MANY-TO-MANY MULTICAST FOR XTP

Tong Ma

A 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

MONTREAL, QUEBEC, CANADA

MARCH 2001

0-612-59336-3

Abstract

Many-To-Many Multicast for XTP

Tong Ma

XTP is a transport protocol. A protocol is an agreement between the communicating parties on how communication is to proceed. For traditional transport layer connections, it is an error free point-to-point channel to provide reliable, cost-effective data transport from the source machine to the destination machine. The XTP is a new protocol with higher data rates, lower bit error rates, user defined service reliability, support multi-communication and latency controls, more suitable for today or the near future as modern networks continue to develop. Compared with traditional point-to-point communications, the XTP can provide efficient multicast data delivery, and the XTP also provides a powerful mechanism for group communication.

Multicasting represents a powerful mechanism for group communication. The multicast service provides a set of hosts to participate in real-time data transfer via a multicast call. This report describes the design and implementation of m by n multicasting in the Sandia XTP.

Acknowledgments

First and foremost, I wish to thank my supervisor, Dr. J. W. Atwood. This report would not have been possible without his unstinting support. Dr. Atwood has made important technical and stylistic suggestions throughout. Moreover, I wish to thank my team members, Tian Fang and Yonglin Jiang, with whom I spent a considerable amount of time finding solutions to the problems which arose while pursuing this project. Finally, a special thanks must go to the Computer Science monitors, who helped me to solve a number of complex system problems and also helped me to configure the Unix environment so that it could handle many-to-many multicast projects.

Table of Contents

List of Figures

List of Tables

1. Introduction

Unicast (one-to-one, point-to-point) services constitute the most popular service offering for many years. Network multicast services have been increasingly used by various continuous media applications. For example, the multicast backbone (Mbone) of the Internet has been used to transport real time audio and video for news, entertainment, distance learning, and other collaborative applications. Multicast communication includes one-to-many (point-to-multipoint) and many-to-many (multipoint-to-multipoint) delivery of data.

The Xpress Transfer Protocol (XTP) is a transport layer protocol designed to provide a wide rage of communication services based on the concept that orthogonal protocol mechanisms can be combined to produce appropriate paradigms within the same basic framework [10]. XTP, as defined [XTP4.0B], provides unicast and one-to-many multicast services.

Ramasivan [6] has proposed extensions to XTP to permit many-to-many operation, which extend the range of applications for which XTP is useful.

Sandia XTP is an object-oriented implementation of XTP designed by Dr. Strayer [Sandia XTP] at the Sandia National Laboratories. Its design is based on extending a set of Transport Protocol Base classes, called the Meta Transport Library (MTL).

This report gives the design and implementation of Ramasivan's proposals within the Sandia XTP implementation.

The report is organized as follows. Chapter 2 describes the existing XTP. Chapter 3 gives a brief introduction to Sandia XTP and Sandia MTL. Chapter 4 outlines the changes that I made in Sandia XTP to provide m by n multicast. Chapter 5 provides more details about the top level that will be added to Sandia XTP to complete the implementation of Ramasivan's proposals and the design of top level built on the basic many-to-many multicast service. Chapter 6 presents performance tests of the result after adding the many-to-many multicast to Sandia XTP. Chapter 7 presents my conclusion and an outline of possible future work.

2. Xpress Transport Protocol (XTP)

2.1 XTP the New Protocol and New Technology

Over the past 30 years, computer networks have undergone exponential growth. Much of this growth occurred through connecting existing networks to the Internet. Consequently, there has been a drive to provide numerous services on the Internet for both companies and individuals, and many industries are making elaborate plans to produce a high-speed data service. As networks grow larger and faster, they become harder to manage. For example, teleconferences require the support of a multicast management program. Because of this exponential growth, special network management systems and protocols have been devised. On the Internet, such programs run the TCP/IP protocol stack through an IP address and the ability to send IP packets to all the other machines. Traditionally, the Internet has had four main applications—Email, News, Remote login and File transfer—yet as high speed networks have become increasingly available, multimedia applications are gaining ascendancy on the Internet, providing such services as remote medical diagnostics, video-phone conversational exchanges, tele-shopping and so on, which require enormous bandwidth and a substantially new communication system. In terms of this rapid change in technology, the XTP (Xpress Transfer Protocol) project began as a project to design and implement a new transport and network level protocol to produce a significant new network system. Compared with traditional point-to-point communications, the XTP can provide efficient multicast data delivery, and it also provides a powerful mechanism for group communication.

2.2 The History of the XTP

Since the early 1980s Dr. Greg Chesson, more recently Chief Scientist of the Research and Development Division of Silicon Graphics Inc, developed the Datakit network at Bell Telephone Laboratories. In the course of this project, he has identified the following problems: (1) the sharing of the CPU between the application program and the protocol processing, (2) the multiplicity of interactions with the operating system (generated by the application, the protocol suite, and the network interface), resulting in many context swaps, and (3) the inflexibility of the protocol suite in providing any service other than a reliable byte stream [2]. Based on his conclusions and experiments, the XTP is defined as a resource to provide what is missing in fundamental communications. The "white papers" defined the XTP protocol in 1987. By the end of 1991, it was being actively developed in many universities, including Concordia University, as well as in research institutes, including Sandia National Laboratories, with the express purpose of improving the XTP published version. Today, the protocol design has been developed by an international group of researchers. This report uses the XTP 1.5.1 version implementation, developed by Dr. Tim Strayer of the academic research sector at Sandia National Laboratories, as the basis for an implementation of the m by n multicast on XTP.

The High Speed Protocol Laboratory of Concordia University is headed by Professor Dr. J. W. Atwood, president of the XTP forum. He has worked in this field for many years. Up to the present, several students have been directed by Professor Dr. J. W. Atwood in his lab to study the XTP. He also actively cooperates with

Sandia National Laboratories and they are jointly responsible for publishing the XTP 4.0B Specification and test, and for the improvement, development, and implementation of XTP 1.5.1.

2.3 The Network OSI Reference Model

Before we present the XTP, we have to look at the OSI reference model, because it outlines the network architecture. Following Figure 1 is the OSI reference model.

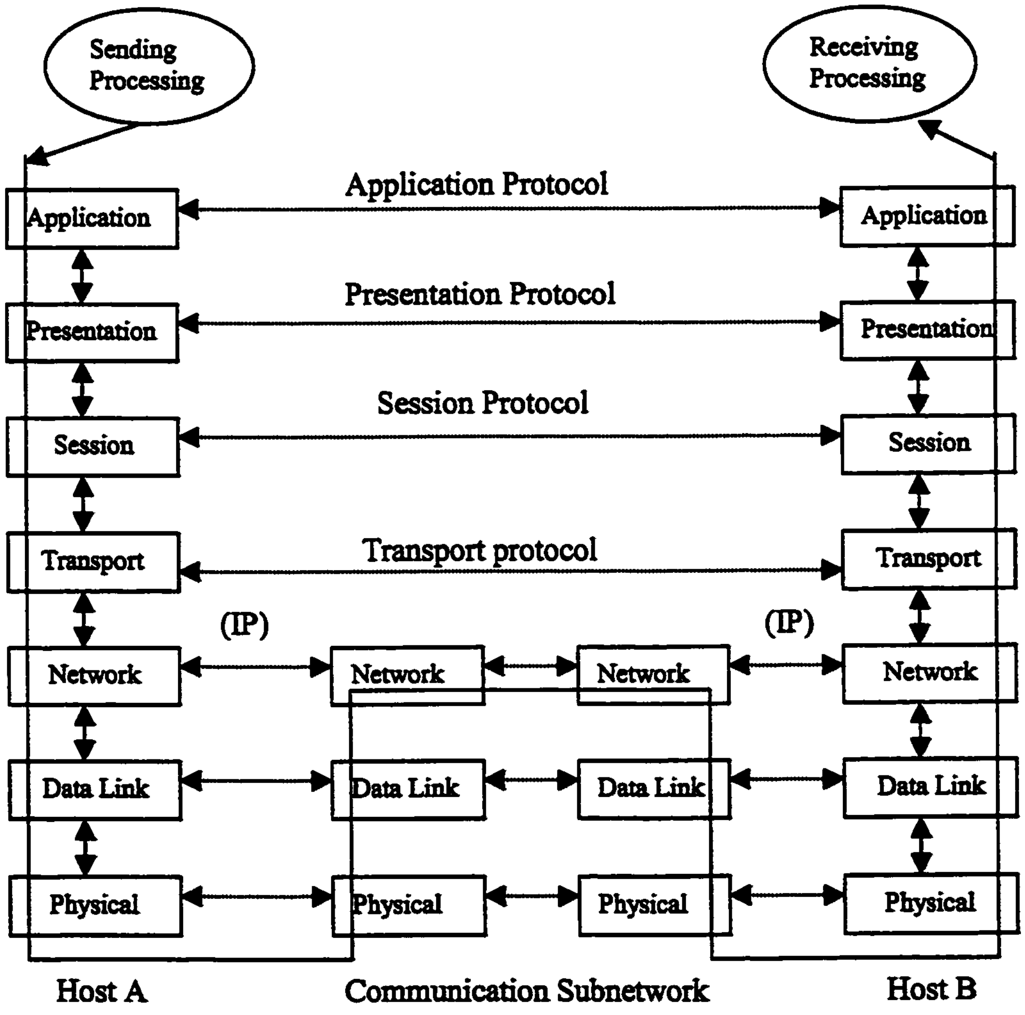


Figure 1. The OSI Reference Model

The OSI (Open System Interconnection) Reference Model, is well known to anyone who works in the field of networks, and it is based on a proposal developed by the International Organization for Standardization as a first step toward international standardization of protocols used in the various layers [3]. It presents network architecture that generates layers and protocols to deal with network open system communication with other systems. By means of the network, communication is achieved through sending messages from an application program on one through layers of software protocol across the network to another machine, and subsequently, through layers protocol software on the receiver side interface. The OSI is demonstrably better for the organization of protocol software and implementation details in the network, since each layer makes decisions to choose an appropriate action based on the transfer message type or address. For example, when a machine receives a message, the network layer must decide whether to process this message according to the address, while another layer has to decide which application should receive the message. Further details about each of those seven layers are not presented in this report, however a clear understanding of OSI is the necessary first step in the understanding of a network protocol like XTP.

2.4 XTP Architecture

The XTP is essentially the same as a small network operating system for users to transfer messages from one machine to another. When the user sets up application programs to send or receive data, the XTP daemon will handle message formats and describe the computer responses through seven successive layers to either transmit data to its destination, or accept data from its source without the user understanding the XTP technology. To gain insight into the XTP daemon we can examine Figure 2, which shows the host architecture of the XTP, while Figure 3 presents an XTP model which demonstrates how the XTP works.

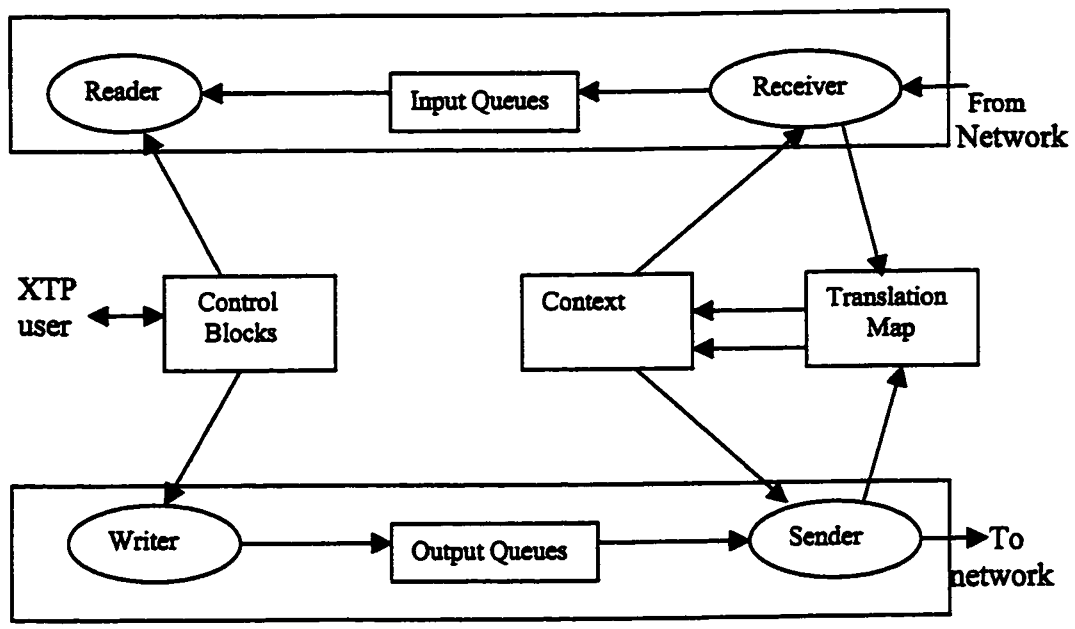


Figure 2. XTP Host Architecture

Figure 2 shows the protocol procedures during data transmission. There are four main processes: (1) the writer process, (2) the reader process, (3) the sender process, and (4) the receiver process in the XTP. The writer and reader processes ensure that the XTP provides an interface to the user to establish the communication between the user and the XTP. Through the XTP interface, the bridge between the user and the XTP, the XTP can receive commands from the user to launch data for traveling, and for outputting received data and information to the user. The sender and receiver process is interfaced through the lower layer, the network data delivery service, for communication between the XTP and the underlying networks.

When the user starts to order service for data transmission, the writer process will output the user commands to put data into output queues and offer the key to match an appropriate context in the sender process. The sender will take the data from output queues, encapsulate it in a packet, set the address and related information in a packet header, then emit this packet with data to its destination. When the receiver process receives packets from the network, it will match the appropriate receiver context from the context database, and parse the packet based on the packet's control information. If the data is complete in this packet, it will be put into receiver input queues. The reader process will unblock the user request and give the data to the user. The XTP context is the XTP end point to hold the stated information about each active XTP connection upon XTP host implementation.

