given interaction point.

edb> d$q(n->r)

where n is the submodule number in the hierarchy, r is the interaction point.

2. RANDOM SIMULATION

In this mode of simulation, the simulation goes automatically for a given number of computation steps or a predefined amount of time. At each step, EDB selects randomly

from among the fireable transitions, the next transition to fire.

In this project, we used our own intuition to select a number that was large enough to allow a user module (transmitter) to transfer a reasonably large file to the corresponding user module (receiver). This number was set by using the following commands.

```
edb> $fs := number;
```

where fs stand for firing steps.

```
edb> c
```

3 XTP PACKET STRUCTURE

XTP defines 9 packet types, which are represented in physical form. These packets are within 2 formats, a control format and an information format. The packets are made up of three segments: the header segment, the middle segment and the trailer segment. The header and the trailer segments are the same for all the packets. The middle segment can be a control segment or an information segment. Figure 3 shows the structure of these segments.

The header has a fixed length of 40 bytes and the trailer a fixed length of 4 bytes. The header size is twice as long than that of TCP because of the fact that XTP takes into account both the transport and network functions. Another reason is that all XTP control fields are 32 bits aligned. The middle segment is of varied length and a multiple of 4 bytes. The information segment consists of 4 segments.

The control segment provides the mechanisms to communicate state information from one protocol machine to another. Such information includes data reception status, rate and flow control parameters, context and route identifier, and values used to synchronize the state machines. The state machines exchanging this information may be at the endpoints of an association or within the switches between the endpoints.

The information segment provides mechanisms for communication of user or protocol data rather than protocol state information. The data provided by the user is carried from end to end without being interpreted or inspected. Protocol data such as diagnostic information, may be inspected or examine by the protocol as well as given to the user. Also included within the information segment is the mechanism for providing address information.

3.1 XTP HEADER.

Figure 4 show the structure of the XTP header segment

- ROUTE FIELD.

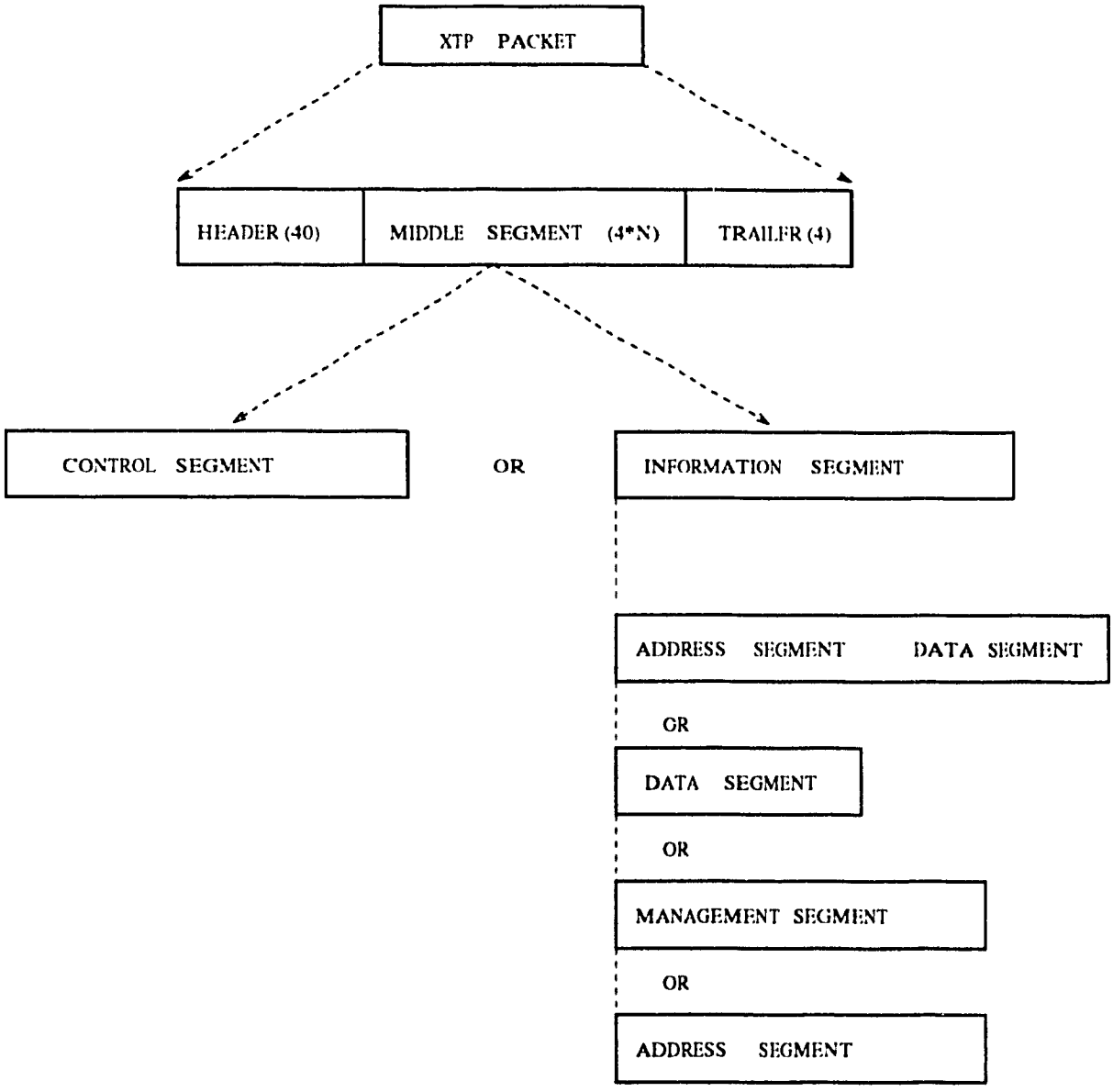
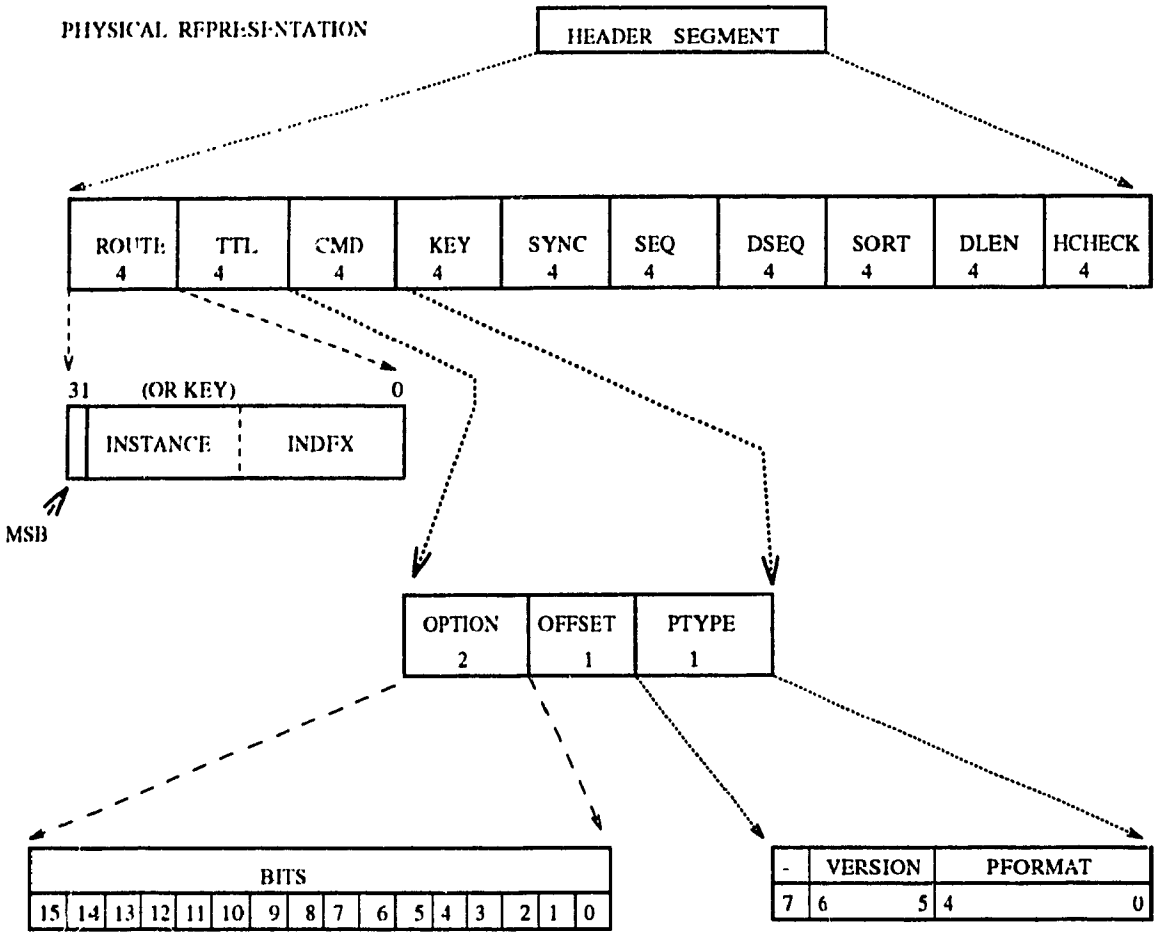


Figure 3: XTP SEGMENTS



BIT FIELD KEY:

0	BTAG	1	END	2	EOM	3	WCLOSE
4	RCLOSE	5	DREQ	6	SREQ	7	FASTNAK
8	NOFLOW	9	SORT	10	RES	11	MULTI
12	NOERR	13	-	14	NOCHECK	15	-

Figure 4: XTP HEADER

XTP uses a technique called *cut through* switching, which has been designed specifically for compatibility with the ATM (asynchronous transfer mode) and emerging gigabit networks.

As the FIRST packet cuts through the network using the address information in the address segment, it leaves behind a trail of route values at each intermediate node along the path. Subsequent packets moving in the forward direction route themselves through the network by using the trail of route values that were left behind by the FIRST packet.

The use of route values to route a packet, by indexing into a route table, eliminates the overhead involved in processing a full destination address. A route value is a 32 bit unsigned integer with the most significant bit (MSB) cleared. The route value is a return route when the MSB bit is set. The destination host uses the return route value for sending response packets back to the source.

- TTL FIELD (time to live).

The TTL field indicates the amount of time that the packet has to remain valid while in the network. As a packet traverses the network, it may become lost or misdirected. A mechanism is required to stop such packets from meandering through the network. The TTL field is set by the originator with a value that is expressed in 100nsec clock ticks. This field is examined and modified as a packet transits an intermediate system such as a switch or router.

A TTL value of zero disables the mechanism. The packet never times out. For non zero values, the intermediate node decrements the value by an amount equal to the expected network propagation and latency time. If the TTL value becomes zero or negative, the packet is discarded. Otherwise, it is forwarded to the next switching

destination.

- CMD FIELD (command).

  The CMD field encodes four kinds of information within three subfields

  1. protocol options
  2. number of offset bytes before the beginning of user data
  3. the format of the remainder of this packet and
  4. the version number of the XTP protocol being used.

OPTION SUBFIELDS.

The option subfield is a 16 bit field with 14 bits defined. These options control many service features of the protocol. They select the XTP operation modes and mechanisms. A function is enabled if the corresponding bit is set to one and disabled if the bit is zero.

The fields that are meaningful to this project are as follows

1. NOCHECK BIT.

   When set, the receiver does not perform a checksum for the middle segment. This bit has no effect on the checksum for the header, which is always enabled. The bit could be set for efficiency reasons.

2. NOERR BIT.

   When set, it inform the receiver that the sender will not retransmit and directs the receiver to disable error correction processing. The receiver ignore any input error. If the sender requests a control packet response by setting the SREQ or DREQ bit, then the receiver must acknowledge the highest received sequence number. The sender ignores nspan and spans on the received CNTL packet.

   The use of this bit allows XTP to provide two types of services to the upper layer. When the bit is set, the protocol provides unreliable services equivalent to

those provided by UDP. When the bit is off, the protocol provides reliable services equivalent to those of TCP.

3. MULTI BIT.

   When set, indicates the multicast mode is on. This feature provides a mechanism for group communication, which is useful in distributed database systems and teleconferencing.

4. NOFLOW BIT.

   When set, indicates that the sender does not observe sequence based flow control. This mode of operation is used on a rate controlled connection or in situations where unconstrained operation is desired. A receiver may decline to accept NOFLOW mode by rejecting the FIRST packet with a DIAG message.

5. FASTNACK BIT (fast negative acknowledgement).

   When set, directs the receiver to generate a fast negative acknowledgement immediately if out of sequence data are detected. This is an aggressive strategy designed for those networks that either never or seldom loose or reorder packets, and hence out of sequence data implies that some data have been lost. The value of this field is ignored if the NOERR bit is set.

6. SREQ BIT.

   When set, requests the receiver to respond immediately with a CNTL packet.

7. DREQ BIT.

   Request the receiver to respond with a CNTL packet after it has delivered any already received data to the higher layer application.

8. WCLOSE, RCLOSE and END.

   These bits indicate the progression toward association termination. The WCLOSE bit indicates that the sender's Write process is closed and hence no more new data will be sent. Retransmissions if necessary are still allowed even after the Writer closes but the sequence number associated with this data stream will not advance. Upon receipt of the WCLOSE bit, and after obtaining any packets that needed to be retransmitted, the receiver sets the RCLOSE bit in outgoing packets. This indicates the sender's reader process is closed and hence no new data will be read. The END bit indicates that the context sending is being relinguished and no other communication of any kind is allowed.

   A context that sets both WCLOSE and RCLOSE bits indicates that it has shut off all user data exchange functions and cannot respond to packets with new data.

9. EOM BIT.

   This bit marks the end of the message. The bit is ignored in packets other than the FIRST and DATA packets.

10. BTAG BIT.

    Indicate that the first 8 bytes of the data segment within the information segment contain tag information for the higher layer application. It is meaningful only in FIRST or DATA packets.

- OFFSET FIELD.

  This field indicates the number of bytes of padding that come before user data when data is present in an information segment.

- PTYPE SUBFIELD.

  This subfield contain two types of information. The pformat subfield occupies the 5 lower bits and specifies the format of this XTP packet. The next 2 bits specify the XTP version number.

- KEY FIELD.

The key field is a 32 bit field with the most significant bit reserved as a flag. It is used to uniquely identify the context that will serve as one end point of an association. The 31 bits are further subdivided into an instance and an index. The index selects the context by indexing into the context table, and the instance is used to validate the index value and discriminate against inactive index values. It is an end to end value communicated unchanged through switches to a destination end point.

- DSEQ FIELD.

DSEQ indicates the sequence number of the next byte expected to be delivered to the user. XTP receiver uses dseq to indicate to the transmitter what data it can free from the buffer space without fear that the data will ever be requested for retransmission.

- DLEN FIELD.