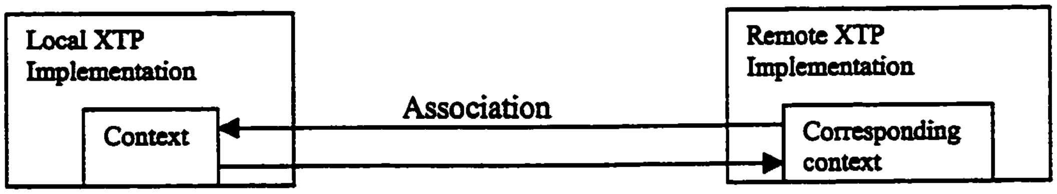


Figure 3. XTP Communication Model

In the XTP, the aggregate of active contexts and the data streams between them are collectively called an association. Each context manages both an outgoing data stream and an incoming data stream as in Figure 3, the XTP communication Model, while the end point context provides most of the real communication services such as addressing, establishing an association, data transfer, flow control, and release connection. For example, when a user wishes to communicate with other hosts, the XTP has to build an association for them. The transmitter sends a FIRST packet to all of the destination end points to request the creation of an association. After the receiver hosts have found a context to handle the first packet so as to set up the association, the data will then be transferred from sender to receiver.

2.5 The XTP Packet Types

A packet is a basic transport and information exchange unit. It contains all of the mechanisms for transferring data and state information from one end point of an association, through all of the intermediate switches, to the other end point or group of end points. The XTP provides the following seven packets.

The FIRST packet carries all of the information necessary to find a listening context at a destination host, and to establish an association between that context and the sending context. The FIRST packet contains the address specification and a traffic specification. The traffic specification is examined by the listening context to see whether it can support the requested type of traffic or not. If it can support this type of traffic, the data in the FIRST packet are given to the context and finally delivered to the user. Basically, the FIRST packet is used to make a group association either in unicast or multicast.

The DATA packet carries user data subsequent to the association's establishment. The DATA packet consists of an information segment with only a data segment. There is *seq* field in the Header, which is used to order and identify bytes in the data stream. The end of the message is marked by the EOM bit in the options field of the Header of a data-bearing packet.

The CNTL packet is the common control packet. It transfers state information between protocol state machines at the end points of an association. The *seq* field holds the sequence number of the next untransmitted byte to be sent on the outgoing data stream of the sender of the packet.

The ECNTL packet is used to convey error control information as well as common control information. It signals contain error conditions encountered during the processing of a packet.

The TCNTL packet is used to convey and negotiate traffic shaping information as well as common control information.

The JCNTL packet is used to join an in-progress multicast communication.

The DIAG packet is used to report pathological conditions that are either fatal or which require corrective action. The information contained within a DIAG packet is designed to give the receiver of the packet some idea as to what caused the error condition so that this receiver can make informed decisions about how to process it.

3. Sandia XTP and MTL

3.1 Introduction to Sandia XTP and the MTL

Sandia XTP (SXTP) is an object-oriented implementation of XTP 4.0B derived from the the Sandia MTL (Meta Transport Library) base class library. Its transport protocols provide data delivery from one transport user to one or more transport users, using the services of the network layer. There are six main classes within the XTP implementation derived from the MTL base classes. The XTP context class, the XTP context-manager class, the XTP packet class, the XTP daemon class, the XTP user interface class and the XTP states machine class, are derived from the MTL. The Sandia XTP functionality resides in the user's application program, which makes requests to the daemon, and the daemon satisfies them. The MTL is designed to present an infrastructure for building transport protocols. There are six main classes within the MTL library package: a daemon class, a packet class, a context class, a state machine class, a context manager class, and a user interface class. The daemon class converts everything into an entity that can be handled by the operating system. Given the MTL's characteristics—portability, adaptability, configurability, and readability—the goals of the MTL permit a developer to rapidly prototype a protocol implementation without any kernel modification or special hardware support, and with as little use of root privilege as possible [9]. Figure 4 and Figure 5, which immediately follow, are diagrams of the main classes of the Sandia XTP and the MTL in daemon and user interface. Daemon is the XTP protocol procedure. The user interface is the path between the user application program and the Sandia XTP daemon.

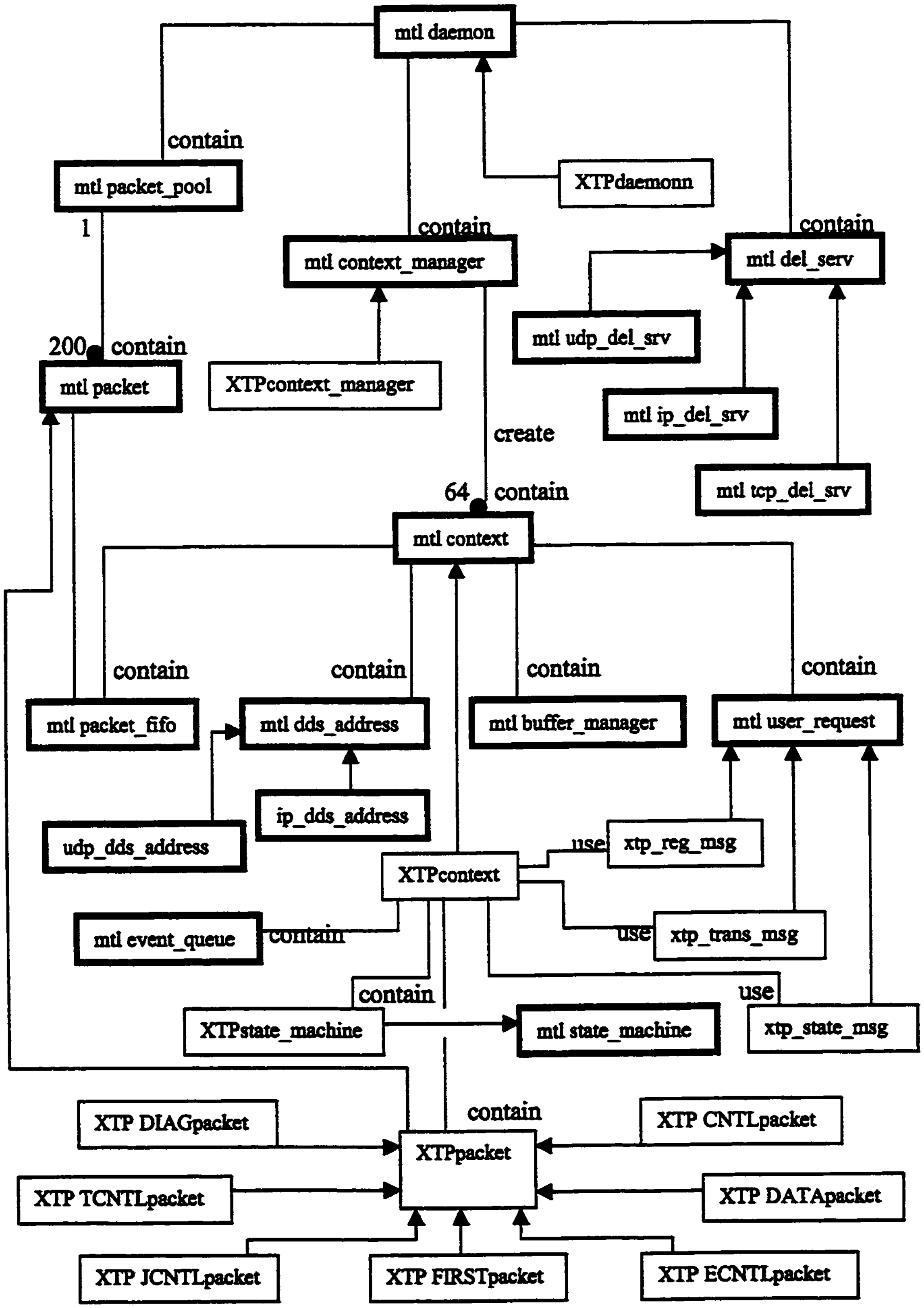


Figure 4. Sandia XTP and MTL Daemon main classes diagram

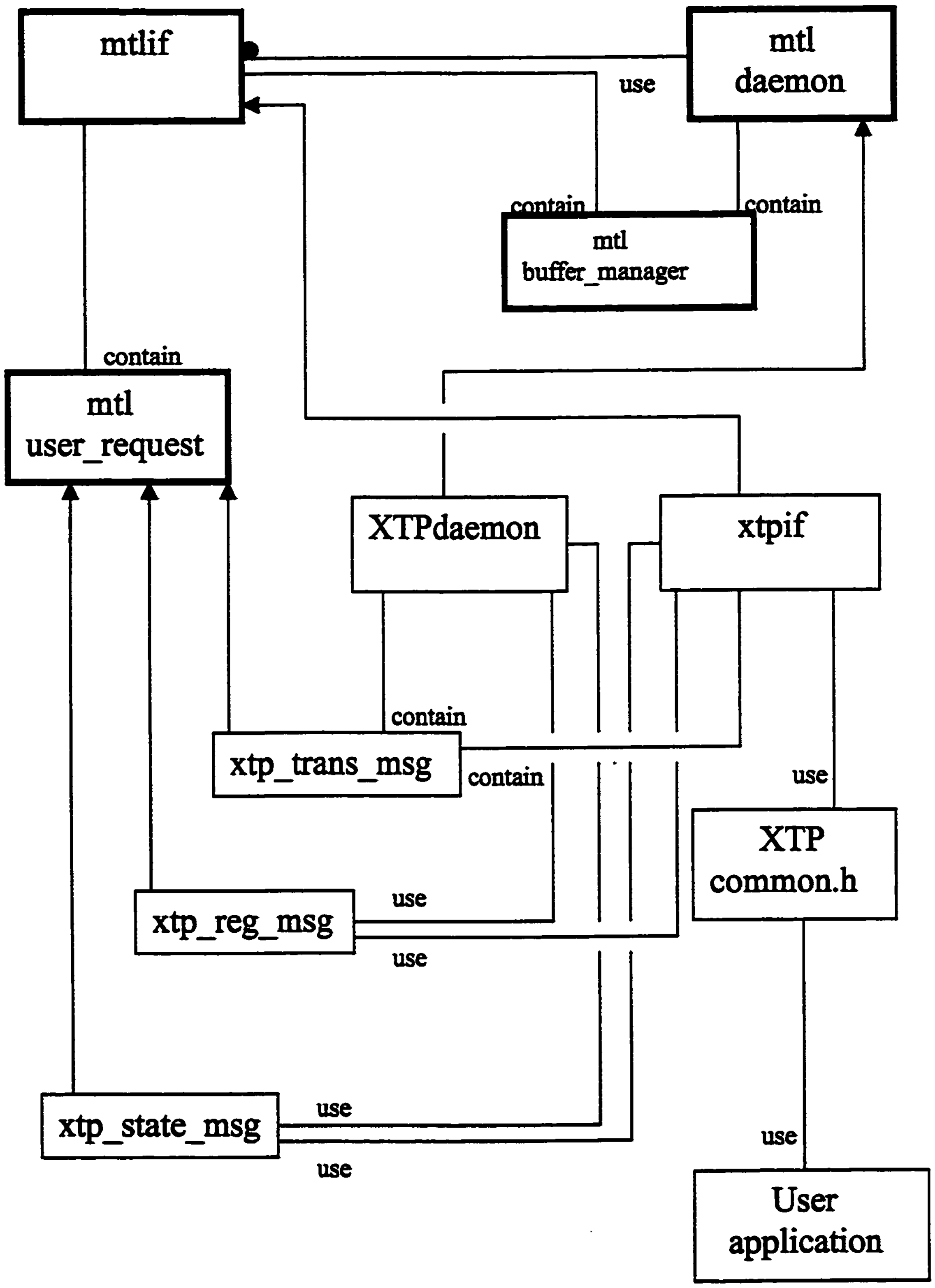


Figure 5. Sandia XTP user interface main classes and application model diagram

3.2 The Main Concepts of Sandia XTP Implementation

The goal of a transport layer protocol such as XTP is to provide the procedures necessary for the transparent, efficient, and reliable delivery of arbitrarily long messages for an arbitrarily long duration between two or more entities not necessarily sharing the same local network segment. In the Sandia XTP there are five procedures that facilitate this goal: association management, flow control, rate control, error control and data transfer.

3.2.1 The Association Management of the Sandia XTP

The association management procedures are invoked when the context receives the FIRST packet to build the association. The XTP association provides a full-duplex communication. Each end point of the association is both a transmitter and a receiver. After the context has received the FIRST packet, the destination address is set with the transmitter address to establish a path between the sender and the receiver.

In the Sandia XTP, the behavior of the changing context indicates the establishment, maintenance and termination of an association. The progression of context states from Quiescent to Active indicates that the context has become an end point of an association, while a transition from Active to Quiescent shows that the association is going to be terminated. The context management is based on different states in the context to handle the different kinds of processing, for example, through processing the DATA packets with the appropriate Active context state or closing the association with an Inactive context state. Figure 6 shows different states in the context during the life of the association from beginning to end, and the following is the explanation of Figure 6's letters.

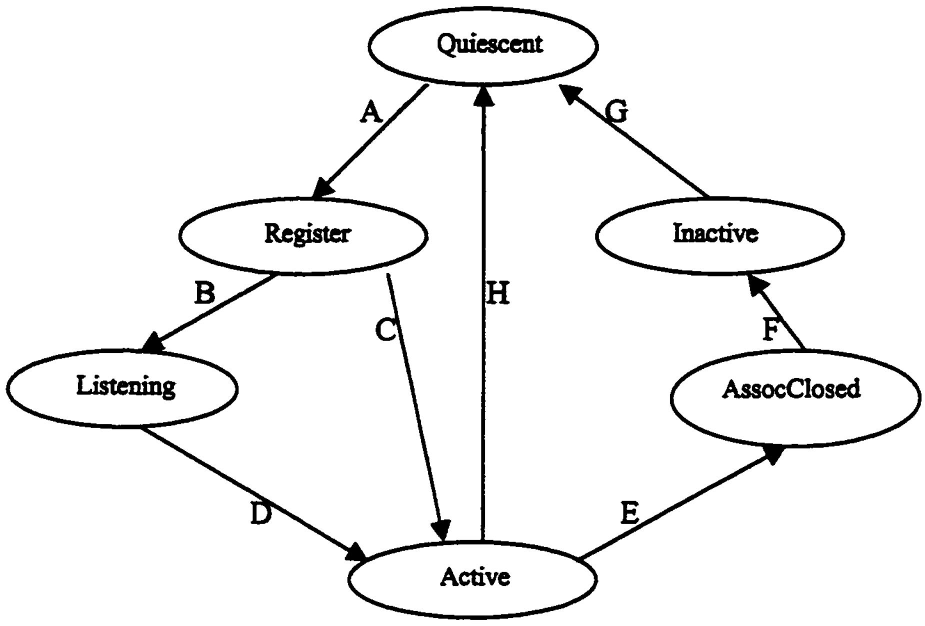


Figure 6. States in Context in Sandia XTP

- Before the XTP end point starts to establish association, the contexts are initialized to have the register state in both transmitter and receiver in the letter A.
- Receiver goes to the listening state from the register state to listen for the FIRST packet in the letter B.
- Transmitter starts to send FIRST packet making the association, then to change its state to Active in letter C.
- Receivers set state to Active when receivers receive FIRST packet to join the association in letter D.
- When the END bit is set, both transmitter and receiver convert outstream and instream to AssocClosed state. The association is closed in the letter E.

- After the context detects that the outstream and instream are in the AssocClosed state, the context will go to the Inactive state to leave the association in the letter F.
- When association is terminated, the context goes back to the Quiescent state for the next user of the letter G.
- If an error is detected, association is aborted, and then the context is converted to Quiescent in the letter H.

The setting of the mechanism states in a context is used to manage the context better, and to recover from errors during data delivery. The XTP provides a rich set of mechanisms, since it recognizes that only the user has sufficient knowledge concerning the application to truly optimize the parameters of a data exchange [2].

3.2.2 Flow Control in the Sandia XTP

The transport layer has to provide flow control and a buffering service. In the XTP, it performs in the same manner as the traditional transport layer functionality in protocols such as TCP. The essential point is that they are in the same layer, and a sliding window or other scheme is needed on each connection to keep a fast transmitter from overrunning a slow receiver. The Sandia XTP sets the flow control according to a sliding window of sequence numbers. Two fields, *alloc* and *rseq,* of the control packet are used in these flow control procedures. The Sandia XTP also has a RES parameter to be sent from the transmitter to indicate how much buffer space is required in the receiver. Consequently, it pushes the receiver to allocate a conservative flow control to be sure that the data is in a safe place, and that no packets are lost due to the lack of a buffer. In contrast, if the NOFLOW bit is set this means the transmitter does not require flow control during data transmission, and so the flow control is disabled.

3.2.3 Rate Control in the Sandia XTP

The rate control in the XTP solves the congestion problem during data transmission. It controls how fast packets can be processed, or consumed at the receiver [1]. This means there is a maximum rate of packets or amount of data at the transmitter for the output stream, and a maximum rate of packets or amount of data at the receiver for the input stream, so that the rate control is able to protect the quality of communication effectively. In the Sandia XTP implementation, there are three parameters for the rate control: *rate*, *burst* and *credit*. The *rate* parameter shows the maximum transfer rate in bytes per second. The *burst* parameter indicates the maximum number of bytes to be emitted in each group of packets, indicating the maximum the receiver can handle. The function of *credit* parameter is to cooperate with burst to measure the outgoing packets. There are certain rules for *credit* and *burst*: (1) if *credit* is zero or negative, add *burst* to the value of *credit*, (2) if *credit* is positive, then replace it with *burst*.

3.2.4 Error Control in the Sandia XTP

The XTP fulfils the same function as a traditional transport layer protocol through the provision of reliable service, and so the error control in the XTP has an important role to play in ensuring this reliable service. The error control is necessary to detect loss of data, unexpected packets, protocol errors and wrong states at the end point. There are three functions in the Sandia XTP, that regulate the error control procedure. The first is a checksum, used to test incoming packets so as to make sure of all of the frames, which are eventually delivered to the network layer at the destination, each in their proper order. The second function checks the sequence number for detecting lost packets. It compares the received data sequence and the emitted data sequence value so as to decide whether the transmitter can retransmit the recovered lost packets. If there is any missing data, it will cause the ECNTL packet to be sent back for retransmission. Sandia XTP supports both a go-back-n algorithm and a selective repeat algorithm. The go-back-n algorithm tells the receiver simply to discard all subsequent packets, sending no acknowledgements for the discarded packet [4]. If the transmitter times out, it will retransmit all unacknowledged packets in order. The selective repeat algorithm requires retention of out-of-order packets, but only requires retransmission of missing packets. Sandia XTP supports only go-back-n operation for multicasting. The third function of the Sandia XTP sets up the DIAG packets to notify the end points. When an error occurs, and there are, for example, unexpected packets or the wrong status, or the mechanism state is in the wrong status in context, or the data fails to be delivered, this will cause DIAG packet to be emitted.

3.2.5 Data Transmission in Sandia XTP

The data transfer procedures implement the movement of data from a transmitting XTP user data buffer space to a remote receiving XTP user data buffer space. This data transfer includes the moving of data from an XTP user data buffer space into a data buffer space, placing it into data fields of either the FIRST packet or subsequent DATA packets, traversing the network, placing the data into receiving XTP data buffer space, and finally, delivering the data into the receiving XTP user data buffer space. Figure 7, demonstrating the Architecture of Sandia XTP and MTL implementation shows how the data is transferred in Sandia XTP.

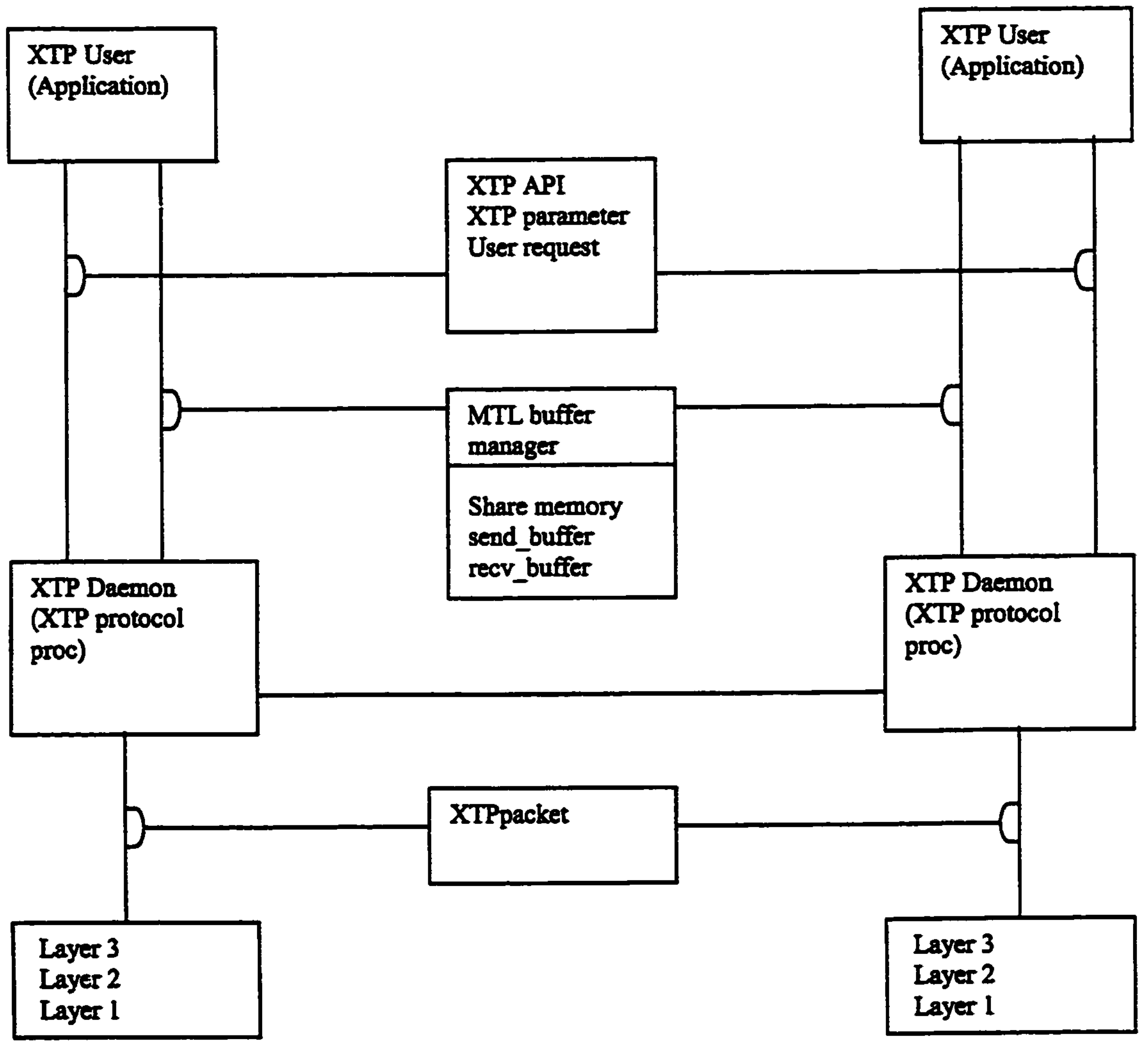


Figure 7. The implementation architecture of Sandia XTP and MTL

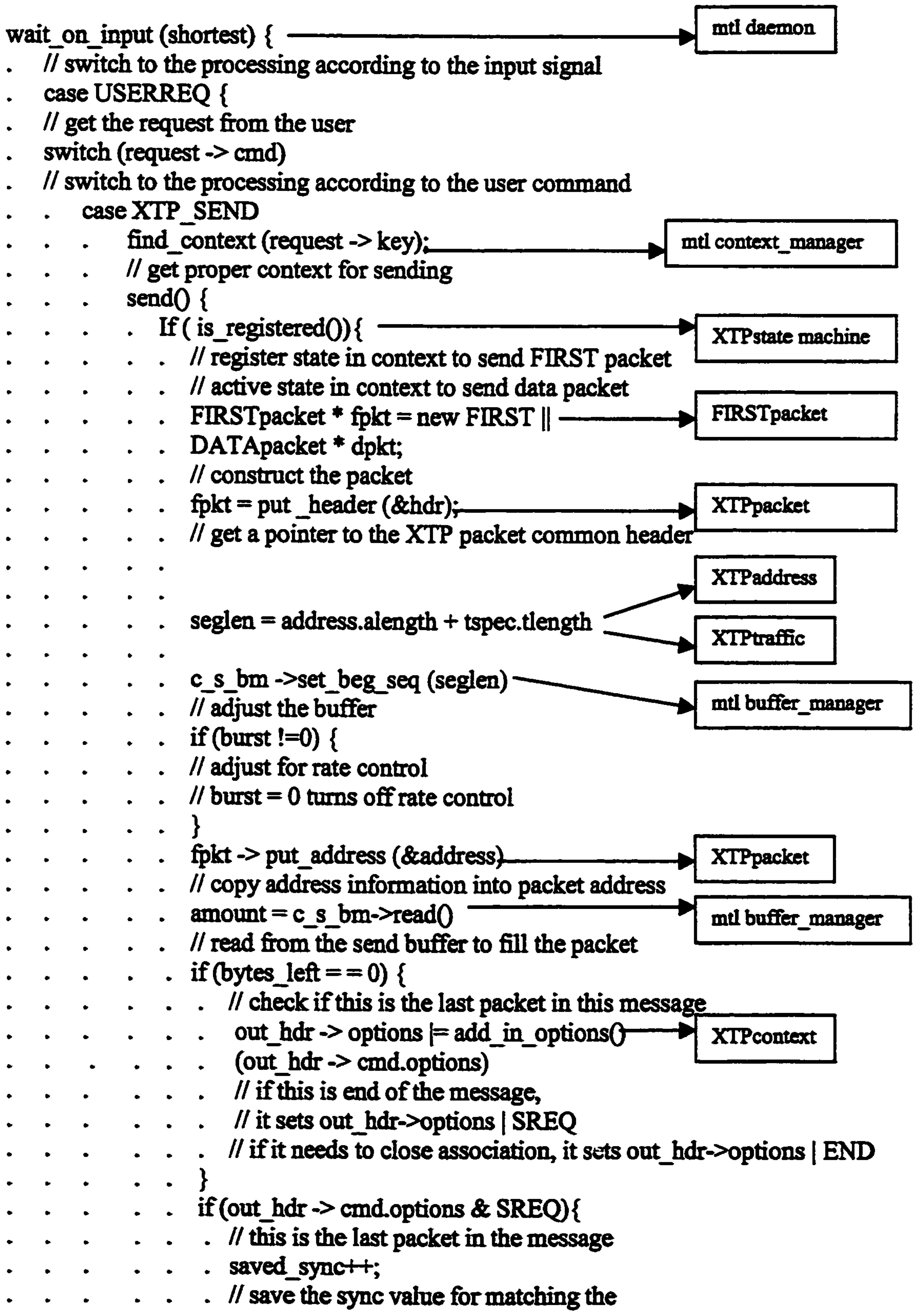
Sandia XTP offers two paths between the daemon and user-interface. One is to transfer the parameters, using API socket, to establish the information exchange communication. For example, the user inputs the command to the daemon. The other path is to write the data into the buffer space with a SEND command or to read the data from the buffer space with a RECEIVE command. In the daemon, to establish communication between the context and the network are the XTP packets. One of the main tasks in a context is to construct a packet for sending or processing a received packet for each single host. Table 1 and 2 below outline the Sandia XTP receive and send algorithms with their related classes in the process of implementation.