The data length field indicates the number of bytes present in the middle segment. It includes the offset bytes but not the alignment bytes.

- SEQ FIELD (sequence number).

The sequence number field is the mechanism by which bytes within a data stream are correctly ordered and identified. Each byte in the data stream is assigned a sequence number to uniquely identify that byte in the data stream in order to ensure proper sequencing.

For data bearing packets, the field specifies the first byte of data in DATA packets, and the first byte of address information in the FIRST packet. For control packets, the SEQ field is used to identify the next byte expected to be sent on the outgoing

data stream.

The seq field is 32 bits wide, allowing up to a maximum of 2^{32} - 1 bytes to be in the data transmission pipeline. TCP has a sequence number of 16 bits, which restricts the number of bytes that can be in the pipeline. This has the effect of reducing transmission bandwidth utilization in a high speed network.

- HCHECK FIELD.
  The HCHECK field contains the result of a mandatory checksum calculation over the header fields excluding the ROUTE and TTL fields.

3.2 XTP TRAILER FIELD

The XTP trailer contains the result of an optional checksum calculation over the middle segment of the XTP packet. Its value is ignored if the NOERR bit is set.

3.3 MIDDLE SEGMENT

3.3.1 CONTROL SEGMENT

Packets containing a control segment as their middle segment are called control packets. Control packets are used to exchange state information between endpoints.

- CONTROL SEGMENT FIELDS.

1. RATE AND BURST FIELD (rate control procedure).
   The rate value specifies the maximum number of bytes per second that can be consumed by the receiver. The burst value specifies the maximum number of bytes that can be consumed in one burst of packets, that is packets sent in rapid succession. XTP

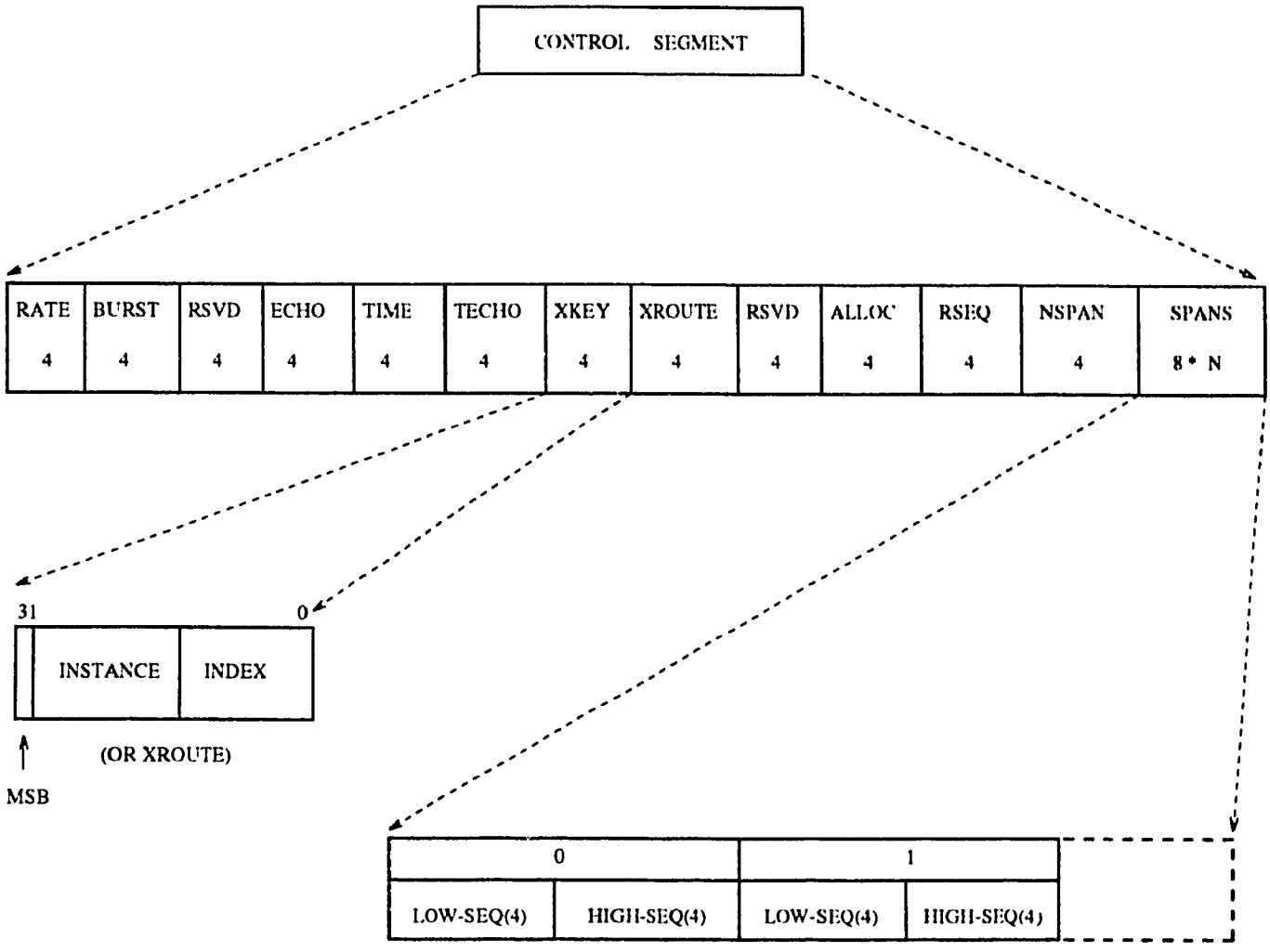


Figure 5: CONTROL SEGMENT

defines RTIMER (refresh timer) variable, which is initalized by the value obtained in dividing burst by rate. The CREDIT variable is instantiated with the burst value each time the RTIMER is reset. As packets are transmitted, CREDIT is decremented by the amount of data sent. When the value of CREDIT becomes zero, the transmission of data is suspended until RTIMER expires. RTIMER is then refreshed and the CREDIT renewed. A value of zero for burst means transmission is unconstrained. A value of zero for rate halts transmission.

2. ECHO FIELD.

   This field along with the SYNC field of the header is used to synchronize protocol state machines. The receiver copies the SYNC value into the ECHO field of the next outgoing CNTL packet.

3. TIME and TECHO FIELDS (time synchronization).

   The sending context places the current time in the TIME field of each outgoing CNTL request packet. When the receiver receives a CNTL packet with the SREQ bit set, the value in the TIME field is copied into the TECHO field of the outgoing CNTL response packet. When the sending context receives the response CNTL packet, it subtracts the value of TECHO field from the current time to get the estimated round trip time for packets.

4. XKEY and XROUTE FIELDS (key and route exchange).

   The XKEY field is used to inform the receiver of this CNTL packet to use this key value as the return key field in all outgoing packets. The use of this return field to identify a context has the effect of improving performance. The XROUTE field is used similary to improve routing performance.

5. ALLOC FIELD.

   This field conveys to the transmitter the highest sequence number that the receiver is

able to consume. The value is set according to the amount of buffer space available. As the receiver frees the buffer space, the ALLOC value increases. It is used in the flow control procedure to set the upper edge of the transmission window.

6. RSEQ FIELD.

   The receiving side uses RSEQ value in CNTL packet to acknowledge receipt into buffer space of contiguous data on an incoming data stream. If the transmitter side receives a CNTL packet with the RSEQ value equal to the sequence number of the next byte to be transmitted, then the transmitter infers that no retransmissions are required. If the value of RSEQ is less, then the transmitter infers that the receiver has not received all the data sent and some retransmission may be necessary.

7. NSPAN AND SPANS FIELDS (selective retransmission).

   The SPANS field tells the transmitter which contiguous group of data (pairs of lowest and highest sequence numbers) have been received after the first gap in the data stream. NSPAN field indicate how many of these contiguous groups are present in the SPANS field. The gaps determined from the two fields are used to provide information for selective retransmission.

3.3.2 INFORMATION SEGMENT

This segment provides a mechanism that allow users to transfer data between two communication end points.

- DATA SEGMENT.

This segment is always present in DATA packets and optionally present in FIRST packets. The length of this field is limited only by the maximum frame size of the underlying network. If the BTAG bit in the option field is set, the tagged data occupy the first 8 bytes

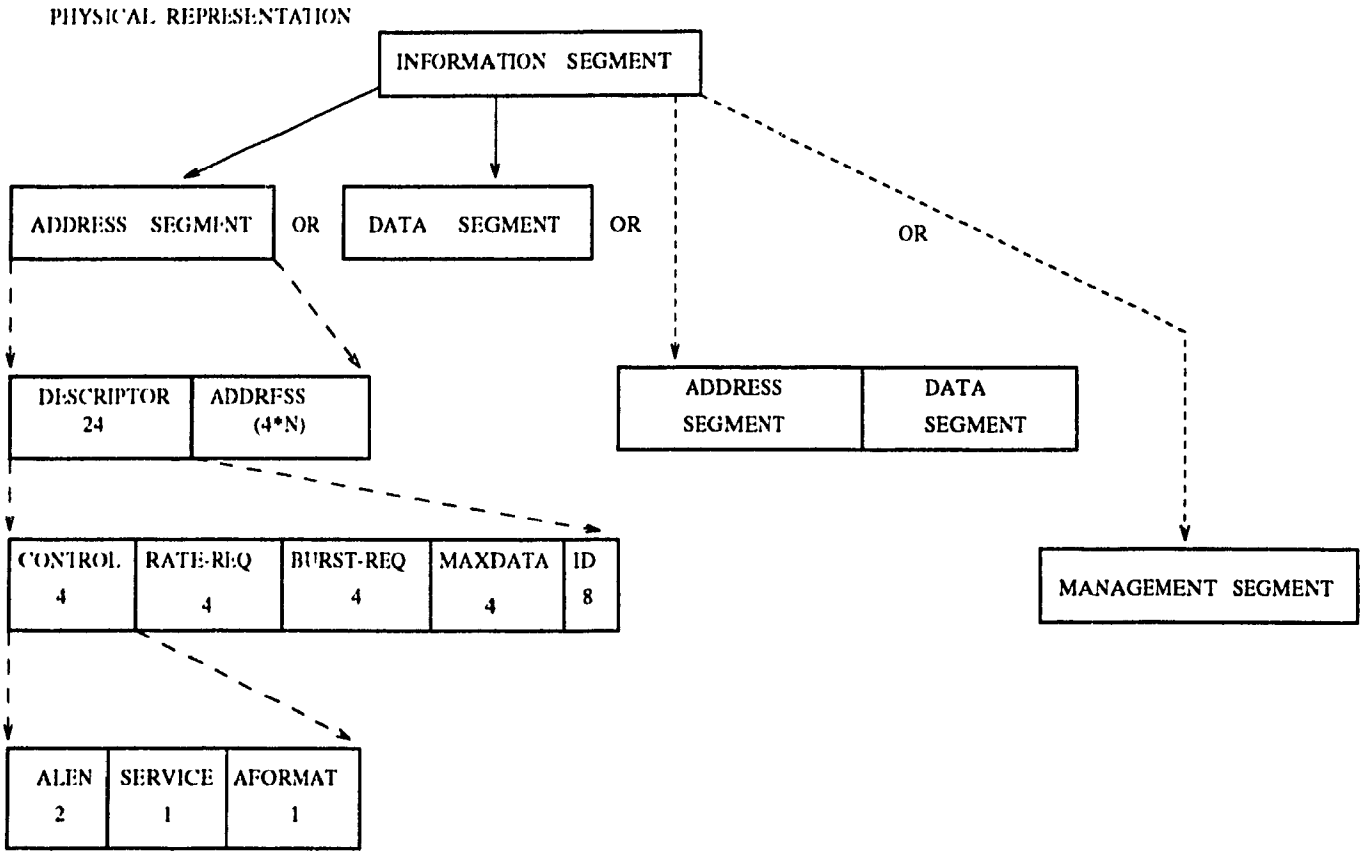


Figure 6: INFORMATION SEGMENT

of the data segment. The BTAG field is used to convey higher layer control information. For example, in an intergrated service data transfer where a single data stream must switch between video, voice and data, each of these types of information can be proceeded with a BTAG that indicates the type of data that follows.

- ADDRESS SEGMENT.

The address segment holds all the information necessary to deliver a packet from one end to another end across the network. The segment is included in both the FIRST and PATH packets. It enables the FIRST packet to establish both the forward and return path through the network. XTP does not have its own addressing scheme, instead it uses a parametric addressing scheme that supports the address format of several widely used address schemes [1].

The MAXDATA field indicates the length in bytes of the maximum information segment that the initiating context expects to transmit during the lifetime of the association. Intermediate switches can reject this value by sending a diagnostic packet containing the maximum size that the switch can forward. XTP avoids the processing overhead involved in fragmenting and reassembling of packets by using this field to determine the smallest MTU (maximum transmission unit) along the route.

3.4 XTP PACKETS

- FIRST PACKET.

This packet is used only once, during the establishment of an association. As the *first* packet threads its way through the network switches, it establishes a bidirectional path for all subsequent packets in the association. Upon receipt of the *first* packet, the receiver activates a context, which, together with the context of the transmitting endpoint, defines an association.

- CNTL PACKET.

  Its function is to exchange state information between the interacting contexts within an association.

- PATH PACKET.

  This packet is used to rethread a path if during the lifetime of an association, the original path become undesirable or unavailable. It is also used in a multicast association, to allow a receiver to join an in-progress data transfer.

- DIAG PACKET.

  This packet is used for diagnostic messages. It contains an information segment that reports errors encountered by the protocol machine while trying to process a packet.

- ROUTE PACKET.

  This packet is used for path release. The packet uses the management segment to signal to all switches along the path that the path is being released.

- RCNTL PACKET.

  This packet allows communication of state information between XTP switches.

- DATA PACKET.

  This packet carries user data only.

- MGMT AND MAINT PACKETS.

  These packets are not yet defined.