Table 1. Sandia XTP receive data procedure algorithm with its related classes

```
wait_on_input(shortest){ ------------------------------------> mtl daemon
// switch to processing according to the signal that the
// daemon received
.   swich (request -> cmd) {
.   // switch to the user commands process
.   .   case XTP_RECEIVE {
.   .   .   find_context (request -> key) ------------------> mtl context_manager
.   .   .   // get proper context for process receiving
.   .   .   receive(){ ------------------------------------> XTPcontext
.   .   .   // start to receive incoming packet
.   .   .   .   if (state_mach.is_Active()) { --------------> XTPstate machine
.   .   .   .   .   event = in_eq.pull(ev_seq); ------------> mtl event_queue
.   .   .   .   .   // the context is ready to be received
.   .   .   .   .   // and processed to pull events
.   .   .   .   .   // off the queue
.   .   .   .   }
.   .   .   .   if (state_mach.is_Listening()) { -----------> XTPstate machine
.   .   .   .   .   return ;
.   .   .   .   .   // wait for FIRST packet coming
.   .   .   .   }
.   .   .   }
.   .   .   set_blk_reg (); --------------------------------> mtl buffer_manager
.   .   .   // set up block to wait for coming packet
.   .   }
.   .   case PACKET {
.   .   .   get_packets_from_dds () { ----------------------> mtl daemon
.   .   .   // get incoming packet from delivery service
.   .   .   // then find appropriate context to handle it
.   .   .   .   handle_new_packet( ); ----------------------> XTPcontext_manager
.   .   .   .   // the packet is put in the proper context
.   .   .   }
.   .   .   satisfy ( ) { ----------------------------------> XTPcontext_manager
.   .   .   //
.   .   .   .   process_CNTL_packet( ); --------------------> XTPcontext
.   .   .   .   process_DIAG_packet( ); --------------------> XTPcontext
.   .   .   .           .
.   .   .   .           .
.   .   .   .   process_FIRST_packet( ) { ------------------> XTPcontext
.   .   .   .   .   hdr.key  = set_hi_bit ( in_hdr -> key)
. . . . . .    //set up packet return key for any return packet


. . . . . .    c_r_bm -> set_beg_seq( );                              mtl buffer_manager
. . . . . .    // set the receive buffer's header, tail
. . . . . .    datalength = c_r_bm -> write ( );
. . . . . .    // get data from received packet and write it to the receive buffer
. . . . . .    process_sequence_number( );                            XTPcontext
. . . . . .    // update context for received packets sequence number
. . . . . .    // set the buffer tail to reflect previously received data
. . . . . .    in_addr = fpkt -> get_address;                         mtl udp_dds_address
. . . . . .    // get packet source address
. . . . . .    c_ucast_dest -> put_hostid(&(in_addr->IPaddr.srchost))
. . . . . .    // set up the destination address for packet to be sent from
. . . . . .    // receiver to transmitter with unicast address          XTPstate machine
. . . . . .    state_mach.trans_on_rcvd ( );
. . . . . .    // make the context states result according to received
. . . . . .    // packet options bits set. If the packet options is set (WCLOSE |
. . . . . .    //RCLOSE)
. . . . . .    // the context has to be turned to Inactive.
. . . . . .    if (this is the last packet) {
. . . . . . .      events = in_hdr -> cmd.options
. . . . . . .      in_eq.put (events, in_hdr-> dlen )                mtl event_queue
. . . . . . .      // put into the queue with sequence number and events
. . . . . .    }
. . . . . .     settle_block_on_data( ) {                             XTPcontext
. . . . . .     // set unblock for user with received information
. . . . . . .       receive (c_blk_reg);
. . . . . . .       // pull off the events from queue to user
. . . . . . .       // return how many bits received information to user.
. . . . . . .       unblock_user ( );                                XTPcontext
. . . . . . .       // reply to user with data and current local model set.
. . . . . .     }
. . . . . .     if ( !(in_hdr -> cmd.options & END )) {
. . . . . . .        send_cntl ( );                                  XTPcontext
. . . . . . .        // this is ACK and ask transmitter about the current status
. . . . . .      }
. . . . . }
. . . . }
. . . }
. . }
. }
}
if_r_bm.read( );                                                      mtl buffer_manager
// xtpif read the received data from the received buffer then output to user.
```

Table 2. Sandia XTP send data procedure algorithm with its related classes



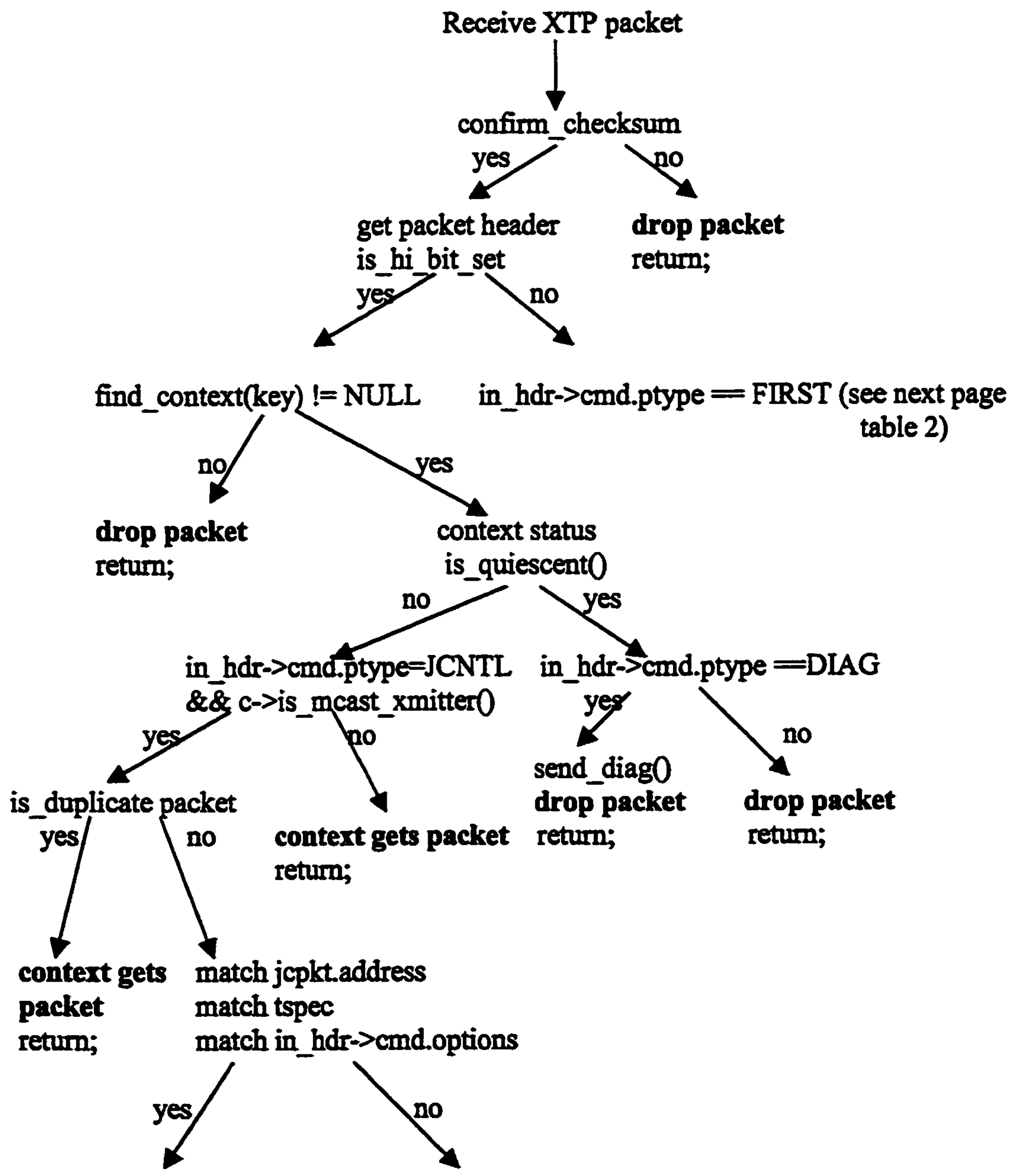
```
wait_on_input (shortest) {
.   // switch to the processing according to the input signal
.   case USERREQ {
.   // get the request from the user
.   switch (request -> cmd)
.   // switch to the processing according to the user command
.   .   case XTP_SEND
.   .   .   find_context (request -> key);
.   .   .   // get proper context for sending
.   .   .   send() {
.   .   .   .   If ( is_registered()){
.   .   .   .   .   // register state in context to send FIRST packet
.   .   .   .   .   // active state in context to send data packet
.   .   .   .   .   FIRSTpacket * fpkt = new FIRST ||
.   .   .   .   .   DATApacket * dpkt;
.   .   .   .   .   // construct the packet
.   .   .   .   .   fpkt = put _header (&hdr);
.   .   .   .   .   // get a pointer to the XTP packet common header
.   .   .   .   .
.   .   .   .   .
.   .   .   .   .   seglen = address.alength + tspec.tlength
.   .   .   .   .
.   .   .   .   .   c_s_bm ->set_beg_seq (seglen)
.   .   .   .   .   // adjust the buffer
.   .   .   .   .   if (burst !=0) {
.   .   .   .   .   // adjust for rate control
.   .   .   .   .   // burst = 0 turns off rate control
.   .   .   .   .   }
.   .   .   .   .   fpkt -> put_address (&address)
.   .   .   .   .   // copy address information into packet address
.   .   .   .   .   amount = c_s_bm->read()
.   .   .   .   .   // read from the send buffer to fill the packet
.   .   .   .   .   if (bytes_left = = 0) {
.   .   .   .   .   .   // check if this is the last packet in this message
.   .   .   .   .   .   out_hdr -> options |= add_in_options()
.   .   .   .   .   .   (out_hdr -> cmd.options)
.   .   .   .   .   .   // if this is end of the message,
.   .   .   .   .   .   // it sets out_hdr->options | SREQ
.   .   .   .   .   .   // if it needs to close association, it sets out_hdr->options | END
.   .   .   .   .   }
.   .   .   .   .   if (out_hdr -> cmd.options & SREQ){
.   .   .   .   .   .   // this is the last packet in the message
.   .   .   .   .   .   saved_sync++;
.   .   .   .   .   .   // save the sync value for matching the
. . . . . . . // returned control packet
. . . . . . . start_wtimer(); ──────────────────► XTPcontext
. . . . . . . //
. . . . . . }
. . . . . . int state_changed = state_mach.trans_on_sent
. . . . . .  (out_hdr -> cmd.options);
. . . . . . // set context states according to cmd.options
. . . . . . int res = fpkt -> send (); ─────────────► XTPpacket
. . . . . . // launch the packet with a multicast address in a
. . . . . . // multicast association
. . . . . . // launch the packet with a unicast address in a
. . . . . . //unicast association
. . . . . . if (bytes = = 0) {
. . . . . . // there are no more bytes left to be sent
. . . . . . set_sent_modes (xtr -> options);
. . . . . . // set sent modes
. . . . . . return (xtr-> data_len);
. . . . . . // return to user the number of  bytes completely sent
. . . . . }
. . . . . num_packets ++;
. . . . . // keep track of the number of packets sent
. . . . . if (is_active()) {
. . . . . . // start to send data packet as above
. . . . . . //presented the algorithm
. . . . . .       .
. . . . . .       .
. . . . . .  int res = dpkt.send ();────────────────► XTPpacket
. . . . . .  // launch the packet
. . . . . .       .
. . . . . .       .
. . . . . .   return (xtr -> data_len);
. . . . . .   // return to user the number of  bytes completely sent
. . . . . }
. . . . }
. . . }
. . case RPLY
. . .    send_reply(reqmsg, &user_addr);
. . .    // reply to user without ACK
. . if (!daemon_stop)
. . .    get_packets_from_dds()
. . .     // get ACK CNTL packet from del_service
. . .    d_cm-> satisfy()
. . .     // process received ACK packet to determine whether retransmit or
. . .     // reply to user
. . }
   }
}
```

Data is transferred to a remote Sandia XTP user in the Sandia XTP implementation. The local host issues an input command, the data is placed in Sandia XTP buffers, and when appropriate, is copied into the data fields of outgoing data-bearing packets. The packet contains the sequence number to ensure all the data have been sent. When the data-bearing packet is received by the destination, its contents are placed into an XTP data buffer according to the sequence number in the *seq* field. The receiving context keeps track of outstanding data and, when requested via an input command, delivers contiguous data to the Sandia XTP user. The Sandia XTP uses a special mechanism to mark the end of a message. The user data is conceptually one or more messages, where a message is considered as a user-defined grouping of the data. The association is equipped to handle an arbitrary number of messages over its lifetime.

Data transfers between two Sandia XTP users are packet unit transmissions. When a host receives incoming packets, it is important to find a proper context to handle the packet, and then let it process the packet. The Sandia XTP establishes the rule with the key value in the packet header. The key value is a 64-bit unsigned int. When the transmitter emits the packet, the packet key value is equal to the send context key value. If a receiver wants to send a packet to the transmitter, the packet's key value will have its high bit set "on" as a return key. After the FIRST packet has landed in the receiver context, the context sets a high bit for the FIRST packet's key and stores it in the context for the returning packet. The JCNTL packet with no high bit set key is then called a late join case.

Handle new packet function in the XTPcontext_manager class performs the task of handling the incoming packet in the proper context. Figures 8, 9 and 10 below are the decision tables used to handle new packet function.

Data transfer may not be limited to only two users. In fact, the Sandia XTP supports one-to-many and many-to-many data flows. The many–to-many data flow is a significant new feature added in this project.



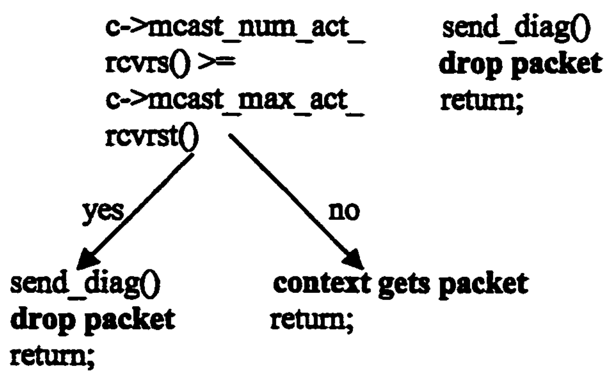


Figure 8. Handle new packet decision table 1

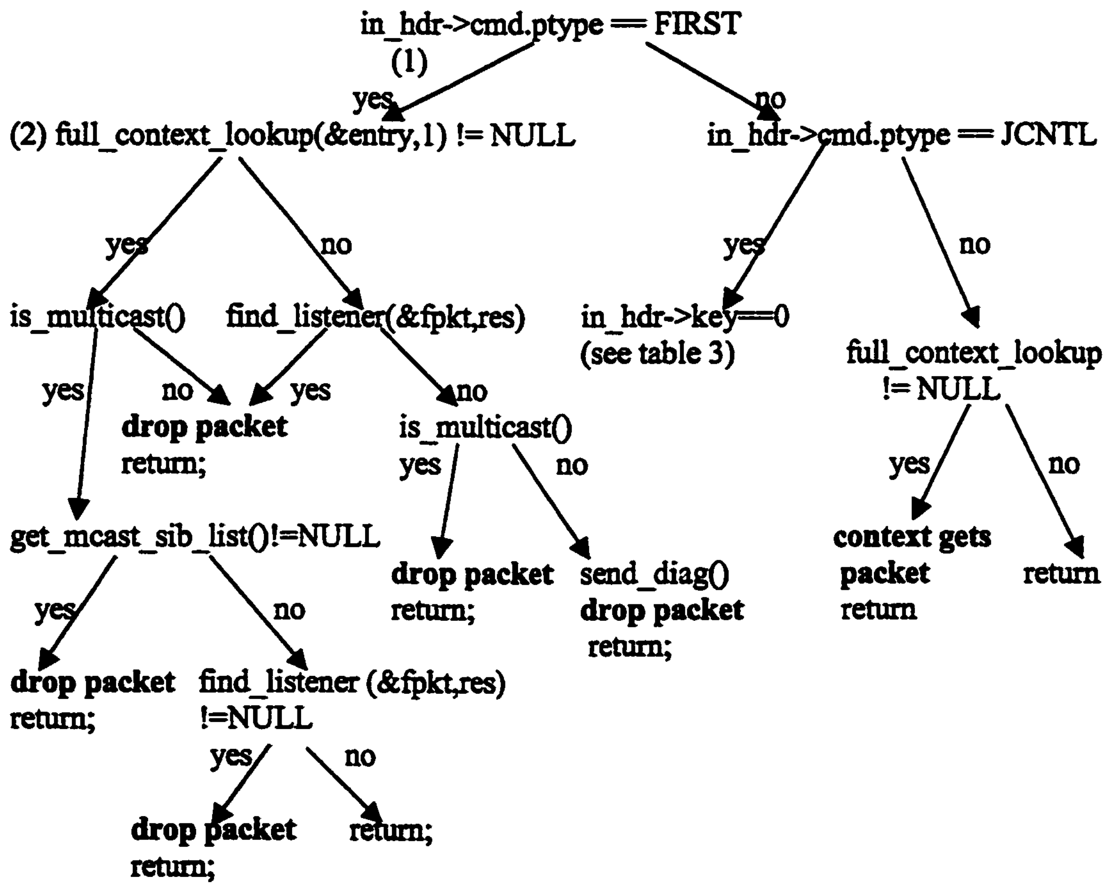
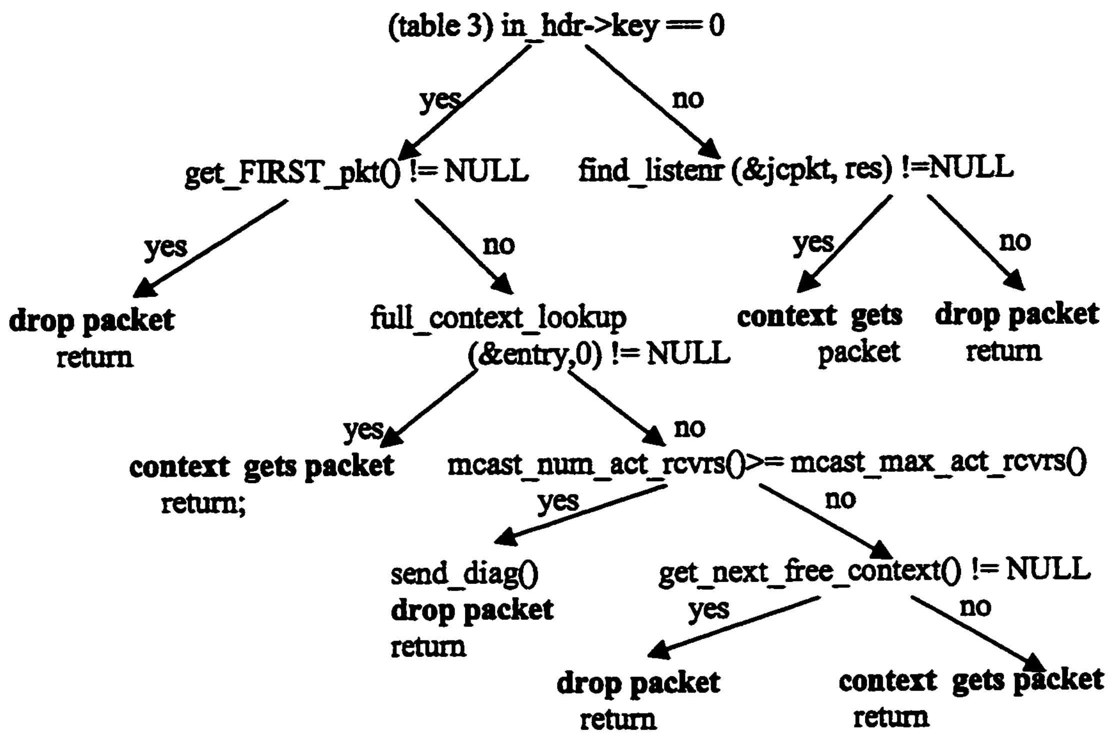


Figure 9. Handle new packet decision table 2



(table 3) in_hdr->key == 0
yes
no
get_FIRST_pkt() != NULL
find_listenr (&jcpkt, res) !=NULL
yes
no
yes
no
drop packet return
full_context_lookup (&entry,0) != NULL
context gets packet
drop packet return
yes
no
context gets packet return;
mcast_num_act_rcvrs()>= mcast_max_act_rcvrs()
yes
no
send_diag() drop packet return
get_next_free_context() != NULL
yes
no
drop packet return
context gets packet return

Figure 10. Handle new packet decision table 3

3.3 Addressing in the Sandia XTP

When an application process wishes to set up a connection to a remote application process, it must specify which one to connect to and which destination is defined for ongoing packets. It is the same, as when we send a letter to someone, we have to write a valid address on the envelope in order for the mailman to deliver it. On the Internet, the IP address and the local ports of each host identify the desired destination. Sandia XTP operates with a dozen network addresses including IP, the Internet protocol. This implies the Sandia XTP can map popular network addresses, which can be indicated by the user.

3.4 Unicast in the Sandia XTP

The Sandia XTP provides the functionality of unicast, which permits two end points to communicate with each other. This provides a high degree of functionality through protocol mechanisms. Once an association is established, a listening context has received a FIRST packet and sent back TCNTL packet as an acknowledgement, then the initial contexts in both hosts have moved into an active state. Once an incoming packet arrives at the appropriate active context during the transmission, this context can be the receive context, since after one host finishes sending data, the other host can use this context to send a data packet to the initial transmitter. The association is created once by the first transmitter to send the FIRST packet, and is terminated by exchanging WCLOSE, RCLOSE and END bits in the packets.

3.5 Multicast in the Sandia XTP

The XTP multicast is an abstraction of hardware multicast. It is not like broadcasting. XTP multicast permits each machine to choose whether to participate in the multicast or not. When a group of machines want to communicate, they choose one multicast address to use for communication. After establishing an XTP group communication association, the XTP sets a context for recognizing the selected multicast address, then all the machines in the group will receive a copy of every packet sent to this multicast address. The designers of the Xpress Transfer Protocol strove to balance the need for group communication and support with the desire to avoid distorting or overburdening the protocol with special case processing for the multiparty case [2]. Before this report the XTP unicast and the one by n multicast already existed in the Sandia XTP implementation. The Sandia XTP provides the one by n multicast and unicast functionality—to let the user choose between different options in his association. In the Sandia XTP implementation, if the user sets the *MULTI* parameter in options, the Sandia XTP begins its multicast process for group communication. Since this process is a transport layer multicast (not a datalink multicast or broadcast), it is a reliable peer-to-peer multicast, providing flow control, rate control and error control transmission of arbitrary messages of arbitrary sized groups [3]. The unicast is also used in the established Sandia XTP multicast association, in processing the ACK packets from receiver to transmitter.

3.5.1 Addressing of the Multicast in the Sandia XTP

In the Sandia XTP, the communication between machine and machine is controlled through packets. The packet can find its path according to its encapsulated destination address information. This address information is necessary to establish a path between the Sandia XTP end points. The Sandia XTP provides a global address, expressed in a number of existing address formats [2]. In the Sandia XTP multicast, the value to be set in the address segment of the FIRST packet is the multicast address corresponding to the set of destinations. In this report, I use an IP multicast address as the group address and as the destination address in the application program. The IP multicast address is an Internet class D address between 224.0.0.0 and 239.255.255.255. IP multicast addresses can be used only as destination addresses. They can never appear in the source address field of a datagram. In order to map an IP multicast address to the corresponding Ethernet multicast address, the low-order 23 bits of the IP multicast address must be inserted into the low-order 23 bits of the special Ethernet multicast address 01-00-5E-00-00-00 [5]. Since group membership is associated with a particular network address, an application program must specify a particular network address when it asks to join a multicast group. The Sandia XTP multicast is designed for networks with a physical broadcast medium, and the destination address of the FIRST packet is a multicast or broadcast address, derived from the group address in the Address Segment. It is assumed that address assignments are accomplished through an outside mechanism, such as agreement among peers or an address management protocol [1].

3.5.2 One by N Multicast in Sandia XTP

Sandia XTP one by n multicast is a group communication. One transmitter and many receivers are located in one group. Since this is a transport multicast – rather than a data-link multicast or broadcast – flow control, rate control, and error control procedures are applied in the transmission of arbitrary size messages to arbitrary size groups [1]. Following the transport service primitives, Sandia XTP provides a mechanism to simulate the primitives in implementation which readily permits the user to set application programs for establishing, using, and releasing connections. Consequently, reliable multicast transport can be derived from the participants who can group together in one association in a transport service to access multicast service. This service uses the same best-effort datagram delivery, which can be lost, delayed, or duplicated, but because XTP is a transport layer protocol, it requires reliable transmission. Consequently, the transmitter has to keep the messages until it gets all of the acknowledgements it requires from its group members. Figure 11 below shows the architecture of the one by n multicast in the Sandia XTP.

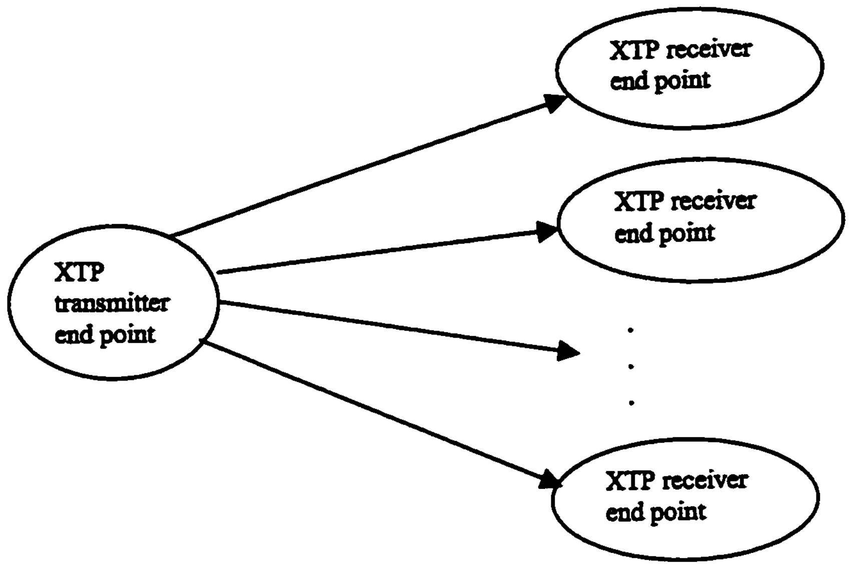


Figure 11. The Sandia XTP One by N Multicast Architecture

One by n multicast encapsulates the group communication management in its protocol mechanisms, which are sufficient to manage the entire group. In order to initialize a multicast association it is necessary to send a FIRST packet with association information, so that one or more slave hosts can receive it. After the host context is set up by the FIRST packet, it knows the specified group multicast address and other packets can be sent to that group address. Each receiver in the multicast group acknowledges the initiation to join by returning a JCNTL packet. The transmitting host then issues a JCNTL to each receiver, to complete the exchange of necessary information and fully establish the multicast association.

4. The Proposed Changes in the Sandia XTP

4.1 The Extension of One by N Multicast to M by N Multicast

The existing Sandia implementation provides unicast communication and one by n multicast as defined in the XTP specification 4.0B. The goal of this major report project is to implement the changes proposed in Ramasivan's thesis, which specify the extensions necessary to permit m by n XTP communication. The implementation of XTP m by n multicast is based on the existing Sandia XTP unicast and one by n multicast implementation. It extends their functionality and provides essential new features. These changes imply that there are many transmitters and receivers in one communication group. Figure 12 below presents the architecture of the m by n multicast.

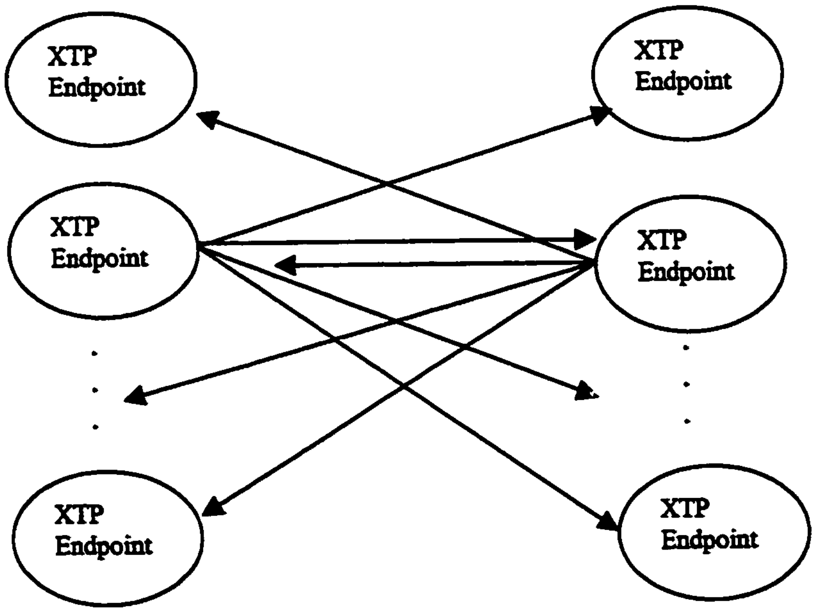


Figure 12. The M by N Multicast Architecture

The XTP m by n multicast permits many members in one association to both send and receive, as with peer to peer transmission. The m by n multicast features transport layer communication including flow control, rate control, and error control procedures. But the XTP host management has to extend to group context management in a single host from one by n multicast in the Sandia XTP. The m by n multicast's group address management continues to adopt one by n address management. The main differences between one by n and m by n multicast in XTP can be outlined as follows.

The first difference between one by n and m by n is that one by n has one context in each host either sending or receiving data, while with the m by n multicast each host can hold many generated contexts so as to receive data from the different machines in an association. In each single host, one receive context corresponds to one transmitter in the association.