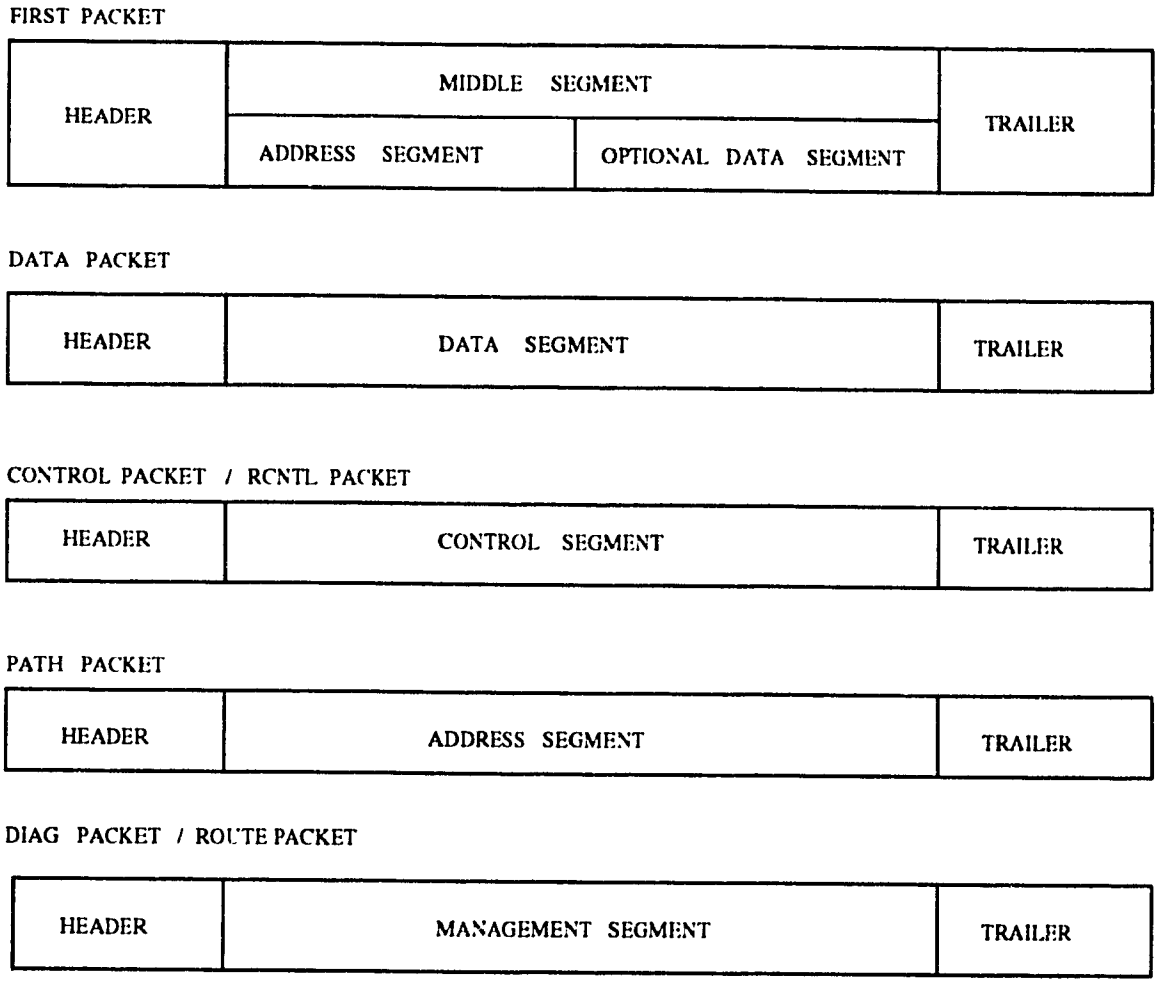


Figure 7: XTP PACKET LAYOUTS

4 XTP ERROR RECOVERY MECHANISM

Error recovery mechanisms outline the functionalities within XTP protocol that facilitate reliable data transfer between XTP users. The mechanism make use of the timers, sequence number, checksum and synchronizing handshake in detecting errors and if possible recovering from them through retransmission.

As packets are being transmitted from one end point to the other, it is possible for the bits within the packet to get corrupted. In order to preserve the integrity of the packet, a checksum is applied on each packet. XTP defines two fields in each packet for checksum purposes. The field used to preserve the integrity of the control information in the XTP header is called the HCHECK field. The DCHECK field in the trailer is used to preserve the integrity of the middle segment.

The checksum over the header is mandatory because compromizing the integrity of the control information fields in the header could lead to a more dangerous operation that might be difficult to recover from than recovering from the loss of the packet itself. At each intermediate switching node, the checksum for the header is recalculated and compared with the value stored in the HCHECK field. A mismatch means the packet has been corrupted and should be discarded. This ensures that each node deals with a valid packet.

The validity of the middle segment depends on the value of the NOCHECK bit in the option field of the header. If the bit is set, the checksum is disabled. The integrity of the middle segment is an end to end issue.

Timers are another aspect of the error control procedures that are used to detect the non occurrence of a specific event, mostly a lost control packet. The following example illustrates how a lost control packet could lead to a deadlock.

Consider a scenario where the transmitter has sent all the data in the flow control

window and has issued a request for a response control packet by setting the SREQ bit in the outgoing packet. Assume the packet gets lost before it is received by the receiver or it was received, but the transmitted response control packet gets lost on the way. Since the XTP protocol defines the receiver as the slave, the two end points will wait for ever.

The use of timers in error recovery help in detecting and recovering from such a scenario. There are 3 timers defined in the protocol. The WTIMER (wait timer) specifies the amount of time a context has to wait for a CNTL response packet after a request packet with the SREQ bit set has been issued. When WTIMER expires before the requested response CNTL packet arrives, the transmitter assumes the request packet got lost and it therefore initializes a synchronizing handshake. This type of handshake involves the exchange of control packets between the two contexts in order to determine the state of the receiver. During this period, the transmission of data packets is suspended until the handshake is completed.

CTIMER signals the inactivity on the association. It is enabled when loaded with a value greater than zero. When CTIMER expires and no packet has since been received, the context is forced to initialize a synchronizing handshake. CTIMER is reloaded and started again. The value of this timer should be sufficiently large so that if it expires with no packets received, it would indicate with high probability a possibility of a broken path or a dead end point.

In addition to timers, there are variables that play an important role in error recovery. Retry-count variable is used in conjunction with the CTIMEOUT timer to provide an upper limit on the number of retries in message exchange. RTT provides an estimate of the round trip time across the network.

When the context enters a synchronizing handshake, the CTIMEOUT is enabled, the exponential back-off constant (k) is set to one, the retry-count is set to some value greater than zero and the new waiting time (wt) is set to the product of the WTIMER and k.

wt = k * WTIMER.

If the remote context is alive, an indication that the association was successfully initialized by the FIRST packet, a CNTL request packet is retransmitted or else the FIRST packet with SREQ bit set is retransmitted. The context then sets the waiting time limit for the reply to the value of wt.

If the requested response CNTL packet arrives before the waiting time (wt) expires, the synchronizing handshake is considered to have completed successfully. Conversely, if the time expires and the requested response control packet has not arrived, the transmitter assumes the request packet could have become lost due to congestion in the network. It therefore doubles the value of the exponential back-off constant, recalculates the new waiting time and decrements the retry-count by one before repeating the retransmission and waiting process. If the retry-count reaches zero or CTIMEOUT expires before the requested response control packet arrives, the context is aborted.

RTT is used to load the WTIMER with the value that is smoothed over multiple RTT observations. It ensures WTIMER is dynamically assigned with values that reflect the true congestion in the network. It therefore reduces the possibility of unnecessary retransmission, which could be caused by values smaller than network delay or unneccessary retransmission delays due to larger values.

XTP uses a sequence numbering scheme to detect lost data packets. Each data byte being transmitted is assigned a sequence number that is monotonically increasing. The sequence number for the first data byte in the data carrying packet is stored in the seq field in the header. At the receiver end, the receiver will continue delivering the data to the XTP user as long as it is being received in sequence.

Once a gap is detected in the sequence. the receiver suspends the delivery of data to the upper layer until the lost packet is received. Meanwhile, the receiver continues to buffer

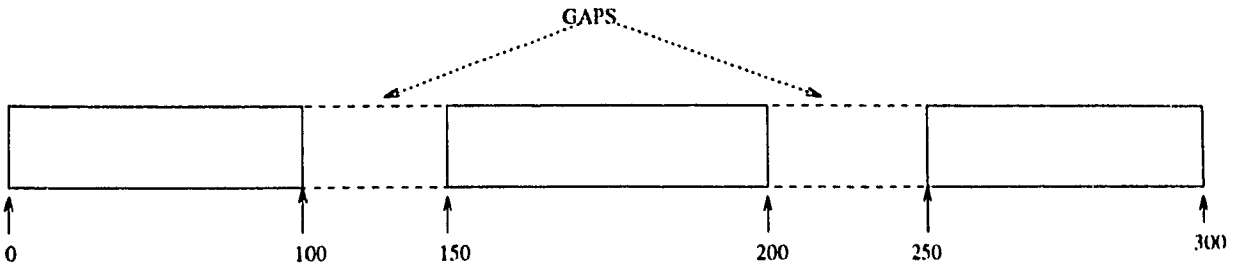


Figure 8: DATA STREAM WITH GAPS

the data being received. When a request packet is received, the receiver generates a response control packet with the neccesary information that will facilitate selective retransmission of the lost packets. The fields in the control packet that are involved in this exercise are the RSEQ, SPANS and NSPAN. RSEQ represents the first byte of the first gap in the data. NSPAN represents the number of contiguous data groups received and SPANS gives the list of pairs of the sequence numbers for the first and last bytes in each of the contiguous groups.

Figure 8 is a hypothetical example of an input data stream with gaps at the receiver end and the values assigned to these fields in the response control packet.

RSEQ = 100

NSPAN = 2

spans[1].lowest_seq = 150

spans[1].highest_seq = 200

spans[2].lowest_seq = 250

spans[2].highest_seq = 300.

Given this information the transmitter will retransmit two DATA packets with the following information in the seq and dlen fields

```
first packet :  seq = 100, dlen = 50
second packet : seq  = 200, dlen = 50.
```

To impose the *go back n* strategy, the receiver assigns the nspan field a value of one and set both fields in the spans to the same value of (hseq,hseq), the highest received sequence number.

5 INTERNET PROTOCOL

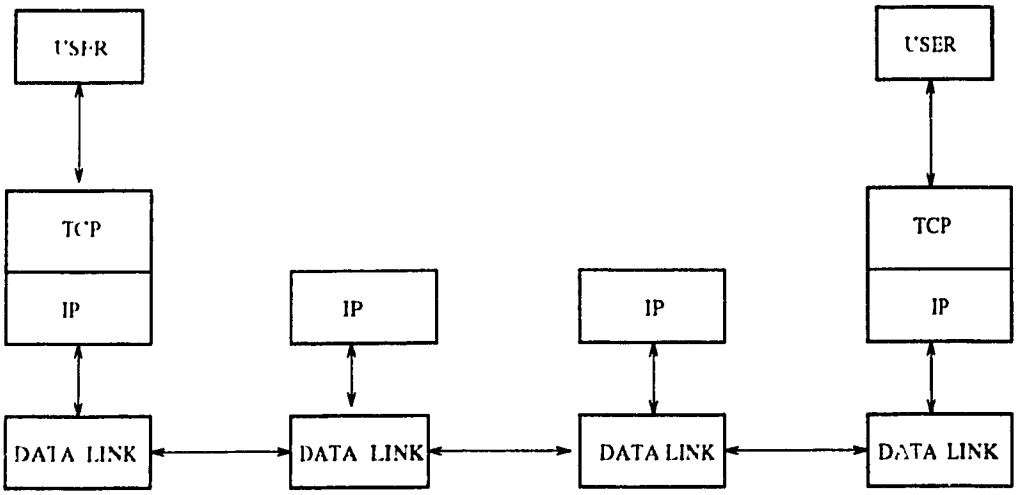
An internet is a connection of two or more distinct networks so that computers on one network are able to communicate with computers on another network. One way to connect two distinct physical networks is to have a gateway that is attached to both networks. This gateway is sometimes called a router. A router operates at the network layer. Its functionality is to move packets from one network to another.

In an internet that uses the TCP/IP family suite, the Internet Protocol(IP) [5] layer is responsible for the forwarding of packets. Each IP packet contains enough information (ie., its final destination address) for it to be routed through the TCP/IP internet. Figure 9A shows an example of two user processes interconnected with IP.

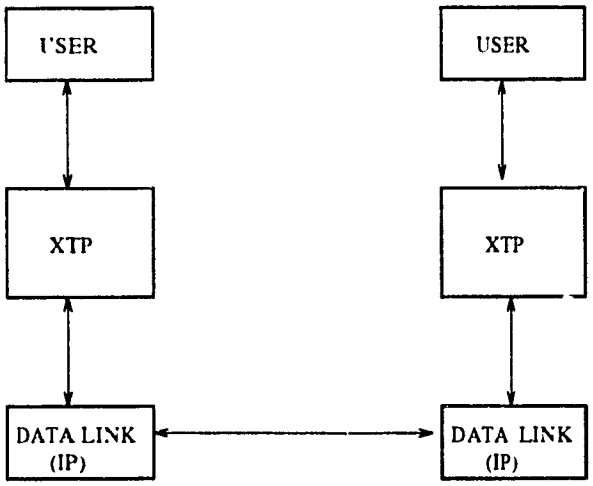
The internet protocol uses four key mechanisms in providing its services: type of service, time to live, options and header checksum. The options is only useful when testing new protocol or network reachability, otherwise it is unnecessary for common communication. Figure 10 shows the internet datagram header together with the internal structure of the options field. Without the options, the size of the IP header is only 20 bytes. The options that are defined include the

1. `loose source routing`
2. `strict source routing`
3. `time stamp`
4. `record route`
5. `internet timestamp`

- NOTE: since the Internet Protocol multiplexes several upper layer protocols, for safety reasons, its accessibility is limited to the superuser only.



(A)



(B)

Figure 9: TRANSMISSION PATH

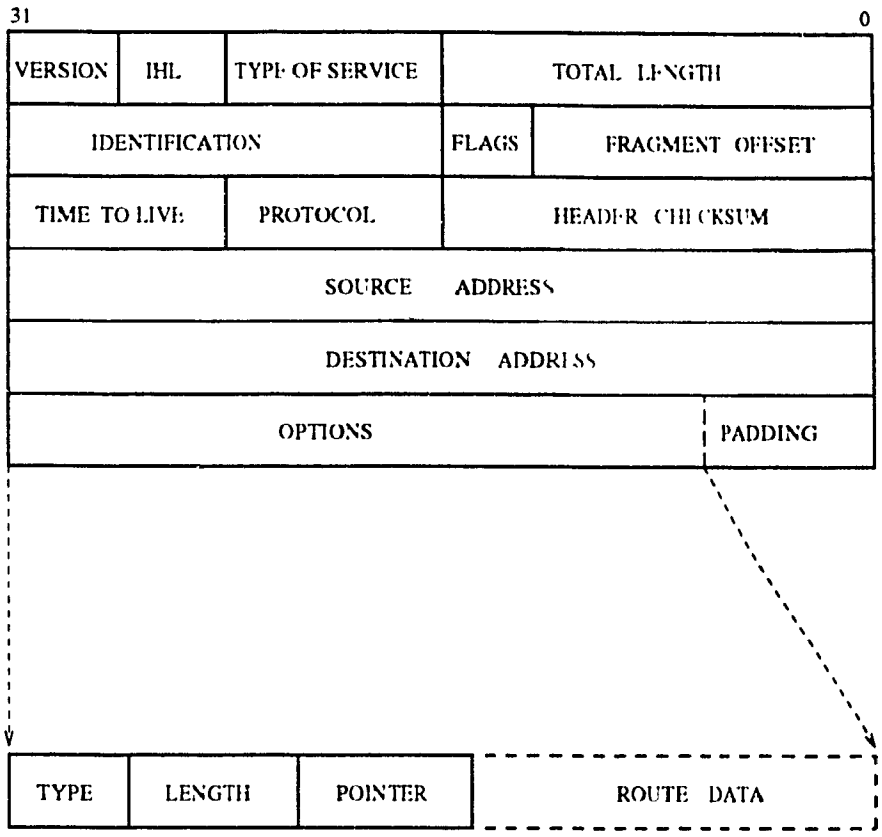


Figure 10: IP HEADER

- LOOSE SOURCE ROUTING

Loose source routing option is identified by the type field having a value of 131. The length field gives the length of the option. For the option having type number 131, the route data field consists of internet addresses that the packet has to visit on its way to the final destination specified in the IP header. The pointer field points to the next address to be processed. Since the option is defined as loose, the packet is not restricted to take the direct path between two adjacent addresses. Depending on the congestion status on the network, the packet is allowed to traverse several other intermediate routers.