The second difference between one by n and m by n is that the m by n multicast case not only features a master host in an association, but also has a master context in each host to control the slave contexts in each context group during the data transmission.

The third difference concerns group communication management. With the m by n multicast, the association is not closed until the last transmitter has finished sending a message, but with the one by n multicast, because there is only one transmitter in an association, once the data source has finished its delivery data, the association terminates.

From the above differences, we can deduce that the m by n multicast association has been devised to transform and extend unicast and one by n multicast in the Sandia XTP, and to build on all the advantages of functionality built into unicast and one by n multicast association.

4.2 Rate Control, Flow Control and Error Control in M by N Multicast

The m by n multicast obeys the same rules of flow control, rate control and error control as the Sandia XTP one by n multicast. Another difference between the Sandia XTP one by n multicast and the new proposed m by n multicast concerns retransmission and error detection procedures. First, RCLOSE and WCLOSE will no longer be forced in the outgoing and return packets, respectively. Second, the context states continue in Active during the entire data transmission. Third, the WCLOSE bit will now be set after one transmitter has finished sending a message. All of the above considerations will mean that the existing Sandia XTP will identify these features as errors, but they are not errors in the m by n multicast.

4.3 Setting the Context for M by N Multicast

The XTP context presents information to the XTP endpoint. To set an appropriate context in each host will be the first step to be performed by the m by n multicast. In the Sandia XTP, one host has one context to record information from the end point, and the context is only used to send or receive data since the host can either send or receive. However, the auxiliary context can be retained so that the host can handle special cases. For example, it can handle the JCNTL packet for join and later join cases. In m by n multicast, every host has the ability to send and receive after the association has been established. This means that each host needs at least two contexts to accomplish data transmission, one for sending and one for receiving. When the user sends commands to the daemon to make an association, two contexts will be created in one host, one being the send context, the other being the receive context. The receive context will be used to receive the first FIRST packet and the send context will be used so that the daemon can send data. Another reason for setting the two contexts before establishing the association is to keep the original information. For example, the receive context will change its destination address from the multicast address to the data source, which is the transmitter's address, after it has received the FIRST packet. After the association has been established, when a host receives an incoming packet, the handle_new_packet function of the XTP context manager decides which context is qualified to take the packet according to the key value of the packet's header. If there is no proper context in the context list, it will generate a new context to handle this case.

The algorithm of the m by n multicast comprises one master host and many slave hosts in one association. The slaves remain under the master host's control. The master host is the host, which sends the first FIRST packet to build the association. No one can close the association except the master host, which emits the packet with the WCLOSE value set. In each host, there is one master context and many slave contexts, which correspond to each of the data source hosts. This indicates that the numbers of transmitters in an association are equal to the number of slave contexts in one single host. In each host, the slave contexts are generated by the master context and they are placed in the m by n sibling context table. The master context is the send context for each host since it retains the original information. There are two places through which to generate the context during m by n multicast processing. One is located in the daemon dispatch function; the other is in the handle_new_packet function. When the user issues a command (send or receive) to the daemon with *request*, the daemon will find the initialized context for the user according to the key value in the *request*. At this time, if the context is in the Register state for the transmitter or the Listen state for the receiver before building the association, then they begin to generate a context.

The following seven points describe the main changes in Sandia XTP implementation for the m by n multicast.

1. Three main new principal parameters were added in the context class, *master_host*, *master_context*, and *m_key*. The *master_host* parameter can indicate whether this host is the master host or not. It is set to "1" when the master host sends the FIRST packet. Otherwise, it is set to "0". The *master_context* parameter indicates whether this context is the master_context or not in this host. The *m_key* parameter identifies group contexts for each host. All the contexts that are part of the same association in a host have an identical m_key value. The *m_key* is equal to the key value of the first initialized context key in each host. This is set when the context is generated.

2. In slave hosts, the master context (send context) is created before the slave host receives the first FIRST packet. When the receive context starts to receive the FIRST packet, the *master_host* parameter of the receive context has to be set at "0", and then the context is set from the information received by the FIRST packet, such as the path address establishment for the unicast address between receiver and sender.

3. The state of each generated Context has to be set to an Active state, because the new context does not need to be initialized again and the value is copied from the initialized master context. Meanwhile, the context in the Active state means that communication is enabled in the existing association.

4. Another important new parameter added in context is the *return_key* (data source key). Since the full_context_look_up function can not cover the m by n muticast so as to map the proper context to handle the FIRST packet, there were two main reasons for adding this parameter: to cooperate with the new functions set_nbym_rcv_list and find_nbym_sib_next to map the proper context so as to handle the FIRST packet. Consequently, when the *return_key* is zero, this means that the receive context is not being used and the daemon can then use this context to receive incoming packet. Moreover, in duplicate the packet case, if the incoming packet key value is the same as the *return_key* value, the context does not take this_packet and the packet is consequently dropped. The *return_key* parameter is set to the FIRST packet key value when the context receives the FIRST packet from the transmitter. The principal reason for the duplicate packets is to ensure that as many potential receivers as possible are attracted to the group and to deliberately send the duplicate FIRST packets upon expiration of the WTIMER from the transmitter. Similarly, the join receiver is permitted to deliberately send duplicate JCNTL (key =0) packets after it receives the ACK from the transmitter. Regarding the DATA packet, CNTL packet and other XTP packets, the full_context_look_up will be used to map the proper context in an m by n multicast. The explanatory detail can be found in section 4.4.

5. A new data structure nbym_sibling_info was added in context to list the group contexts in one host. This was necessary to put the generated contexts on an m by n sibling list table located around their own host. The master context is the header on the table. When the daemon wants to get obtain an appropriate context to handle the different cases, it looks at the m by n sibling list to identify a suitable context. If no proper context can be found, the daemon will generate a context to handle that case, then put it on the m by n sibling list.

6. The buffer, which has been created by the first context in its initialization, has to be attached to each generated context for receiving data, since every context has to use one created buffer in a single host. The start point and end point of the buffer segment have to be considered. When the context sets the correct address and segment length in the buffer, the user can read the correct data from the buffer in the XTP user interface.

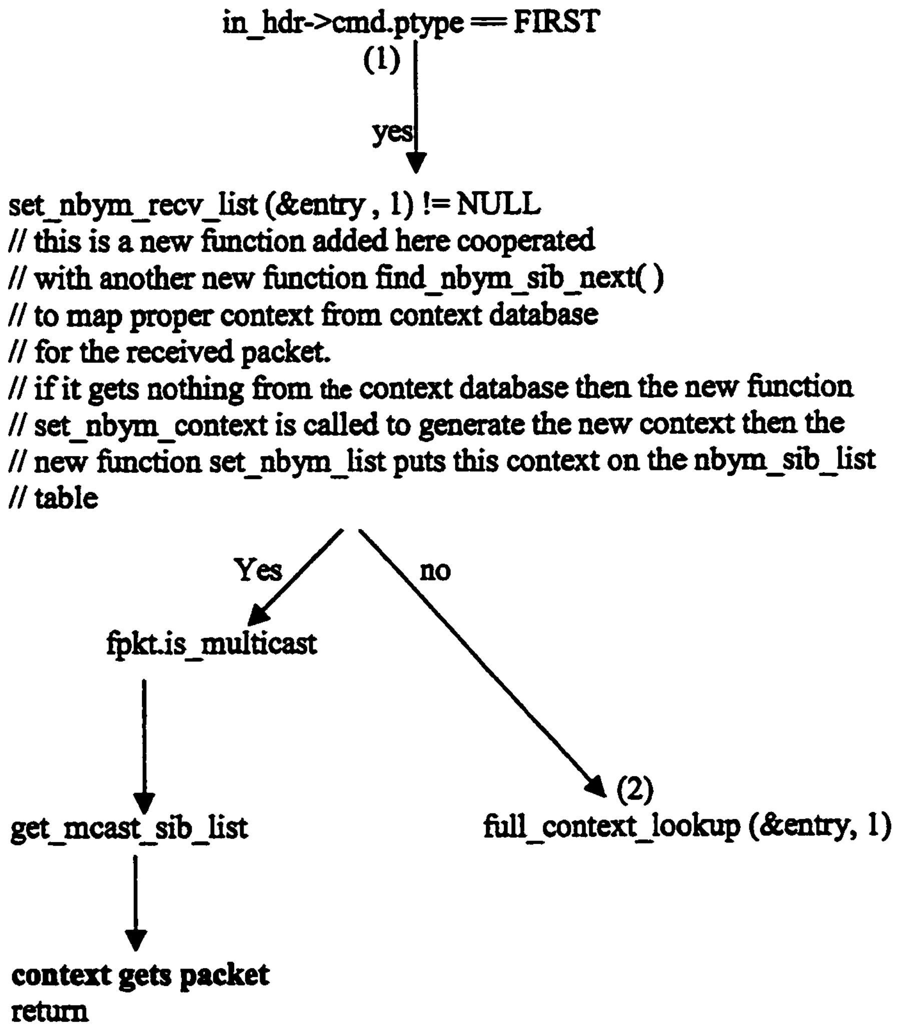
7. In the Sandia XTP, the RCLOSE bit indicates that the sender's incoming data stream is closed and the RCLOSE bit has to be set in each outgoing packet from the sender, starting with the FIRST packet. In the m by n multicast, the RCLOSE bit will not be set in the FIRST packet and not every outgoing packet from the sender has to be set in the RCLOSE. Section 4.7, provides more details concerning RCLOSE, WCLOSE and END bit settings in m by n multicast.

4.4 The Major Changes in the Sandia XTP for M by N Multicast

To satisfy the m by n multicast, the main aspects of context management, process packet, XTP interface and handle WTIMER have to be modified and extended. The following six points indicate the main changes that have been executed so far.

1. The Full_context_look_up function works with a handle_new_packet function. When the packet is sent, the full_context_look_up function maps all the active contexts so as to get a proper context to handle the packet according to the context states and the key value of the packet. However, the full_context_look_up function is not powerful enough to cover the m by n multicast case, and after association establishment the FIRST packet is sent again from another transmitter but the full_context_look_up function does not work for this case. It consequently needs to add set_nbym_rcv_list function and find_nbym_sib_next function so as to cooperate with the full_context_look_up function to map the proper context for the incoming packet. After the association is established, except for the FIRST packet, the full_context_look_up function keeps mapping the proper context for the other packets in an m by n multicast case.
2. In context management, the handle new packet function handles each received new packet through a proper context. In terms of original design, there are three rules to be used here to find the proper context to handle the new packet. First, does the packet's key (return_key) have the high bit set? If the packet's key is high bit set, this means the packet is a return packet from receiver to transmitter with the unicast address so the find_context function in the MTL context manage class is used here to get the transmitter context with the return key to handle the packet. There are no FIRST and DATA packets with high bit set key because the FIRST and DATA packet comes from the transmitter to the receiver. Second, if the high bit is not set in the packet's key, this means the packet comes from the transmitter. If the packet is the FIRST packet to make association, the

find_listening_context function is used here to get all of the listening status contexts to handle the packet. If the packet is DATA or another packet, the full_context_lookup function is used here to map all of the context database to ensure the proper contexts to handle the packets. Third, if the packet's key is zero in the JCNTL packet, this implies it is a join packet from the receiver to the sender. The difference in performance between the one by n case and the m by n case in handle_new_packet function is as follows. With the one by n multicast case, the context is in a listening state to receive the FIRST packet and it can only receive one FIRST packet. But with the m by n multicast, only one host can send the FIRST packet in the association, while the receive context is placed in an Active state to handle the FIRST packet. To deal with new situations, the handle_new_packet function needs to be modified so as to find the appropriate contexts to receive the packets, or to generate a new context for them. The m by n sibling list has to be used here to identify the appropriate received context. The algorithm is necessary if the FIRST packet comes and the association is already established, and the handle_new_packet has to fulfil a new function (set_nbym_rcv_list) by looking up each host's m by n sibling list to check each receive context. If the context parameter *return_key* (data source key) is zero then the context starts to handle the packet. If there is no proper context to be found from the m by n sibling list, a new context is subsequently generated to handle the packet. If the FIRST packet is the duplicated packet, then the packet's key is the same as the context *return_key*, and the packet will be dropped. There is no context to receive the duplicate FIRST packet. To handle DATA and other packets, the algorithm for the m by n multicast is the same as that for the one by n in this (handle_new_packet) function. Figure 14 below shows the changes in the handle_new_packet function.



```
in_hdr->cmd.ptype == FIRST
(1)
yes
set_nbym_recv_list (&entry, 1) != NULL
// this is a new function added here cooperated
// with another new function find_nbym_sib_next( )
// to map proper context from context database
// for the received packet.
// if it gets nothing from the context database then the new function
// set_nbym_context is called to generate the new context then the
// new function set_nbym_list puts this context on the nbym_sib_list
// table
Yes
no
fpkt.is_multicast
get_mcast_sib_list
context gets packet
return
(2)
full_context_lookup (&entry, 1)
```

(1), (2) are in correspondence with figure 10

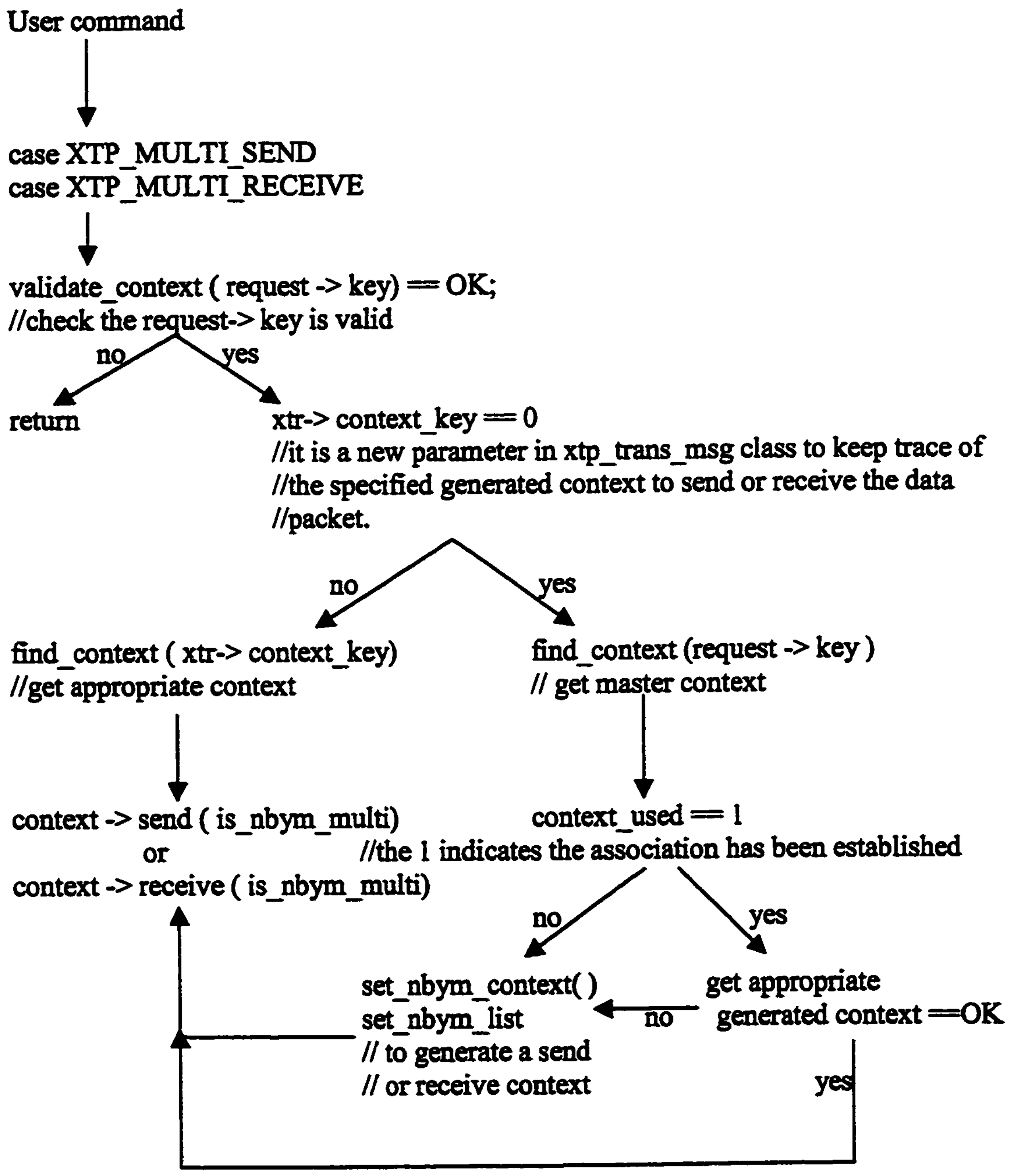
Figure 13. The changes in the handle_new_packet function

3. The process_packet function indicates which process is used for the received packet according to the context state and the packet type. If it is another FIRST packet and the context is in the Active state, then the Sandia XTP of the

process_packet function performs it as a duplicate case. It must therefore be modified here for the m by n multicast, since each transmitter has to send the FIRST packet to make a path between the transmitter and the receiver.

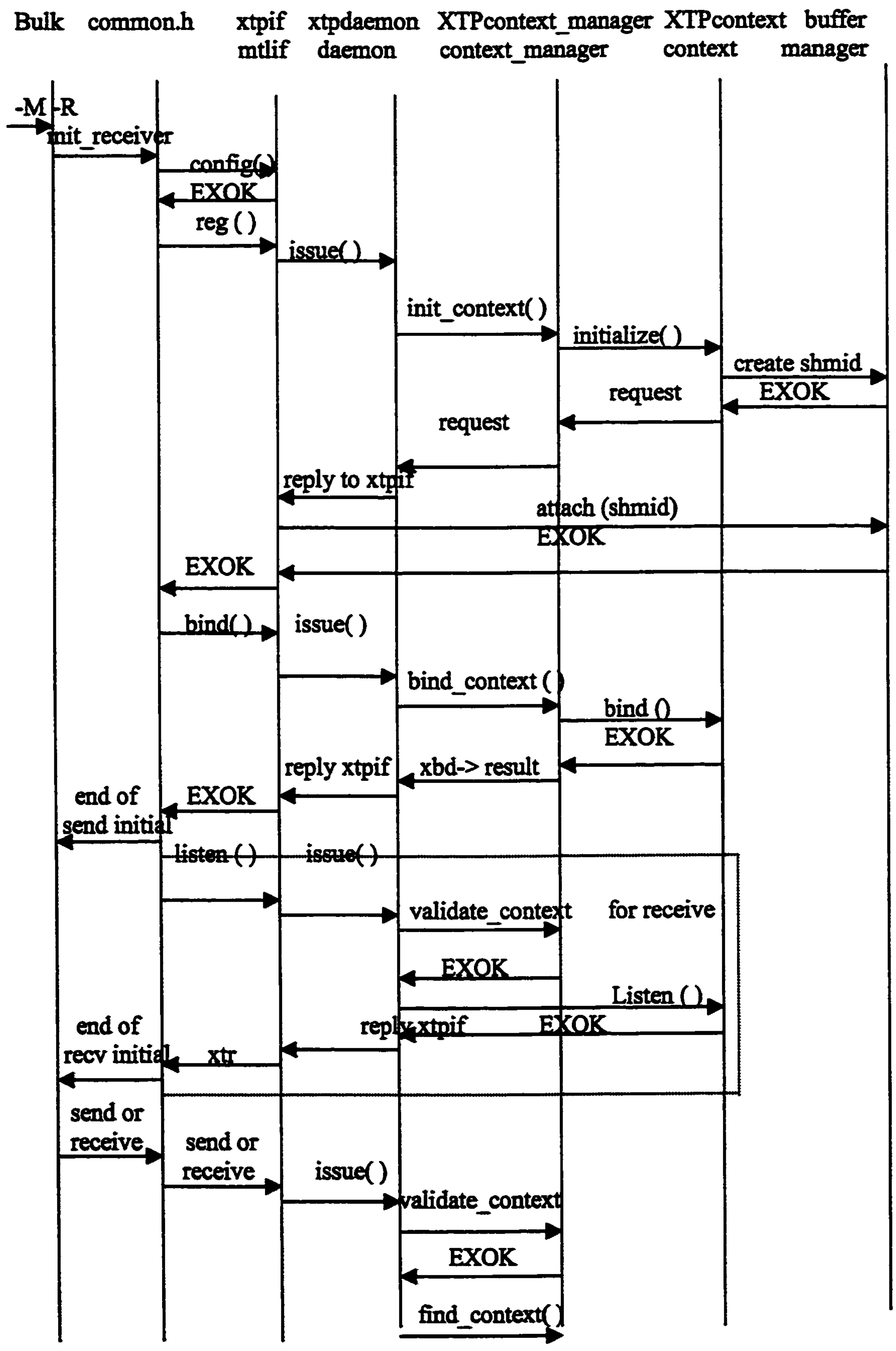
4. The Sandia XTP user interface is a bridge between the user and the XTP daemon to pass user commands to the daemon and the send back received messages from the daemon to the user. To perform the m by n multicast, it is necessary to distinguish the m by n multicast from the one by n in both the XTP user interface and the XTP daemon classes. In order to specify that the process used is the m by n multicast, two commands (XTP_MULTI_SEND and XTP_MULTI_RECEIVE) were added so that the user can tell the daemon to use the m by n procedure. After the daemon has been given a correct command, the daemon engages the right process. Figure 14 shows the algorithm for the m by n case in the daemon.
5. The retransmission function is used in the handle_WTIMER function. It is caused by timeout, and there are no responses from destination hosts, and its context states. To adapt to m by n multicast, it is necessary to avoid from retransmission if the transmitter context's state is Active and all the data are received in the receiver side. This will cause retransmission in the Sandian XTP 1 by n and unicast. To perform this, it has to issue a stop_WTIMER function once the transmitter has received all of the responses.
6. The m by n multicast send and receive procedures have their own API functions to enable the user to access the XTP. There are two new functions comprising send and receive to be added in the common.h, xtpif class and the context class. Other functions include modified inside codes of Sandia XTP implementation to

perform the m by n multicast case. Figure 15 shows an XTP m by n multicast send and receive events trace.



User command
case XTP_MULTI_SEND
case XTP_MULTI_RECEIVE
validate_context (request -> key) == OK;
//check the request-> key is valid
no
yes
return
xtr-> context_key == 0
//it is a new parameter in xtp_trans_msg class to keep trace of
//the specified generated context to send or receive the data
//packet.
no
yes
find_context (xtr-> context_key)
//get appropriate context
find_context (request -> key)
// get master context
context -> send (is_nbym_multi)
or
context -> receive (is_nbym_multi)
context_used == 1
//the 1 indicates the association has been established
no
yes
set_nbym_context( )
set_nbym_list
// to generate a send
// or receive context
get appropriate
generated context ==OK
no
yes

Figure 14. The new algorithm in the daemon dispatch_request function



Bulk
common.h
xtpif mtlif
xtpdaemon daemon
XTPcontext_manager context_manager
XTPcontext context
buffer manager
-M -R
init_receiver
config( )
EXOK
reg ( )
issue( )
init_context( )
initialize( )
create shmid
EXOK
request
request
reply to xtpif
attach (shmid)
EXOK
EXOK
bind( )
issue( )
bind_context ( )
bind ()
EXOK
reply xtpif
xbd-> result
end of send initial
EXOK
listen ( )
issue( )
validate_context
for receive
EXOK
Listen ( )
reply xtpif
EXOK
end of recv initial
xtr
send or receive
send or receive
issue( )
validate_context
EXOK
find_context( )

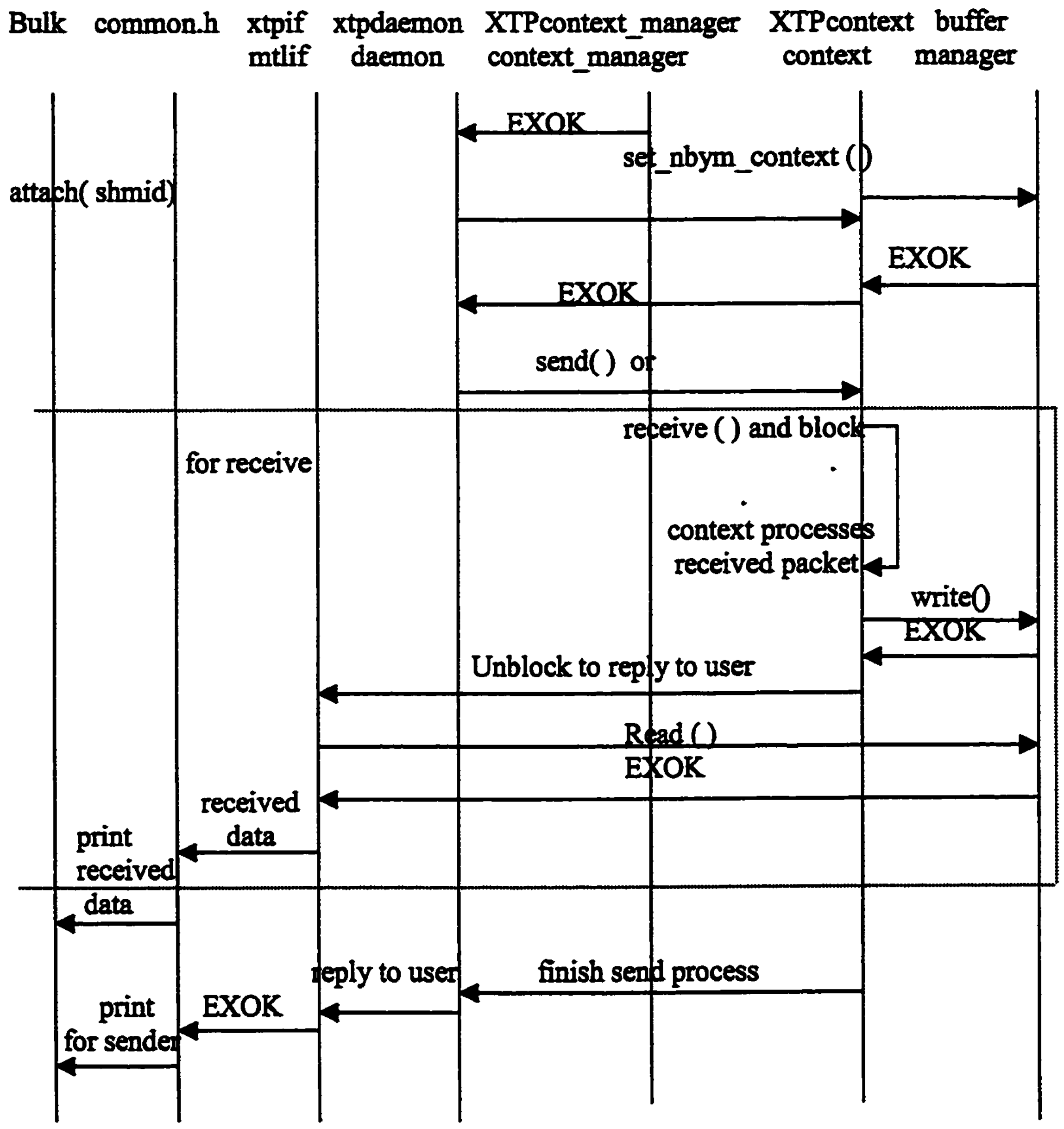


Figure 15. M by N multicast event trace in send and receive procedure

4.5 Association Establishment in the M by N Multicast

When the user sets an output command to send the FIRST packet with a multicast address, if certain contexts are listening on the same multicast address from the other hosts, the FIRST packet will be given all the appropriate Listening contexts. Once the contexts have received the FIRST packet, their state changes from listening to Active simultaneously, while the receiver hosts send back JCNTL packets as a join acknowledgment. The first host to issue the FIRST packet will be the master host and the rest of the hosts will be slaves. Once the group has been established, we can assume that the group will be kept until everyone has finished sending information. With the m by n multicast, the slave host turns from receiver to sender, and it still works in the same group, while the context is still in the Active state and there is no later join case in the m_by n case which means that the later participant issues a JCNTL packet with a key value equal to 0. This is the difference between the one by n and m by n multicast. In the one by n multicast, after the master transmitter has finished sending data, the group is closed and the later joining is permitted in the one by n multicast through issuing a JCNTL packet from the joining host.