XTP, like TCP/IP, is another family suite of internetworking protocol that combines

the functionalities of both the network and the transport layer. Since the network mechanisms are not so well defined as the transport mechanisms and the theme of the project is confined within the transport layer, for simplicity, the ubiquitous IP was used as the data link layer to provide point to point movement of XTP packets as shown in figure 9B. A MAC table was defined that mapped the simple integer MAC addresses being manipulated by the XTP router into the internet addresses.

6 XTP SIMULATION PHASE

6.1 CHANNEL INTERACTIONS

Three modules are defined in the test configuration. These are the user module, the XTP module and the medium module. Figure 11 shows the channels that interconnect the modules.

The channel that connect a user module to an XTP module is called the GLOBAL-USER-CHANN and the one connecting the XTP module to the medium module is called the MEDIUM-CHANNEL. The interactions that are exchanged between the user module and the XTP module are as follows:

- FROM USER-MODULE TO XTP-MODULE

    1. TCONreq
    2. TCONresp
    3. TDISreq
    4. TLSTreq
    5. TSORTreq
    6. BUSY
    7. IDLE

- FROM XTP-MODULE TO USER-MODULE

    1. TCONind
    2. TDISind
    3. TDATind
    4. TERRind

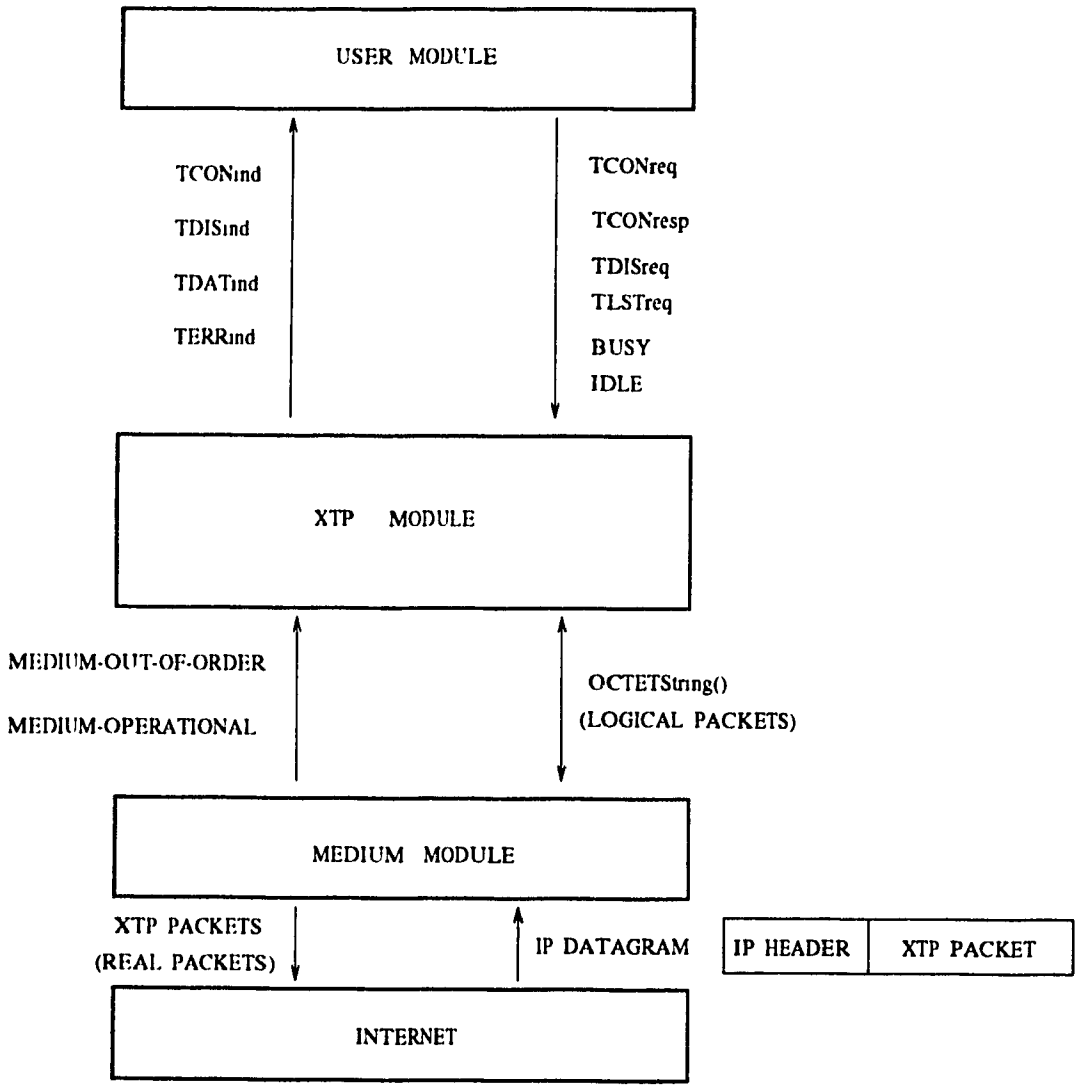


Figure 11: FLOW OF INTERACTIONS

TCONreq

TCONreq is the primitive issued by the user to request the creation of an association. The parameters specify the type of service required; the source and destination addresses and if possible some user data. The XTP module uses this primitive to create one end of a new association and generate the FIRST packet to be sent to the specified destination.

TLSTreq

TLSTreq is a primitive issued to create a listening context. It is created by the receiver module prior to the arrival of the FIRST packet. The address parameter identifies the address of the local host. If the listening flag is set, the context module will ask for connection confirmation when it receives the FIRST packet.

TDATAreq

TDATreq is used to transfer user data to the XTP module.

TDISreq

TDISreq is used to reject a new association.

BUSY

BUSY it indicates that the user cannot accept new indications from the XTP module (back pressure).

IDLE

IDLE shows that the user is ready to accept new indications from the XTP module.

TCONind

TCONind indicates to the user that a new association has been created.

TDISind

TDISind shows the rejection of a new association.

TDATind

TDATind is used to transfer data to the user.

TERRind

TERRind indicates that an error has occurred.

The interactions that are exchanged between the xtp module and the medium module are defined below

- FROM XTP-MODULE TO MEDIUM-MODULE

    1. octetstring

- FROM MEDIUM-MODULE TO XTP-MODULE

    1. octetstring
    2. medium_out_of_order
    3. medium_operational

OCTETSTRING

OCTETSTRING contains the data parameter that is the union of all packets. The other parameters represent the mac addresses of the source and destination machines.

INITIALIZATION.

The modules were initialized as follows

```
MODVAR
     user   : user_module(i);
     xtp    : xtp_module(i);
     medium : medium_module(i);

     where i = 1..2 identify the host number.
```

Once the modules are declared, the channels are created between the interaction points using the following Estelle command.

```
CONNECT  user.x    TO  xtp.x[1];
CONNECT  xtp.m[1]  TO  medium.x;
```

6.2 MODULES.

6.2.1 USER MODULE.

This module defines the XTP users at both end of an association (see page 53). It is a simplified version of an application layer that makes use of XTP to transfer data reliably from one user in one machine, to another user located in a different machine. The SENDER module is the originator of data, while the RECEIVER module is the recipient of the transferred data.

1. SENDER MODULE.

The functionalities of this module are executed by the following transitions.

START TRANSITION

This is the first transition to fire. It fills the parameters for the TCONreq primitive before dispatching it to the XYP module via the interaction point using the command OUTPUT.

The abtract representation of the transition is given below

```
TRANS
    FROM  idle_state
    TO    transfer_state
    NAME start_connection:
    BEGIN
        fetch data from buffer
        fill in parameters, ie.., source and dest addresses,
        type of service, etc.
        OUTPUT  TCONreq(...)
    END;
```

SEND-DATA TRANSITION

Once the start connection transition fires, the module moves to the next transition state, which makes this transition fireable. The transition then enters a loop until all the data have been transferred from the user buffer. The abstraction of the transition is as follows

```
TRANS
    FROM  transfer_state
    TO    SAME
    NAME SEND-DATA:
    BEGIN
        read data from the buffer;
        fill in the parameters;
        OUTPUT x.TDATreq(...);
    END;
```

At the end of the data transfer, the module will move to the final state and display the amount of time it took to transfer the data from the user module to the XTP module.

2. RECEIVER MODULE.

The first transition to fire issues a TLSTreq primitive with the flag parameter set. The module moves from idle to listen state. When the TCONind primitive is received from the XTP module, the second transition fires by issuing a TCONresp primitive to accept the connection.

```
TRANS
     FROM  idle
     TO   listen_state
     NAME start_receiver:
     BEGIN
         assign local host address;
          fill other parameters;
  OUTPUT x.TLSTreq(..);
     END;


TRANS
     FROM  listen_state
     TO    active_state
     WHEN x.TCONind(..)
     NAME accept_connection:
     BEGIN
  fill parameters;
  OUTPUT x.TCONresp(..);
     END;
```

Once the accept connection transition fires, the module moves to the next transition state and loops to receive the data until the end of message parameter is set.

```
TRANS
     FROM  active_state
     TO    same
     WHEN  x.TDATind(..)
     NAME  receive_data:
     BEGIN
   inspect the received parameters for eom;
   store data in the buffer;
     END;
```

6.2.2 MEDIUM MODULE

This module provides the interface between the XTP module and the Internet Protocol (see page 62). Although the Internet Protocol is a network layer in OSI model, in this simulation it is being used as the unreliable data link layer. The simple integer MAC addresses being manipulated by the XTP router (in Estelle) are mapped into the internet addresses using the following simple switch statement

```
switch (destination-mac-address)   {
    case 1 : dest_address = "132.205.62.2"; break;
    case 2 : dest_address = "132.205.45.24";
}
```

where mac one identifies the source host (sender) and mac two identifies the destination host (receiver). The transitions defined by the module are the sender and the receiver transitions.

SENDER TRANSITION.

The sender transition becomes fireable when the OCTETString primitive from the XTP module is present at the head of the queue of this module's interaction point. The received logical packet is transformed into the physical representation and the destination mac address into the corresponding internet address. The physical packet is then transmitted through the IP socket. The abstract representation of this transition is as follows

```
TRANS
WHEN x.octetstring(..)
NAME send_data:
BEGIN
    convert the received packet into physical representation
    map mac destination address into internet address
    send the data through the IP socket
 END;
```

RECEIVER TRANSITION.

This transition become fireable when the flag in the PROVIDE clause indicates that there is data to be read from the IP socket. This flag is a function that uses the *recvfrom* system call with the MSG_PEEK as one of its parameters. MSG_PEEK causes the system call to return the number of bytes the socket has received without doing the actual read. When the transition fires, the same system call with the MSG_PEEK flag replaced by a zero, is used to read the data from the socket. The data read is an IP datagram containing the XTP packet as its data segment.

The XTP packet is then extracted from the datagram, converted into logical form before being channelled to the XTP module.

```
TRANS
```

```
PROVIDED  (data_to_read  > 0)
NAME receive_data:
BEGIN
     read datagram from the socket,
     extract the XTP packet,
     convert the packet to logical form,
     OUTPUT x.OCTETString(...);
END;
```

6.3 SIMULATION.

In order to be able to make use of the IP socket, the user must have logged in as the super user. The C code library routines are compiled into an object file using the C compiler as show below

> cc c_code.c

an archive file is then created as follows

ar rcv lib.a c-code.o

ranlib liba.a

At the source host machine, the simulation is initialized with the following command

> edb -Lliba.a source.stl

At the destination host machine, the initialization is done as follows

> edb -Lliba.a destination.stl

where -L is the option specifying the library to be linked to the edb executable file. source.stl is the estelle specification module initialized with host identification number one. destination.stl is the specification module initialized with host identification number two.

Once the compilation and linking is successful, an edb prompt is produced.

STEP BY STEP SIMULATION.

This mode of simulation was used to monitor the flow of interactions, to display the contents of the interactions, to validate the correctness of the C code routines defined in the library and to select the next transition to fire.

CONTINUOUS SIMULATION.

To perform a continuous simulation of up to 5000 transition steps, the number is set using the following command

```
edb> $fs := 5000;
```

followed by the following command

```
edb> c
```

These procedures are carried out on both machines.

7 SIMULATION OBSERVATION

Since the Estelle specification defines the queues of the interaction points as unbounded, the XTP module piles up the user interactions containing data at its external interaction point. The actual transmission starts when all the data have been received. For a large amount of data, this process has the effect of increasing the overall transmission time. For better performance, some kind of piping mechanism needs to be incorporated to allow the XTP module to start the transmission process as soon as a certain amount of data (low water mark) has been received.

One of the EDB properties that turned out to be a drawback in the project is the one that issues a deadlock prompt when there is no fireable transition to choose from. At the destination machine, if the received data have been delivered to the user and no more packets have arrived from the source machine, there will be no fireable transition and hence the simulation will be terminated and an edb deadlock prompt issued, even though additional data are "on their way".

To circumvent the termination of the simulation due to lack of fireable transitions, the receiver transition in the medium module was made fireable regardless of whether there were data or not in the IP socket. This was done by removing the PROVIDED clause in the transition. Some test procedures were added within the transition to distinguish between a system call that returned due to some internal error or because there was no data to be read. This modification had the effect of almost doubling the overall transmission time. Having at least one fireable transition at any given time created another side effect. The DELAY clause that provides the delay mechanism for the error recovery protocol, becomes operational only when there is no fireable transition. To keep things moving, we improvised a DELAY mechanism by introducing two additional transitions with flip-flop characteristics to provide timing control to each XTP transition with a DELAY clause. This problem was actually introduced because the "simulation" time maintained by EDB had no relationship to "real" (*wall clock*) time. The two additional transitions established this relationship.

The first transition would fire when the conditions required to start the DELAY clause are met. It would then get the time from the system clock before flaging off the second transition. The second transition would fire when the current time minus the time observed by the first transition is greater than or equal to the specified delay. During the execution of this transition, a condition will be set that will make the XTP transition fireable.

The size of the data segment in the data carrying packets was limited to only 20 bytes. The data being exchanged between the XTP module and the user module was limited to a maximum of 30 bytes. Such small sizes increase the number of packets that have to be exchanged between the two XTP protocol engines. Considering the fact that we are dealing with the unreliable data link layer, more packets are likely to get lost and hence considerable delay is expected in the overall transmission time.

As the interactions move from one interaction point to the next, there is an enormous amount of copying that takes place, which by itself, is another factor that contributes to the transmission delay. To miminize the copying to at most twice, once when the data are received from the medium and once when the data is being delivered to the user space, a data descriptor should replace the data parameter in all data-carrying interactions. The descriptor should have only two parameters, the address and a counter, limiting its size to 8 bytes. The address specifies the starting address of the data in an array while the counter specifies the number of valid bytes in the array. Performing the copying operation on the descriptor rather than the data itself would have a significant impact on the performance of the XTP protocol engine. Finally, the most disturbing issue during the whole exercise was the unpredictable breakdown of simulation process due to an EDB internal error number 30018. This is currently being investigated by the staff at INT (where the EDB system is maintained).

8 CONLUSION

The operation of the error recovery mechanisms of the XTP specification are capable of delivering data reliably to the XTP user despite the unreliability of the underlying data link layer. However, the overall performace was not good enough because the transfer time for 5000 bytes was being measured in seconds.

The XTP specification that has been implemented in C programming language and run on the 486 DX processor transfers the same amount of data in milliseconds. Such a great difference thwarted our next hope of connecting these two protocol engines together using an FDDI link.

In order to improve the performance of the XTP Estelle specification to a level where it can be considered practical for implementation purposes, the suggestions outlined in the previous section need to be implemented.

XTP = Xpress Transfer Protocol

TCP = Transport Control Protocol

VLSI = Very Large Scale Integration

ATM = Asynchronous Transfer Mode

Ip = Internet Protocol

PAR = Positive Acknowledgement with Retransmission

ARQ = Automatic Repeat Request

OSI = Open Systems interconnect

MAC = Medium Access Control

ISO = International Standardization Organization

VMTP = Versatile Message Transaction Protocol

NETBLT = Network Block transfer

MSB = Most Significant Bit

UDP = User datagram Protocol

EOM = End Of Message

FDDI = Fiber Distributed Data Interface

RTT = Round Trip Time

FIFO = First In First Out

Sreq = Status request

EDB = Estelle debugger/Simulator

References

[1] J.W. Atwood, Anindya Das.M'Hamed Nour, Jean-Marc Jé zéquel: *Addressing and routing in heterogeneous data networks.* Accepted for presentation at High Performance Networks '91, Grenable, France, June, 1994, 12 pages.

[2] J.W. Atwood, G.K.C. Chung: *Error Control in the Xpress Transfer Protocol.* Proceedings of the 18th Conference on Local Computer Networks, Minneapolis, MN, September 19-22, 1993, pp.423-431.

[3] D. Sanghai and Ashok K. Agrawalla: *DTP an efficient transport protocol.*

[4] *Protocol engine design.* Greg Chesson, Silicon Graphics, Inc.

[5] RFC 791 *Internet Protocol (IP).*

[6] *XTP Protocol Definition revision 3.6.* Protocol Engines Incorporated.

[7] O.Catrina and E.Lallet: *Contributions to the specification and validation of the Xpress Transfer Protocol.* Research report 931005, Systems and Networks Department, Institut National des Télécommunications, Évry, France, October 1993.

[8] Nicky G. Ayoub:*Using the network interface tap.* Project report, Department of Computer Science, Concordia University.

[9] George C.K. Chung:*Design and validation of error control in an XTP simulator.* Major report, Department of Computer Science, Concordia University, February 1993.

[10] Richard Stephens:*Unix network programming.* Prentice Hall, Englewood Cliffs, N.J., 1990.

[11] W. Timothy Strayer, Bert J. Dempsey and Alfred C. Weaver: *XTP:The Xpress Transfer Protocol.* Addison-Wesley Publishing Company, 1992.

[12] *Estelle tutorial, Estelle to C compiler (version3.0) manual and Estelle simulator/debugger (EDB).* Institut National des Télécommunications Systéms et Réseaux Departement, France.

[13] Douglas E. Comer:*Internetworking with TCP/IP (Principles, Protocols, and Architecture).* Prentice Hall, Englewood Cliffs, N.J., 1988.

[14] D. Sanghai and Ashok K. Agrawalla:*DTP: an efficient transport protocol.* Proceedings of NETWORKS '92, Trivandrum, India, October 1992.

9 APPENDIX A. user module

```
*****************************************************************
*    NOTE:  rl_   means the specified function performs real    *
*                to logical  conversion.                        *
*           lr_   the conversion  is from logical to real       *
*                representation.                                *
*****************************************************************

{ ---------------USER  MODULE-------------------}

{ It must be noted that the user module acts as a virtual user instead of a
real user. It is because we are specifying the XTP protocol, not the behaviour
of some user. Therefore, we decided to simplify the user module and we
assigned to it a "standard" user channel}


MODULE User_module SYSTEMACTIVITY (host_id:integer);
   {host_id parameter allows host dependent  user instances to be created. }
IP      x : GlobalUserChannel(user);    {the number of interaction points x is
                                        irrelevant for the specification of
                                         xtp. This is the reason why we chose
                                         a number of one interaction point.}
END;

BODY user_body for User_module;
```

```
{ To be defined according to the tests. }

STATE idle,active,listen_state,end_receive,end_send,rejected,error;
VAR
  lnoerr,lsreq,ldreq,lfastnak,lnocheck,lnoflow,sleom,
  rleom,lbtag,lsort,lwclose,lrclose,lend : BOOLEAN;
lsortval : INTEGER;
ctl_nfp : Cntl_param_with_noflow_type;
ctl_p   : Cntl_param_type;
closep  : Close_param_type;
datap   : Data_type;
dst,src : Host_address;
 itime1,ftime1,itime2,ftime2 : REAL;

PROCEDURE initflags;
BEGIN
    lsortval := 0;
    lnoerr := FALSE;
    lsreq := FALSE;
    ldreq := FALSE;
    lfastnak := FALSE;
    lnoflow := FALSE;
    lnocheck := FALSE;
    sleom := FALSE;
    rleom := FALSE;
    lbtag := FALSE;
    lsort := FALSE;
    lwclose := FALSE;
    lrclose := FALSE;
```

```
    lend := FALSE;
END;

PROCEDURE fill_nf_cntlp(VAR ctp : Cntl_param_with_noflow_type;a,b,
c,d,e,f,g,h,i: BOOLEAN; z : INTEGER);
BEGIN
   WITH ctp DO
   BEGIN
    noerr := a;
    sreq := b;
    dreq := c;
    fastnak := d;
    noflow := e;
    nocheck := f;
    eom := g;
    btag := h;
    sort := i;
    sort_val := z;
  END;

PROCEDURE fill_cntlp(VAR ctp : Cntl_param_type;a,b,c
                  ,d,e,f,g,h : BOOLEAN; z : INTEGER);
BEGIN
    WITH ctp DO
    BEGIN
        noerr := a;
        sreq := b;
        dreq := c;
        fastnak := d;
        nocheck := e;
```

```
        eom := f;
        btag := g;
        sort := h;
        sort_val := z;
     END;
END;

PROCEDURE fill_closep(VAR ctp : Close_param_type;a,b,c : BOOLEAN);
BEGIN
     WITH ctp DO
     BEGIN
        wclose := a;
        rclose := b;
        endc := c;
     END;
END;

INITIALIZE
      TO  idle
      BEGIN
          IF (host_id = 1) THEN  {sender}
          BEGIN
             init_data;     {load data for transmission}
          END;
          initflags;
          int_time;         {initialize time}
      END;

{   ********** USER HOST No. 1 (sender) ***************  }
```

```
TRANS
     FROM idle
     TO active
     PROVIDED (host_id = 1)
     NAME start_connection:
     BEGIN
        itime1 := local_time;
        fetch_data(datap);
        IF (datap.nb_data = 0) THEN
      BEGIN
             sleom := TRUE;
             lwclose := TRUE;
      END;
       fill_nf_cntlp(ctl_nfp,lnoerr,lsreq,ldreq,lfastnak,lnoflow,
                    lnocheck,sleom,lbtag,lsort,lsortval);
       fill_closep(closep,lwclose,lrclose,lend);
         dst.host_identifier := 2;    {receiver MAC address}
         dst.selector := 1;
         src.host_identifier := 1;    {sender  address}
         src.selector := 1;
         OUTPUT x.TCONreq(ctl_nfp,datap,closep,dst,src,1);
         IF lwclose THEN
            ftime1 := local_time;
     END;

TRANS            {loop to transmit data}
    FROM active
    TO  SAME
    PROVIDED (NOT lwclose AND (host_id = 1))
    NAME give_data_for_trans:
```

```
    BEGIN
        fetch_data(datap);
       IF (datap.nb_data = 0) THEN
         BEGIN
             sleom := TRUE;
             lwclose := TRUE;
         END;
    fill_cntlp(ctl_p,lnoerr,lsreq,ldreq,lfastnak,lnocheck,
               sleom,lbtag,lsort,lsortval);
    fill_closep(closep,lwclose,lrclose,lend);
    OUTPUT x.TDATreq(ctl_p,datap,closep,1);
             IF lwclose THEN
             BEGIN
               ftime1 := local_time;
             END;
    END;


TRANS               {no more data to transmit}
    FROM active
    TO end_send
    PROVIDED (lwclose  AND (host_id = 1))
    NAME all_data_sent:
    BEGIN
         printtime(TRUNC(ftime1 - itime1));
    END;

{*********** USER HOST no. 2 (receiver) ************  }

TRANS
```

```
    FROM idle
    TO listen_state
    PROVIDED (host_id = 2)
    NAME start_listen:
    BEGIN
        src.host_identifier := 2;
        src.selector := 1;
        OUTPUT x.TLSTreq(src,TRUE,1);
    END;

TRANS
    FROM listen_state
    TO active
    PROVIDED (host_id = 2)
    WHEN x.TCONind(cntl_param,destination,source,user_number)
    NAME rec_active:
    BEGIN
        itime2 := local_time;
        OUTPUT x.TCONresp(user_number);     {confirm connection}
        rleom := cntl_param.eom;
        IF rleom THEN
           ftime2 := local_time;
    END;

TRANS
    FROM listen_state
    TO active
    PROVIDED (host_id = 2)
    WHEN x.TCONcnf(user_number)
    NAME rec_confirm:
```

```
     BEGIN
     END;

TRANS                        {loop to receive  data}
    FROM active
    TO SAME
    WHEN x.TDATind(data_param,btag,eom,user_number)
    PROVIDED (NOT rleom AND (host_id = 2))
    NAME user_get_data:
    BEGIN
          storedata(data_param);
         IF (data_param.nb_data = 0) THEN
               lrclose := TRUE;
         rleom := eom;
         IF lrclose THEN
           ftime2 := local_time;
    END;

TRANS                              {end of received data}
    FROM active
    TO end_receive
    PROVIDED ((rleom OR lrclose) AND (host_id = 2))
    NAME time_taken:
    BEGIN
       printoutput;
       printtime(TRUNC(ftime2 - itime2));
    END;
```

```
TRANS
     FROM idle,listen_state,active
     TO rejected
     WHEN x.TDISind(user_number,code)
     NAME rec_disconnect:
     BEGIN
     END;


TRANS
    FROM idle,listen_state,active
    TO error
    WHEN x.TERRind(error,user_number)
    NAME rec_abort:
    BEGIN
    END;

END; { of User body }
```

10 APPENDIX B. medium module

```
{_____________ MEDIUM _____________ }



MODULE Medium_module SYSTEMACTIVITY(host_id : INTEGER);
IP      x: ARRAY[1..2] OF  MediumChannel(medium);
END;

BODY medium_body for Medium_module;
{ To be defined according to the tests. }
INITIALIZE
       BEGIN
       open_socket;
       END;

TRANS
   PROVIDED  (checkdata > 0)   {packet has arrived}
    VAR
    data : bitstring;
    flag,src,dst,addr : INTEGER;
    NAME from_network:
    BEGIN
       get_data(data,flag);
       IF (flag = 1) THEN     { valid packet}
       BEGIN
   src := 1;
```

```
    dst := 2;
    IF (host_id = 1 ) THEN
    BEGIN
       src := 2;
       dst := 1;
    END;
  OUTPUT x[1].octetstring(data,src,dst);
        END;
     END;


TRANS
      WHEN x[1].octetstring(data,src_mac,dst_mac)
      NAME to_network:
      BEGIN
      send_data(data,src_mac,dst_mac);
      END;

END; { end of medium body }
```

11 APPENDIX C. C-code primitives

```
{     C - CODE LIBRARY ROUTINES.    }


#include <stdio.h>
#include <sys/stat.h>
#include <signal.h>
#include <fcntl.h>
#include <sys/types.h>
#include <errno.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netinet/ip_var.h>
#include <netdb.h>

#define FORE_ADDR "132.205.45.24"  /* forest  destination address */
#define PINE_ADDR "132.205.62.2"   /* pine    source      address */

/* service types */

#define   connection    0x00
#define   transaction   0x01
#define   unack_dgram   0x02
#define   ack_dgram     0x03
#define   iso_stream    0x04
#define   bulk_data     0x05
```

```
/* format types */

#define   no_addr             0x00
#define   ip_addr             0x01
#define   iso_addr            0x02
#define   xns_addr            0x03
#define   ibm_src_route_addr  0x04
#define   modsim_addr         0x05
#define   netbios_addr        0x06
#define   ip_loose_src_addr   0x07
#define   ip_strict_src_addr  0x08
#define   xtp_direct_addr     0x09
#define   xtp_x_addr          0x0a
#define   usaf_addr           0x0b


#define   returnmask          0x80000000
#define   instancemask        0x7FE00000
#define   indexmask           0x001FFFFF

/* packet types (real) */

#define   p_data          0x00
#define   p_cntl          0x01
#define   p_first         0x02
#define   p_path          0x06
#define   p_diag          0x08
#define   p_maint         0x0a
#define   p_mgmt          0x0e
```

```
#define   p_route          0x12
#define   p_rcntl          0x13


/* packet types (logical)  */

enum lptype {
      First_pak,
      Data_pak,
      Cntl_pak,
      Path_pak,
      Diag_pak,
      Route_pak,
      Rcntl_pak
      };
typedef enum lptype lptype;

#define   Max_nb_of_span          10
#define   INSTANCE_SIZE           10      /* 1023 */
#define   INDEX_SIZE              21      /* 2097151 */

/* diagnostic messages (real) */
#define  INVAL_CONTEXT  1
#define  REF_CONTEXT    2
#define  UN_DEST        3
#define  DEAD_HOST      4
#define  INVAL_ROUTE    5
#define  REDIRECT       6
#define  NO_ROUTE       7
#define  NO_RESOURCE    8
```

```
#define  ERR_PROTOCOL   9
#define  ERR_MAXDATA    10
#define  REL_ROUTE      11
#define  REL_ACK        12
#define  GOOD           13

#define First_data_seg_max_length 20
#define Data_seg_max_length 20
#define Max_nb_of_spans 10
#define addr_max_len  72
#define Diag_message_max_length 20


typedef unsigned char  bool;
typedef unsigned int word_32;

/* ******************************
        LOGICAL  STRUCTURES
   ******************************  */


struct cmd_type {
      bool btag;
      bool endc;
      bool eom;
      bool wclose;
      bool rclose;
      bool dreq;
      bool sreq;
      bool fastnak;
```

```
        bool noflow;
        bool sort;
        bool res;
        bool multi;
        bool noerr;
        bool nocheck;
        word_32 offset;
        lptype ptype;
        };
typedef struct cmd_type lcmd_type;


enum service_type {
        connection_service,
        transaction_service,
        unack_datagram_service,
        ack_datagram_service,
        isochronous_stream_service,
        bulk_data_service
        };
typedef enum service_type service_type;

enum format_type {
         no_address_format,
         IP_format,
         ISO_format,
         XNS_format,
         IEEE_802_style_format,
         MODSIM_format,
         NetBios_format,
```

```
        IP_style_Loose_format,
        IP_style_Strict_format,
        XTP_Direct_format,
        XTP_Experimental_format,
        USAF_format
        };
typedef enum format_type aformat_type;

/* diagnostic messages (logical) */
enum diagnostic_type {
        invalid_context,
        context_refused,
        unknown_destination,
        dead_host,
        invalid_route,
        redirect,
        cannot_route,
        no_resource,
        protocol_error,
        maxdata_error,
        release_route,
        release_acknowledge,
        good
        };
typedef enum diagnostic_type diagnostic_type;

struct route_value {
      word_32 instance;
      word_32 route_index;
};
```

```
typedef struct route_value route_value;
struct lroute_type {
      bool return_value;
      route_value  value;
      };
typedef struct lroute_type lroute_type;

struct key_value {
      word_32 instance;
      word_32 index;
};
typedef struct key_value  key_value;

struct lkey_type {
      bool return_key;
      key_value value;
      };
typedef struct lkey_type lkey_type;

struct lheader_type {
      lroute_type route;
      word_32 ttl;
      lcmd_type cmd;
      lkey_type key;
      word_32 sync;
      word_32 seq;
      word_32 dseq;
      word_32 sort;
      word_32 dlen;
      word_32 hcheck;
```

```
      };
typedef struct lheader_type lheader_type;

struct trailer_type {
      word_32 dcheck;
     };
typedef struct trailer_type trailer_type;

struct  host_addr_type {
      word_32  host_identifier;
      word_32  selector;
     };
typedef struct host_addr_type host_addr_type;

struct  descr_control_type {
     word_32 alen;
     service_type service;
     aformat_type aformat;
     };
typedef struct descr_control_type descr_control_type;

struct descriptor_type {
     descr_control_type  control_field;
     word_32 rate_request;
     word_32 burst_request;
     word_32 maxdata;
     word_32 id;
     };

typedef struct descriptor_type descriptor_type;
```

```
struct  laddress_segment_type {
     descriptor_type descriptor;
     host_addr_type  destination;
     host_addr_type  source;
     };
typedef  struct  laddress_segment_type laddress_segment_type;

struct data_segment_type {
     unsigned char data_seg[Data_seg_max_length];
     };

typedef struct data_segment_type  data_segment_type;

struct spans_type {
     word_32  lowest_seq;
     word_32   highest_seq;
     };
typedef struct spans_type spans_type;

struct  lcontrol_segment_type {
     word_32 rate;
     word_32 burst;
     word_32 rsvd1;
     word_32 echo;
     word_32 time;
     word_32 techo;
     lkey_type xkey;
     lroute_type xroute;
     word_32 rsvd2;
```

```
        word_32 alloc;
        word_32 rseq;
        word_32 nspan;
        spans_type spans[Max_nb_of_spans];
        };

typedef struct lcontrol_segment_type  lcontrol_segment_type;

struct diag_info_type {
        diagnostic_type  code;
        word_32 value;
        unsigned char message[Diag_message_max_length];
        };
typedef  struct diag_info_type  diag_info_type;

struct  rcntl_segment_type {
        word_32 rate;
        word_32 burst;
        lroute_type xroute;
        };
typedef  struct  rcntl_segment_type    rcntl_segment_type;

 /* logical first packet */
struct lxtp_first_packet {
        lheader_type  header;
        laddress_segment_type  address_segment;
        unsigned char data_segment[20];
        trailer_type trailer;
       };
 typedef struct lxtp_first_packet  lfirst_packet;
```

```
 /* logical data packet  */
struct lxtp_data_packet {
     lheader_type  header;
     data_segment_type  data_segment;
     trailer_type trailer;
     };
 typedef  struct lxtp_data_packet ldata_packet;

struct  lxtp_control_packet {
     lheader_type header;
     lcontrol_segment_type  control_segment;
     trailer_type trailer;
     };

/* logical control packet */
typedef struct lxtp_control_packet  lcontrol_packet;

struct  lxtp_path_packet {
      lheader_type  header;
      laddress_segment_type  address;
      trailer_type  trailer;
     };

/* logical path packet */
typedef struct lxtp_path_packet  lpath_packet;

struct lxtp_diag_packet {
     lheader_type  header;
     diag_info_type  info;
```

```
     trailer_type  trailer;
     };

 /* logical diag packet */
typedef struct lxtp_diag_packet  ldiag_packet;

/* logical rcontrol packet */
struct lxtp_rcntl_packet  {
      lheader_type header;
      rcntl_segment_type  rcntl_segment;
      trailer_type trailer;
      };
typedef struct lxtp_rcntl_packet lrcntl_packet;


 /* logical route packet */
struct  lxtp_route_packet  {
     lheader_type header;
     diag_info_type info;
     trailer_type  trailer;
     };
typedef struct lxtp_route_packet  lroute_packet;

/*  ****************************
      PHYSICAL  STRUCTURE
    ****************************  */


struct rxtp_cmd_bit {
      unsigned int rotused1 : 1;
```

```
        unsigned int nocheck  : 1;
        unsigned int notused2 : 1;
        unsigned int noerr    : 1;
        unsigned int multi    : 1;
        unsigned int res      : 1;
        unsigned int sort     : 1;
        unsigned int noflow   : 1;
        unsigned int fastnak  : 1;
        unsigned int sreq     : 1;
        unsigned int dreq     : 1;
        unsigned int rclose   : 1;
        unsigned int wclose   : 1;
        unsigned int eom      : 1;
        unsigned int end      : 1;
        unsigned int btag     : 1;
        unsigned int offset   : 8;
        unsigned int notused3 : 1;
        unsigned int version  : 2;
        unsigned int pformat  : 5;
        };

union xtp_cmd {
        word_32  word;
        struct rxtp_cmd_bit bits;
        };
typedef union xtp_cmd xtp_cmd;

struct xtp_route_bits {
        unsigned int return_route : 1;
        unsigned int route_value  : 31;
```

```
        };

union xtp_route {
        word_32 word;
        struct xtp_route_bits  bits;
        };
typedef union xtp_route xtp_route;

struct xtp_key_bit {
        unsigned int return_value : 1;
        unsigned int key_value    : 31;
        };

union xtp_key {
        word_32 word;
        struct xtp_key_bit bits;
        };
typedef union xtp_key  xtp_key;

struct xtp_header {
        xtp_route  route;
        word_32     ttl;
        xtp_cmd    cmd;
        xtp_key    key;
        word_32     sync;
        word_32     seq;
        word_32     dseq;
        word_32     sort;
        word_32     dlen;
        word_32     hcheck;
```

```
      };
typedef struct xtp_header xtp_header;

struct rcontrol_segment {
       word_32 rate;
       word_32 burst;
       word_32 rsvd1;
       word_32 echo;
       word_32 time;
       word_32 techo;
       xtp_key xkey;
       xtp_route xroute;
       word_32 rsvd2;
       word_32 alloc;
       word_32 rseq;
       word_32 nspan;
       spans_type spans[Max_nb_of_spans];
       };
typedef struct rcontrol_segment rcontrol_segment_type;

struct rdiag_info_type {
       word_32 code;
       word_32 val;
       unsigned char message[Diag_message_max_length]
       };
typedef struct rdiag_info_type  rdiag_info_type;

struct descriptor_control {
      unsigned int alen    : 16;
      unsigned int service : 8;
```

```
        unsigned int aformat : 8;
        };

union rdescr_control {
        word_32  word;
        struct descriptor_control descr_cntl;
         };
typedef union rdescr_control   rdescriptor_control;

struct rdescriptor_type {
        rdescriptor_control control;
        word_32  rate_req;
        word_32  burst_req;
        word_32  maxdata;
        word_32  id[2];
       };
typedef  struct rdescriptor_type  rdescriptor_type;

struct raddress_segment {
        rdescriptor_type descriptor;
        word_32  address[12];
        };
typedef struct raddress_segment raddress_segment_type;

union all_packets {
      lfirst_packet first;
      ldata_packet data;
      lroute_packet route;
      lrcntl_packet rcntl;
      ldiag_packet diag;
```

```
     lpath_packet path;
     lcontrol_packet cntl;
    };
;
typedef union all_packets all_packets;
struct bit_string {
    lptype   ptype;
    all_packets pack;
   };
typedef struct  bit_string bit_string;

struct rfirst_packet_type {
      xtp_header  header;
      raddress_segment_type address_seg;
      unsigned char data_segment[20];
      trailer_type  trailer;
      };
typedef struct rfirst_packet_type FIRST;

struct rdata_packet_type {
       xtp_header  header;
       unsigned char data_segment[20];
       trailer_type  trailer;
       };
typedef struct rdata_packet_type DATA;

struct rcontrol_packet_type {
       xtp_header header;
       rcontrol_segment_type  control;
       trailer_type trailer;
```

```
        };
typedef struct rcontrol_packet_type  CNTL;

struct rpath_packet_type {
        xtp_header header;
        raddress_segment_type address;
        trailer_type trailer;
        };
typedef struct rpath_packet_type PATH;

struct rdiag_packet_type {
      xtp_header header;
      rdiag_info_type info;
      trailer_type trailer;
      };
typedef struct rdiag_packet_type DIAG;

struct  rrcntl_packet_type  {
      xtp_header header;
      rcontrol_segment_type  control;
      trailer_type trailer;
      };
typedef struct rrcntl_packet_type  RCNTL;

struct rroute_packet_type {
      xtp_header header;
      rdiag_info_type info;
      trailer_type trailer;
      };
typedef struct rroute_packet_type  ROUTE;
```

```
struct ipheader {
   unsigned int version : 4;
   unsigned int hln     : 4;
   unsigned int dtype   : 8;
   unsigned int tlen    : 16;
   unsigned int id      : 16;
   unsigned int flag    : 4;
   unsigned int offset  : 12;
   unsigned int ttl     : 8;
   unsigned int proto   : 8;
   unsigned int hsum    : 16;
   unsigned int src     : 32;
   unsigned int dst     : 32;
};
typedef struct ipheader ipheader;


#define datapt 30
#define mdata 5000          /* amount of data to be transferred   */
struct data_type {
    unsigned int nb_data;
    unsigned char  data[datapt];
};
typedef struct data_type data_type;
static unsigned char arraydata[mdata];  /* store data to be send by the source  */

typedef struct storedatatype { /* store data received by destination host  */
  int count;
```

```
  unsigned char data[mdata];
 };
char *address_type;
typedef struct storedatatype storedatatype;
static storedatatype outdata;

#define IP_HEADERLEN(x)    (((*x).hln) << 2)     /* get ip header length */
#define BUFSIZE 300

 extern int errno;
static int  first_len = sizeof(FIRST);
static int  data_len  = sizeof(DATA);
static int  diag_len  = sizeof(DIAG);
static int path_len = sizeof(PATH);
static int  route_len  = sizeof(ROUTE);
static int  cntl_len  = sizeof(CNTL);
static int  rcntl_len   = sizeof(RCNTL);
static nextwrite = 0;
static int index = 0;
static int source = 0;
static unsigned int probno;
static int nextread = 0;
static  int sockfd,fd,clilen,sockfd1,slen;
static  struct sockaddr_in serv_addr,cli_addr,saddr;
static unsigned char sbuff[BUFSIZE],rbuff[BUFSIZE];
static struct timeval tv0,tv1;
static int count_no = 0;
static unsigned char option[32];   /* IP OPTION  STRUCTURE   */
```

```
/*    ******************************************
        LOGICAL TO REAL  CONVERSION  PRIMITIVES
      ******************************************
*/
void  lr_keytype(lpt, rpt)
register lkey_type  *lpt;
register xtp_key    *rpt;
{
{
     rpt->word = 0;
     rpt->word = lpt->return_key;
     rpt->word = rpt->word << INSTANCE_SIZE;
     rpt->word += lpt->value.instance;
     rpt->word = rpt->word << INDEX_SIZE;
     rpt->word += lpt->value.index;
}

void lr_routetype(lpt, rpt)
register lroute_type  *lpt;
register xtp_route    *rpt;
{
   rpt->word = 0;
   rpt->word = lpt->return_value;
   rpt->word = rpt->word << INSTANCE_SIZE;
   rpt->word += lpt->value.instance;
   rpt->word = rpt->word << INDEX_SIZE;
   rpt->word += lpt->value.route_index;
}

void   lr_ptype(Ptype, rpt)
```

```
lptype  Ptype;
register xtp_header  *rpt;
{
   switch (Ptype)
     {
        case First_pak : rpt->cmd.bits.pformat = p_first; break;
        case Data_pak  : rpt->cmd.bits.pformat = p_data; break;
        case Cntl_pak  : rpt->cmd.bits.pformat = p_cntl; break;
        case Path_pak  : rpt->cmd.bits.pformat = p_path; break;
        case Diag_pak  : rpt->cmd.bits.pformat = p_diag; break;
        case Route_pak : rpt->cmd.bits.pformat = p_route; break;
        case Rcntl_pak : rpt->cmd.bits.pformat = p_rcntl; break;
        default : { printf("error packet type\n");
   exit(1);
 }
}
}

void lr_header(lpt,  rpt)
register lheader_type  *lpt;
register  xtp_header   *rpt;
{
   lr_routetype(&lpt->route, &rpt->route);
   rpt->cmd.bits.btag = lpt->cmd.btag;
   rpt->cmd.bits.end = lpt->cmd.endc;
   rpt->cmd.bits.eom = lpt->cmd.eom;
   rpt->cmd.bits.wclose = lpt->cmd.wclose;
   rpt->cmd.bits.rclose = lpt->cmd.rclose;
   rpt->cmd.bits.dreq = lpt->cmd.dreq;
   rpt->cmd.bits.sreq = lpt->cmd.sreq;
```

```
    rpt->cmd.bits.fastnak = lpt->cmd.fastnak;
    rpt->cmd.bits.noflow = lpt->cmd.noflow;
    rpt->cmd.bits.sort = lpt->cmd.sort;
    rpt->cmd.bits.res = lpt->cmd.res;
    rpt->cmd.bits.multi = lpt->cmd.multi;
    rpt->cmd.bits.noerr = lpt->cmd.noerr;
    rpt->cmd.bits.nocheck = lpt->cmd.nocheck;
    rpt->cmd.bits.offset = lpt->cmd.offset;
    rpt->cmd.bits.notused1 = 0;
    rpt->cmd.bits.notused2 = 0;
    rpt->cmd.bits.notused3 = 0;
    lr_ptype(lpt->cmd.ptype, rpt);
    lr_keytype(&lpt->key, &rpt->key);
    rpt->ttl = lpt->ttl;
    rpt->sync = lpt->sync;
    rpt->seq  = lpt->seq;
    rpt->dseq = lpt->dseq;
    rpt->sort = lpt->sort;
    rpt->dlen = lpt->dlen;
}


void lr_service(serv,rpt)
service_type serv;
register raddress_segment_type *rpt;
{  switch(serv)
     {
      case connection_service :
        rpt->descriptor.control.descr_cntl.service = connection; break;
      case transaction_service :
```

```
        rpt->descriptor.control.descr_cntl.service = transaction;break;
      case unack_datagram_service :
        rpt->descriptor.control.descr_cntl.service = unack_dgram;break;
      case isochronous_stream_service :
        rpt->descriptor.control.descr_cntl.service = iso_stream;break;
      case bulk_data_service :
        rpt->descriptor.control.descr_cntl.service = bulk_data;break;
      case ack_datagram_service :
        rpt->descriptor.control.descr_cntl.service = ack_dgram;break;
      default : { printf(" invalid service information \n");
 exit(1)
      }
      }
}

void lr_aformat(aform, rpt)
aformat_type  aform;
register raddress_segment_type *rpt;
{
{
  switch (aform)
    {
      case no_address_format :
 (*rpt).descriptor.control.descr_cntl.aformat = no_addr; break;
      case IP_format :
 (*rpt).descriptor.control.descr_cntl.aformat = ip_addr; break;
      case ISO_format :
 (*rpt).descriptor.control.descr_cntl.aformat = iso_addr; break;
      case XNS_format :
 (*rpt).descriptor.control.descr_cntl.aformat = xns_addr;break;
```

```
      case IEEE_802_style_format :
 (*rpt).descriptor.control.descr_cntl.aformat = ibm_src_route_addr;break;
      case MODSIM_format :
 (*rpt).descriptor.control.descr_cntl.aformat = modsim_addr; break;
      case NetBios_format :
 (*rpt).descriptor.control.descr_cntl.aformat = netbios_addr; break;
      case IP_style_Loose_format :
 (*rpt).descriptor.control.descr_cntl.aformat = ip_loose_src_addr;break;
      case IP_style_Strict_format :
 (*rpt).descriptor.control.descr_cntl.aformat = ip_strict_src_addr;break;
      case XTP_Direct_format :
 (*rpt).descriptor.control.descr_cntl.aformat = xtp_direct_addr;break;
      case XTP_Experimental_format :
 (*rpt).descriptor.control.descr_cntl.aformat = xtp_x_addr;break;
      case USAF_format  :
 (*rpt).descriptor.control.descr_cntl.aformat = usaf_addr;break;
      default : {  printf(" invalid format value\n");
   exit(1);
       }
      }
}

void rl_aformat(lpt, aform)
register laddress_segment_type  *lpt;
unsigned int  aform;
{
  switch (aform)
   {
     case no_addr :
 lpt->descriptor.control_field.aformat = no_address_format; break;
```

```
        case ip_addr :
  lpt->descriptor.control_field.aformat = IP_format; break;
        case iso_addr :
  lpt->descriptor.control_field.aformat = ISO_format; break;
        case xns_addr : lpt->descriptor.control_field.aformat = XNS_format; break;
        case ibm_src_route_addr :
  lpt->descriptor.control_field.aformat = IEEE_802_style_format;break;
        case modsim_addr :
  lpt->descriptor.control_field.aformat = MODSIM_format; break;
        case netbios_addr :
  lpt->descriptor.control_field.aformat = NetBios_format; break;
        case ip_loose_src_addr :
  lpt->descriptor.control_field.aformat = IP_style_Loose_format;break;
        case ip_strict_src_addr :
  lpt->descriptor.control_field.aformat =  IP_style_Strict_format;break;
        case xtp_direct_addr :
  lpt->descriptor.control_field.aformat = XTP_Direct_format;break;
        case xtp_x_addr  :
  lpt->descriptor.control_field.aformat = XTP_Experimental_format;break;
        case usaf_addr   :
  lpt->descriptor.control_field.aformat = USAF_format;break;
        default :  {  printf(" invalid format type \n");
   exit(1);
}
        }
}


void rl_service(lpt, serv)
register laddress_segment_type  *lpt;
```

```
unsigned int  serv;
{
 switch( serv )
   {
     case connection :
lpt->descriptor.control_field.service = connection_service;break;
     case transaction :
lpt->descriptor.control_field.service = transaction_service;break;
     case unack_dgram :
lpt->descriptor.control_field.service = unack_datagram_service;break;
     case ack_dgram  :
lpt->descriptor.control_field.service = ack_datagram_service;break;
     case iso_stream  :
lpt->descriptor.control_field.service =
    isochronous_stream_service;break;
     case bulk_data  :
lpt->descriptor.control_field.service = bulk_data_service;break;
     default :  {  printf(" invalid service number\n");
   exit(1);
}
      }
}


void  lr_address_seg(lpt,  rpt)
register laddress_segment_type *lpt;
register raddress_segment_type *rpt;
{
  rpt->descriptor.control.descr_cntl.alen = lpt->descriptor.control_field.alen;
  lr_service(lpt->descriptor.control_field.service,  rpt);
```

```
  lr_aformat(lpt->descriptor.control_field.aformat, rpt);
  rpt->descriptor.rate_req = lpt->descriptor.rate_request;
  rpt->descriptor.burst_req = lpt->descriptor.burst_request;
  rpt->descriptor.maxdata = lpt->descriptor.maxdata;
  rpt->descriptor.id[0] = lpt->descriptor.id;
  rpt->address[0] = lpt->destination.host_identifier;
  rpt->address[1] = lpt->destination.selector;
  rpt->address[2] = lpt->source.host_identifier;
  rpt->address[3] = lpt->source.selector;
}


void copydata(lp, rp, n)
register unsigned char *lp;
register unsigned char *rp;
int  n;
{
  int i;
     for (i = 0; i < n; i++)
*rp++ = *lp++;
}


void init_proba(n)
unsigned int n;
{
 probno = n;
}

void storedata(p)
```

```
 data_type p;
 {
 register  int i,s,d;
   d = outdata.count;
   s = p.nb_data;
   for (i = 0; i < s;i++)
       outdata.data[i + d] = p.data[i];
       outdata.count += s;
}


void printoutput()
{
 int i,k,d;
    d = outdata.count;
    printf("rec data   %d\n",d);
    for (i = k = 0; i < d; i++) {
 printf("%c",outdata.data[i]);
 k++;
 if (k >= 100) {
    k = 0;
    printf("\n");
 }
    }
}

void lr_control_segment(lpt, rpt)
register lcontrol_segment_type *lpt;
register rcontrol_segment_type *rpt;
{
```

```
    int i;
    rpt->rate = lpt->rate;
    rpt->burst  = lpt->burst;
    rpt->rsvd1 = lpt->rsvd1;
    rpt->echo = lpt->echo;
    rpt->time = lpt->time;
    rpt->techo = lpt->techo;
    lr_keytype(&(lpt->xkey), &(rpt->xkey));
    lr_routetype(&(lpt->xroute), &(rpt->xroute));
    rpt->rsvd2 = lpt->rsvd2;
    rpt->alloc = lpt->alloc;
    rpt->rseq = lpt->rseq;
    rpt->nspan = lpt->nspan;
    for (i = 0; i < lpt->nspan && i <= Max_nb_of_span; i++)
       {
rpt->spans[i].lowest_seq = lpt->spans[i].lowest_seq;
rpt->spans[i].highest_seq  = lpt->spans[i].highest_seq;
       }
}



void lr_diag_segment(lpt,  rpt)
register diag_info_type *lpt;
register rdiag_info_type *rpt;
{
  int i;
  switch (lpt->code)
     {
       case invalid_context : rpt->code = INVAL_CONTEXT;break;
```

```
        case context_refused : rpt->code = REF_CONTEXT;break;
        case unknown_destination : rpt->code = UN_DEST;break;
        case dead_host : rpt->code = DEAD_HOST;break;
        case invalid_route : rpt->code = INVAL_ROUTE;break;
        case redirect : rpt->code = REDIRECT;break;
        case cannot_route : rpt->code = NO_ROUTE;break;
        case no_resource : rpt->code = NO_RESOURCE;break;
        case protocol_error : rpt->code = ERR_PROTOCOL;break;
        case maxdata_error  : rpt->code = ERR_MAXDATA; break;
        case release_route : rpt->code = REL_ROUTE;break;
        case release_acknowledge : rpt->code = REL_ACK;break;
        case good : rpt->code  = GOOD;break;
        default :  {  printf(" invalid diagnostic type\n");
      exit(1);
  }
      }
       rpt->val = lpt->value;
       copydata(rpt->message, lpt->message, Diag_message_max_length);
}

void lr_rrcntl_segment(lpt,  rpt)
register  rcntl_segment_type *lpt;
register rcontrol_segment_type *rpt;
{
    rpt->rate = lpt->rate;
    rpt->burst = lpt->burst;
    lr_routetype(&(lpt->xroute), &(rpt->xroute));
}

void lr_first_packet(lf,  rf)
```

```
register lfirst_packet *lf;
register FIRST *rf;
{
    lr_header(&(lf->header), &(rf->header));
    lr_address_seg(&lf->address_segment, &rf->address_seg);
    copydata(lf->data_segment, rf->data_segment,20);
}


void lr_data_packet(lpt, rpt)
register ldata_packet *lpt;
register DATA *rpt;
{
 lr_header(&(lpt->header), &(rpt->header));
 copydata((lpt->data_segment.data_seg), (rpt->data_segment),rpt->header.dlen);
}

void lr_control_packet(lpt ,rpt)
register lcontrol_packet *lpt;
register  CNTL *rpt;
{
   lr_header(&(lpt->header), &(rpt->header));
   lr_control_segment(&(lpt->control_segment),&(rpt->control));
}

void lr_rcontrol_packet(lpt, rpt)
register lrcntl_packet *lpt;
register  RCNTL *rpt;
{
  lr_header(&(lpt->header), &(rpt->header));
```

```
  lr_rrcntl_segment(&(lpt->rcntl_segment), &(rpt->control));
}


void lr_diag_packet(lpt, rpt)
register ldiag_packet *lpt;
register  DIAG *rpt;
{
  lr_header(&(lpt->header), &(rpt->header));
  lr_diag_segment(&(lpt->info), &(rpt->info));
}

void lr_path_packet(lpt,  rpt)
register lpath_packet *lpt;
register PATH *rpt;
{
  lr_header(&(lpt->header), &(rpt->header));
  lr_address_seg(&(lpt->address), &(rpt->address));
}

void lr_route_packet(lpt,  rpt)
register lroute_packet *lpt;
register ROUTE * rpt;
{
  lr_header(&(lpt->header), &(rpt->header));
  lr_diag_segment(&(lpt->info), &(rpt->info));
}


/* ********************************************
```

```
     REAL TO LOGICAL  CONVERSION  PRIMITIVES
   ********************************************
*/

void rl_keytype(lpt, rpt)
register lkey_type *lpt;
register xtp_key *rpt;
{
   lpt->value.index  = indexmask & rpt->word;
   lpt->value.instance = (instancemask & rpt->word) >> INDEX_SIZE;
   lpt->return_key  =  (returnmask & rpt->word) >> 31;
}


void rl_routetype(lpt, rpt)
register lroute_type *lpt;
register xtp_route *rpt;
{
   lpt->value.route_index  =  indexmask & rpt->word;
   lpt->value.instance = (instancemask & rpt->word) >> INDEX_SIZE;
   lpt->return_value =  (returnmask & rpt->word) >> 31;
}
void rl_ptype(lpt, Ptype)
register lheader_type  *lpt;
unsigned int Ptype;
{
  switch(Ptype)
   {
     case p_data : lpt->cmd.ptype =  Data_pak;break;
     case p_cntl : lpt->cmd.ptype = Cntl_pak;break;
```

```
        case p_first : lpt->cmd.ptype = First_pak;break;
        case p_path  : lpt->cmd.ptype = Path_pak;break;
        case p_diag  : lpt->cmd.ptype = Diag_pak;break;
        case p_route : lpt->cmd.ptype = Route_pak;break;
        case p_rcntl : lpt->cmd.ptype = Rcntl_pak;break;
        default :  { printf(" unknown packet type\n");
             exit(1);
}
   }
}


void rl_header(lpt,  rpt)
register lheader_type *lpt;
register xtp_header   *rpt;
{
  rl_routetype(&(lpt->route), &(rpt->route));
  rl_keytype(&(lpt->key), &(rpt->key));
  lpt->ttl = rpt->ttl;
  lpt->cmd.btag =   rpt->cmd.bits.btag;
  lpt->cmd.endc =   rpt->cmd.bits.end;
  lpt->cmd.eom  =   rpt->cmd.bits.eom;
  lpt->cmd.wclose = rpt->cmd.bits.wclose;
  lpt->cmd.rclose = rpt->cmd.bits.rclose;
  lpt->cmd.dreq   = rpt->cmd.bits.dreq;
  lpt->cmd.sreq   = rpt->cmd.bits.sreq;
  lpt->cmd.fastnak = rpt->cmd.bits.fastnak;
  lpt->cmd.noflow  = rpt->cmd.bits.noflow;
  lpt->cmd.sort    = rpt->cmd.bits.sort;
  lpt->cmd.res     = rpt->cmd.bits.res;
```

```
    lpt->cmd.multi   = rpt->cmd.bits.multi;
    lpt->cmd.noerr   = rpt->cmd.bits.noerr;
    lpt->cmd.nocheck = rpt->cmd.bits.nocheck;
    lpt->cmd.offset = rpt->cmd.bits.offset;
    rl_ptype(lpt, rpt->cmd.bits.pformat);
    lpt->sync = rpt->sync;
    lpt->seq  = rpt->seq;
    lpt->dseq = rpt->dseq;
    lpt->sort = rpt->sort;
    lpt->dlen = rpt->dlen;
}

void rl_address_seg(lpt, rpt)
register laddress_segment_type *lpt;
register raddress_segment_type *rpt;
{
    lpt->descriptor.control_field.alen =
                    rpt->descriptor.control.descr_cntl.alen;
    rl_service(lpt, rpt->descriptor.control.descr_cntl.service);
    rl_aformat(lpt,  rpt->descriptor.control.descr_cntl.aformat);
    lpt->descriptor.rate_request = rpt->descriptor.rate_req;
    lpt->descriptor.burst_request = rpt->descriptor.burst_req;
    lpt->descriptor.maxdata  = rpt->descriptor.maxdata;
    lpt->descriptor.id  = rpt->descriptor.id[0];
    lpt->destination.host_identifier = rpt->address[0];
    lpt->destination.selector   = rpt->address[1];
    lpt->source.host_identifier = rpt->address[2];
    lpt->source.selector = rpt->address[3];
}
```

```
void rl_control_segment(lpt, rpt)
register lcontrol_segment_type *lpt;
register rcontrol_segment_type *rpt;
{
   int i;
   lpt->rate = rpt->rate;
   lpt->burst = rpt->burst;
   lpt->rsvd1 = rpt->rsvd1;
   lpt->echo = rpt->echo;
   lpt->time = rpt->time;
   lpt->techo = rpt->techo;
   rl_keytype(&(lpt->xkey), &(rpt->xkey));
   rl_routetype(&(lpt->xroute), &(rpt->xroute));
   lpt->rsvd2 = rpt->rsvd2;
   lpt->alloc = rpt->alloc;
   lpt->rseq = rpt->rseq;
   lpt->nspan = rpt->nspan;
  for (i =0; i < rpt->nspan && i <= Max_nb_of_span; i++)
      {
lpt->spans[i].lowest_seq = rpt->spans[i].lowest_seq;
lpt->spans[i].highest_seq = rpt->spans[i].highest_seq;
      }
}

unsigned int checksum(buf,n)     /* simplified version */
register unsigned char *buf;
int n;
{
 unsigned short *pt;
 unsigned short oddbyte,answer;
```

```
  unsigned int sum,ans;
  pt = (unsigned short *)buf;
  sum = 0;
  while (n > 1) {
       sum += *pt++;
       n -= 2;
    }
   if (n == 1) {
      oddbyte = 0;
      *((unsigned char *)&oddbyte) = *(unsigned char *)pt;
      sum += oddbyte;
   }
   sum = (sum >> 16) + (sum & 0xFFFF);
   sum += (sum >> 16);
   answer = ~sum;
   ans = answer;
   return(ans);

}



void rl_diag_segment(lpt, rpt)
register diag_info_type *lpt;
register rdiag_info_type *rpt;
{
  switch (rpt->code)
   {
     case INVAL_CONTEXT : lpt->code = invalid_context; break;
```

```
        case REF_CONTEXT   : lpt->code = context_refused; break;
        case UN_DEST       : lpt->code = unknown_destination; break;
        case DEAD_HOST     : lpt->code = dead_host; break;
        case INVAL_ROUTE   : lpt->code = invalid_route;break;
        case REDIRECT      : lpt->code = invalid_route; break;
        case NO_ROUTE      : lpt->code =  cannot_route; break;
        case NO_RESOURCE   : lpt->code = no_resource; break;
        case ERR_PROTOCOL  : lpt->code = protocol_error; break;
        case ERR_MAXDATA   : lpt->code = maxdata_error; break;
        case REL_ROUTE     : lpt->code =  release_route; break;
        case REL_ACK       : lpt->code = release_acknowledge; break;
        case GOOD  :   lpt->code = good; break;
        default : {  printf(" invalid code number\n");
    exit(1);
         }
      }
      lpt->value = rpt->val;
      copydata(lpt->message, rpt->message, Diag_message_max_length);
}

void rl_rrcntl_segment(lpt, rpt)
register rcntl_segment_type *lpt;
register rcontrol_segment_type *rpt;
{
    lpt->rate = rpt->rate;
    lpt->burst = rpt->burst;
    rl_routetype(&(lpt->xroute), &(rpt->xroute));
}
```

```
void rl_first_packet(lpt, rpt)
register lfirst_packet *lpt;
register FIRST *rpt;
{
   rl_header(&(lpt->header), &(rpt->header));
   rl_address_seg(&(lpt->address_segment), &(rpt->address_seg));
   copydata((rpt->data_segment),(lpt->data_segment),
       lpt->header.dlen - addr_max_len);

}

void rl_data_packet(lpt, rpt)
register ldata_packet *lpt;
register DATA *rpt;
{
  rl_header(&(lpt->header), &(rpt->header));
  copydata((rpt->data_segment), lpt->data_segment.data_seg);
}

void rl_control_packet(lpt, rpt)
register lcontrol_packet *lpt;
register CNTL *rpt;

{
  rl_header(&(lpt->header), &(rpt->header));
  rl_control_segment(&(lpt->control_segment), &(rpt->control));
}

void rl_diag_packet(lpt, rpt)
register ldiag_packet *lpt;
```

```
register DIAG *rpt;
{
  rl_header(&(lpt->header), &(rpt->header));
  rl_diag_segment(&(lpt->info), &(rpt->info));
}

void rl_path_packet(lpt,  rpt)
register lpath_packet *lpt;
register PATH *rpt;

{
  rl_header(&(lpt->header), &(rpt->header));
  rl_address_seg(&(lpt->address), &(rpt->address));
}

void rl_rcntl_packet(lpt,rpt)
register lrcntl_packet *lpt;
register RCNTL *rpt;
{
 rl_header(&lpt->header, &rpt->header);
 rl_rrcntl_segment(&lpt->rcntl_segment, &rpt->control);
}

void rl_route_packet(lpt, rpt)
register lroute_packet *lpt;
register ROUTE *rpt;

{
  rl_header(&(lpt->header), &(rpt->header));
  rl_diag_segment(&(lpt->info), &(rpt->info));
```

```
}

void fetch_data(dat)
data_type *dat;
{
register  int i = 0;
register  unsigned char  *ch;
 ch = dat->data;
  while (((i + nextread) < mdata) && (i < datapt)) {
 *ch = arraydata[nextread + i];
  i++;
  ch++;
}
  nextread += i;
  dat->nb_data = i;
}
void displaydata(datap)
data_type datap;
{
 int i;
 printf("amount  %d\n",datap.nb_data);
 for(i = 0; i < datap.nb_data;i++)
     printf("%c",datap.data[i]);
     printf("\n");
}
void store_data(dat)
data_type  dat;
{
 int s,i;
     s = nextwrite;
```

```
      for (i = 0; i < dat.nb_data;i++)
     rbuff[i + s] = dat.data[i];
      nextwrite += i;
}


void init_data()        /* init data storage */
{
  int j,k,i,s;
  for (i =0,s=0; i < mdata;i++,s++) {
    if (s >= 25)
 s = 0;
arraydata[i] = 'a' + s;
   }
  outdata.count = 0;
}



double proba()
{                  /* use time to generate a   random seq number   */
 struct timeval tt;
  if (gettimeofday(&tt, (struct timezone *)0) != 0) {
printf("time error");
exit(1);
     }
  return((tt.tv_usec & 0xF) * 0.01);
}

double local_time()       /*   get time in milliseconds   */
```

```
{
  double s,v;
  struct timeval tv1;
  gettimeofday(&tv1,(struct timezone *) 0);
  if ((tv1.tv_usec -= tv0.tv_usec) < 0) {
     tv1.tv_sec--;
     tv1.tv_usec += 1000000;
   }
   tv1.tv_sec -= tv0.tv_sec;
   v = tv1.tv_sec * 1000 + tv1.tv_usec * 0.001;
   return(v);

}

void send_data(datap,src,dst)
bit_string datap;
int src;
int dst;
{
  xtp_header *xpt;
  int dlen,dsize,i,lflag,adlen,s;
  unsigned  char *addrpt,*ch;
  unsigned  char sbuff[BUFSIZE];
  unsigned int *pt;
  unsigned char *ch0;
  switch(datap.ptype) {
    case First_pak :
      lr_first_packet(&datap.pack.first,(FIRST *)sbuff); dsize = first_len;break;
    case Data_pak  :
      lr_data_packet(&datap.pack.data,(DATA *)sbuff); dsize = data_len;break;
```

```
      case Diag_pak :
        lr_diag_packet(&datap.pack.diag,(DIAG *)sbuff);dsize = diag_len;break;
      case Path_pak :
        lr_path_packet(&datap.pack.path,(PATH *)sbuff);dsize = path_len; break;
      case Rcntl_pak :
        lr_rcontrol_packet(&datap.pack.rcntl,(RCNTL *)sbuff); dsize = rcntl_len; break;
      case Cntl_pak  :
        lr_control_packet(&datap.pack.cntl,(CNTL *)sbuff);dsize = cntl_len;break;
      case Route_pak :
        lr_route_packet(&datap.pack.route,(ROUTE *)sbuff);dsize = route_len;break;
         default : return;
      }
    xpt = (xtp_header *)sbuff;
    xpt->hcheck = checksum(&sbuff[8],28);
option[0] = 131;          /* code no. for loose source  routing */
option[1] = 15;           /*  length  */
option[2] = 4;            /*   pointer */
option[3] = 132;          /*  132.206.3.3        intermediate router  */
option[4] = 206;
option[5] = 3;
option[6] = 3;
option[7] = 128;          /*  128.143.8.99    */
option[8] = 143;
option[9] = 8;
option[10] = 99;
option[11] = 133;         /*  133.155.192.3  */
option[12] = 155;
option[13] = 192;
option[14] = 3;
    i = option[1];
```

```
    option[i - 4] = 132;
    option[i - 3] = 205;
    switch(dst) {        /*   assign  predefined final  destination  */
       case 1 :  option[i - 2] = 62; option[i - 1] = 2;break;
       case 2 :  option[i - 2] = 45; option[i - 1] = 24;
       }
    option[i] = 0;
   /*  set IP option */
   if (setsockopt(sockfd,IPPROTO_IP,IP_OPTIONS,option,32) != 0)
      {
  printf("error set IP options");
  exit(-1);
      }
      bzero((char *)&serv_addr, sizeof(serv_addr));
      serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr("132.206.3.3");  /* set first  hop */
    i = sendto(sockfd,sbuff,dsize,0,(struct sockaddr *)&serv_addr,sizeof(serv_addr));
    if (i < 0) {
if (errno != EWOULDBLOCK) {
    perror("send error socket");
    exit(1);
 }
  }

}

/*   receive  packet  from   the IP  socket  */

void receive_data(datap,src,dst,flag)
bit_string *datap;
```

```
int *src;
int *dst;
int *flag;
{
  bit_string datp;
  ipheader *iph;
  xtp_header *xtph;
  int alen,i,minp,dsize,t,s;
  unsigned int hlen;
  unsigned int *dcheck,*pt;
  unsigned char rbuff[300];
  unsigned char *ppt;
  *flag = 0;
  alen = sizeof(saddr);

/* receive data from IP socket */
  i = recvfrom(sockfd,rbuff,300,0,(struct sockaddr *)&saddr,&alen);

  if (i < 0) {
     if ( errno != EWOULDBLOCK) {
  perror("rec socket error");
  exit(-1);
       }
     else
  return;
   }

  if (i < 20)
    { printf("minimum ip header error\n");
     return;
```

```
    }

  iph = (ipheader *)rbuff;
  hlen =  IP_HEADERLEN(iph);
  if ( i < hlen + 44) {          /*     ip header + xtp header + trailer */
     printf("packet size too small\n");
     return;
   }
   xtph = (xtp_header *)&rbuff[hlen];
     /*  recompute the checksum and verify the validity  of the packet  */
  if (xtph->hcheck != checksum(&rbuff[hlen + 8], 28))
     {
       printf("invalid checksum\n");
       return;
     }
   switch(xtph->cmd.bits.pformat) {
 case p_first : dsize = first_len;break;
 case p_data  : dsize =  data_len;break;
 case p_cntl  : dsize = cntl_len; break;
 case p_path  : dsize = path_len; break;
 case p_diag  : dsize = diag_len; break;
 case p_route : dsize = route_len;break;
 case p_rcntl : dsize = rcntl_len;break;
}
    if (i < hlen + dsize) {
printf("packet size small\n");
return;
     }
      /*   convert   from  real to logical  representation  */
   switch(xtph->cmd.bits.pformat) {
```

```
      case p_first :
   rl_first_packet(&datap->pack.first,(FIRST *)&rbuff[hlen]);
   datap->ptype = First_pak;break;
      case p_data  :
    rl_data_packet(&datap->pack.data,(DATA *)&rbuff[hlen]);
    datap->ptype = Data_pak;break;
    rl_data_packet(&datap->pack.data,(DATA *)&rbuff[hlen]);
      case p_cntl  :
    rl_control_packet(&datap->pack.cntl,(CNTL *)&rbuff[hlen]);
    datap->ptype = Cntl_pak;break;
      case p_path :
    rl_path_packet(&datap->pack.path,(PATH *)&rbuff[hlen]);
    datap->ptype = Path_pak;break;
      case p_diag :
    rl_diag_packet(&datap->pack.diag,(DIAG *)&rbuff[hlen]);
    datap->ptype = Diag_pak;break;
      case p_route :
    rl_route_packet(&datap->pack.route,(ROUTE *)&rbuff[hlen]);
    datap->ptype = Route_pak;break;
      case p_rcntl :
    rl_rcntl_packet(&datap->pack.rcntl,(RCNTL *)&rbuff[hlen]);
    datap->ptype = Rcntl_pak;break;
      default :  printf("invalid packet"); return;
  }
  *flag = 1;   /* set valid data flag  */
}

void  open_socket()     /* open ip socket */
{
  if ((sockfd = socket(AF_INET,SOCK_RAW,36)) < 0) {
```

```
      perror("error open socket");
       exit(-1);
    }
      /*   set  asynchronous I/O    */
  if (fcntl(sockfd,F_SETFL,FNDELAY) < 0) {
    perror("error set socket F_SETFL");
      exit(-1);
    }
}

                /* initialize time  */
void int_time()
{
 if (gettimeofday(&tv0, (struct timezone *) 0) < 0) {
     printf("time error");
     exit(1);
   }
}

void final_time()
{
  int i;
       if (gettimeofday(&tv1, (struct timezone *) 0) < 0) {
    printf("time error");
    exit(1);
 }
}

void printtime(n)
unsigned int n;
```

```
{
  printf("     time %d\n",n);
}

int checkdata()     /* if there is data from the medium  */
{
  int nread,slen;
  unsigned char buff[300];
  nread = 0;
  slen = sizeof(saddr);
  nread = recvfrom(sockfd,buff,300,MSG_PEEK,(struct sockaddr *)&saddr,&slen);
     return(nread);
}
```