4.6 Association Management in the M by N Multicast

The establishment procedures occur when the context receives the FIRST packet. These procedures put the facility into place using the FIRST packet delivery address. After the context has received the FIRST packet, it sets the destination address through the transmitter address for establishing a path between the sender and the receiver. For example, the JCNTL packet is emitted with a unicast address from the receiver to the sender. With the m by n multicast, each transmitter needs to send the FIRST packet before the transmission data, but it does not need to build an association many times, since the association is established just once at the beginning.

Sending the FIRST packet from slaves is necessary to establish a path between the transmitter and the receiver.

4.7 To Terminate the Association in the M by N Multicast

Associations are designed to handle an arbitrary number of messages over an arbitrary length of time, but sometimes it is necessary to end the association in specific cases, as when all of the transmissions in the association have been finished. The XTP provides the mechanisms for several different termination semantics, ranging from an independent graceful close to an immediate abort [2].

With the m by n multicast, the graceful close (three-way handshake) is necessary to end the association when all the transmitters have finished transmitting their data. The termination association with the m by n multicast complies with the following policy.

- The association master identifies the process at a given host that first transmits the XTP FIRST packets for a given m by n group. It will not set the RCLOSE bit in the packet header and its contexts. While the receivers receive the packet without an RCLOSE bit, the receivers will leave the context open as regards the incoming data stream and the outgoing data stream.
- When the master host sends the last packet with the WCLOSE bit to the slave hosts, the slave hosts will record WCLOSE then reply to the master host with RCLOSE. This implies the receivers will not receive the transmitter packet because the transmitter has set the WCLOSE in order to close the outgoing data stream. The following three figures a, b and c, describe how WCLOSE bit and RCLOSE bit can be set using four hosts multicast data communication.

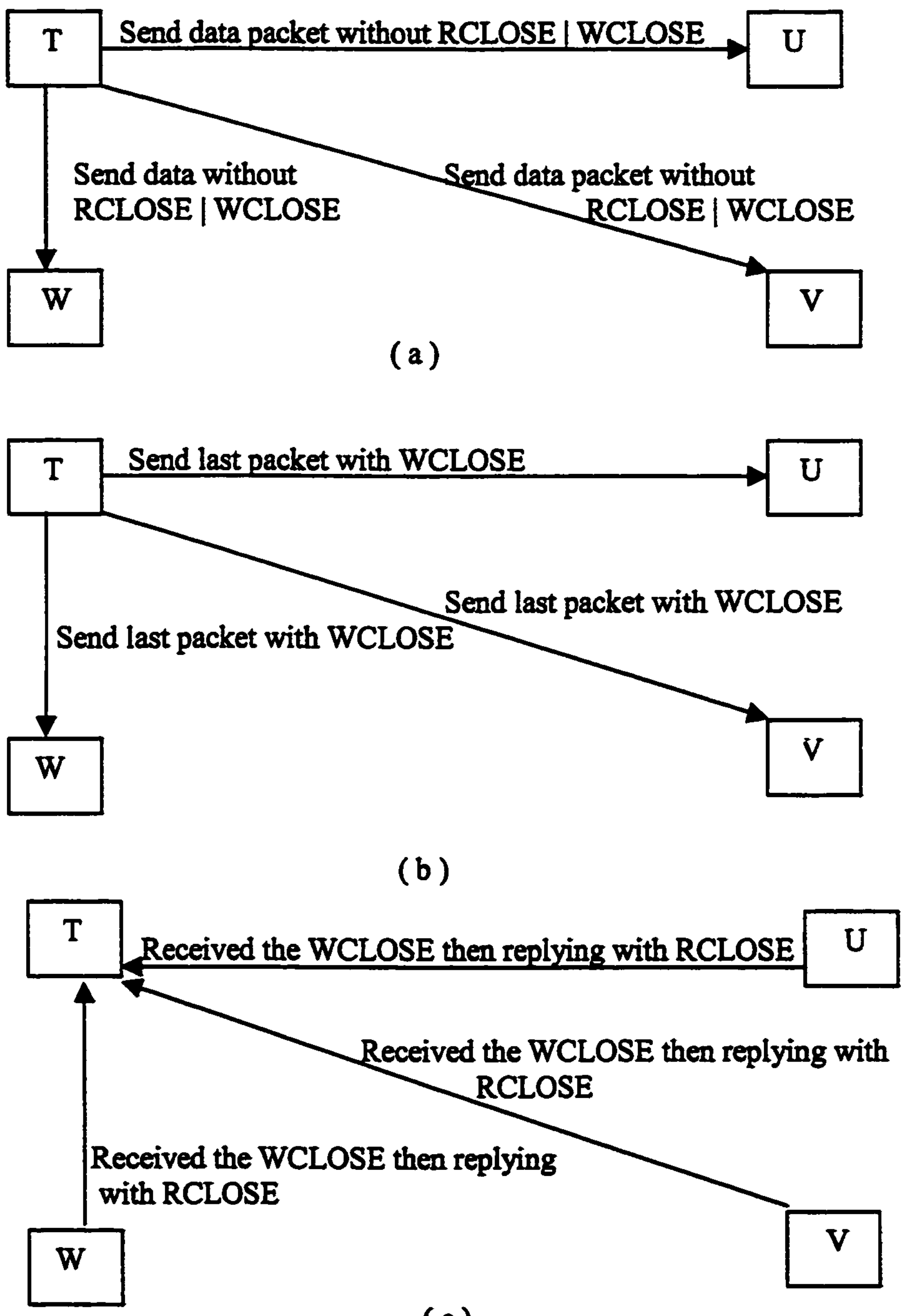

T
U
W
V
Send data packet without RCLOSE | WCLOSE
Send data without RCLOSE | WCLOSE
Send data packet without RCLOSE | WCLOSE
(a)
T
U
W
V
Send last packet with WCLOSE
Send last packet with WCLOSE
Send last packet with WCLOSE
(b)
T
U
W
V
Received the WCLOSE then replying with RCLOSE
Received the WCLOSE then replying with RCLOSE
Received the WCLOSE then replying with RCLOSE
(c)

Figure 16. The architecture for sending RCLOSE and WCLOSE

- Any sender in the association once it has finished issuing WCLOSE looks for RCLOSE in all the other sites. After the association master has received each

WCLOSE from all of the members in the association, it will then send the WCLOSE | RCLOSE | END bits with a multicast address to terminate the established association.

With an additional termination association, the master host has the power to force the association to be terminated. Once the master host has sent the END bit to the members of the association at any given time, then the association will be shut down immediately. However as regards the slave hosts of the association, once any slave host wants to leave the association, it sends the END bit to the members of the association and then it terminates the connection with the association. This implies that these hosts have retired from this association.

5. Top Level (ECSL) in M by N Multicast in XTP

It frequently happens in M by N multicast communication that strict ordering is required among the packets from the multiple senders. Ramasivan [6] proposes the addition of a layer above the M by N XTP features, to manage this ordered delivery. The ordering is achieved by requesting a token from the association master, and combining this token with all data sent by a particular host.

This Enhanced Communications Support Layer (ECSL) is outlined in Ramasivan's thesis [6]. Actual implementation of this proposal requires additional changes to the Sandian XTP. Additional design ideas are contained in this chapter, based on experience with the implementation of M by N multicast. However, the actual implementation is left for future work.

5.1 The Principles of ECSL in XTP

The International Organization for standardization (ISO) and the International Electromechanical Commission (IEC) have defined a new transport service interface named Enhanced Communications Transport Service (ECTS). This draft standard attempts to provide a uniform and universal service interface between transport protocols and applications that require support for powerful multimedia group communication [5]. The Enhanced Communications Support Layer (ECSL) constitutes a new proposal for the XTP multicast. It is based on the ECTS principle to dealing with token management and data ordering. ECSL is a new proposal to perform multicast in XTP for today's high-speed networks.

The principle of ECSL in XTP is to manage multi-sender function in a communication group. Essentially it is m by n multicast that has ordering between each participant who wants to send the data packets. This implies that when the master transmitter has finished making the association, other transmitters can start to send as a multi-sender within one group. Once the receivers receive the packets, the XTP daemon will deal with the received data by means of the token list. This both enhances efficiency and saves time in group communication. ECSL is a natural extension of the m by n multicast, the context generation process, establishment association, path establishment, context states changing, and terminate, and can be operated according to the same processes as m by n multicast.

5.1.1 Token Message Table in ECSL

To perform ECSL for multicast in each host, the buffer has to be set in XTP daemon so as to store massive amounts of data, and the token table needs to be built within the daemon too. Table 3 below permits one example of the taken message table from Ramasivan's thesis that lists token numbers and corresponding packet numbers.

Table 3. Token Message Table

T–1	T-2	T-3	T-4	...	...	P-1	P-2	P-3	...				..	P-16
Token Number						Packet Number								

This token table is set in the packet header for data delivery. Only the master host can change the token value in the token table. We can assume the token apply procedure, once association has been established, the group members who want to send data have to send packet with a request message of token applying with a unicast address to the master host, then the master host assigns the token number in the token list according to its established priority, and sends it to group members with a multicast address immediately. Every participant in the group needs to obtain the same token list (this is different from Ramasivan's thesis for the address mapping as Ramasivan contends that the token list should be sent with a unicast address from the master host). After all the members have received the token message, they can put it in the daemon and start their transmission procedure. In the case of ECSL, the user can not directly affect the daemon if the user wants to send data. The user consequently has to send data to the daemon buffer, and then the daemon will order the data and provide an appropriate context to send it. Once the daemon arranges the data for the user, it will identify the token number corresponding to the packet number so as to write the data to input queues.

5.1.2 The ECSL protocol for session layer connection

Table 4. The New Protocol for ECSL

Protocol Version	Packet Type	Length
Session Source Connection Identifier		
Session Destination Connection Identifier		
Message Acceptance Record		
Data		

- Protocol version: 8 bits. This indicates the protocol version. Here its value is set to 0x01.
- Packet type: 12 bits. This indicates the desired kind of packet for the ECSL protocol, indicated in table 7 below.

Table 5. Token Type List

Service	Parameters
Open	S_OPEN_MASTER S_OPEN_SLAVE
Token Request Token Response Token Deny Token Cancel	S_TOK_REQ S_TOK_ACC S_TOK_DEN S_TOK_CAN
Data	S_DATA

- Length: 12 bits. This indicates the length of the data field.
- Session Source Connection Identifier: 32 bits. This indicates source entity.
- Session Destination Connection Identifier: 32 bits. This session layer identifies the destination entity.
- Message Acceptance Record: 32 bits, comprising 16 bits for the token number and 16 bits for the packet sequence number.
- Data: The packet can supply its own data.

5.2 The Performance of ECSL in XTP

There are two ways to perform ECSL protocol for session layer communication, one is to set up this protocol in each existing XTP packet, and another is to create a new packet for ECSL. The best way is to create new packets for session layer communication. To perform ECSL in XTP, Ramasivan's design should be changed in terms of sending or receiving procedures. Ramasivan's design enables the user to send or receive commands to or from the daemon so that the daemon can identify the context to perform sending or receiving process immediately. In ECSL performance, the daemon would not directly reflect a user command, since it is primarily a send command. The daemon has to wait until all the highest priority's data have arrived by receiving the EOM from the data source host, then the daemon can request a context for transmission. Because user commands do not directly affect the context, the context does not unblock to reply to the user, nor does it write received data into the receive buffer for the user either. There are two ways to transfer data from the daemon to the user. In the first instance, the context writes the data to the daemon buffer, and then the daemon writes the data to receive the buffer according to the daemon token list. The other way is to let the context temporarily store the data until it gets the daemon's order to write the data to receive the buffer.

5.3 The Design of ECSL for the XTP Multicast

To set ECSL in XTP provides an acceptable solution to the problem of raising the efficiency of network transmission for the m by n multicast. Since it is built in the session layer, it is designated the top level of XTP. The primary character of XTP is peer to peer transport layer data transmission. The algorithm is determined after the group members get the FIRST packet through which to make association, and the token applying will be sent to the master host with either the JCNTL packet or a new packet. The master host sets the token number for each of the slave hosts and sends back this token list with either the JCNTL packet or a new packet with a multicast address. After every slave host has received the token list; they start to send data. The six principal steps outlined below should be modified and extended to implement the ECSL in XTP multicast.

1. To set a token table in daemon that will include the parameters of the token number, the packet number and the key value of the data source host.
2. To create a buffer in the daemon which will hold at least twenty data packets.
3. To increase the packet header's length by adding a 32 bits token table to it. Before the packet with data has been emitted, the context will write the token number and packet sequence number in the packet token table.
4. If the user is blocked by the daemon from receiving data, the context should not unblock the user since the receive context just writes the received data into the daemon receive buffer. The daemon will take over this task of unblocking the user and ordering the received data to be sent to the user.
5 In slave hosts, the first process with a receive command is to receive the FIRST packet to join association. If the user wants to send data, the user should notify the daemon by applying for the token after association. Certain parameters should be added in the user request class to facilitate communication between the daemon and the user, since the daemon should know after establishing association whether the host wants to apply for tokens for sending data or not.
6. The token management in ECSL with m by n multicast. When slaves want to send data they have to ask for a token by sending a packet to the master. This could be handled by a JCNTL packet or created by a new packet with the unicast address as its destination address. The master host will then position the

applicants on the token list table according to their priority, and subsequently reply using the token for the user with a multicast address. It is better to use a multicast address, since the token lists are the same for each host. Every user must get token lists from the master host, so that they can send packets with data or wait for the processing of the packets if they don't want to send any data in the association.

6. Experience Test and Result

The xtpif class implements the programming interface for the Sandia XTP. The xtpif object is the user access point ensuring the functionality of XTP. The user can only use the functions provided by xtpif to accomplish the unicast or multicast.

xtpif() is a constructor with no argument and ~xtpif() is a destructor for releasing the user's resources kept for the association.

config() allows the user to configure certain parameters for this association.

reg() registers the user with the XTP daemon.

bind() binds an address to this association.

getaddr(), gettspec(), puttspec(), and getstate() are used to extract and replace information about the association held within the context.

listen() causes the endpoint to listen for incoming FIRST packets.

receive() and send() are overloaded to allow the user to receive the data into the receive buffer without reading the receive buffer, and they also permit the user to chose m by n multicast in one of the overload functions.

release() causes the XTP daemon to release all resources for this association.

sendcntl() causes an unsolicited control packet to be sent.

The implementation of the XTP m by n multicast has to be run under UNIX environment and has been integrated with a new timer system implemented by Yonglin Jiang. The new application program to run the XTP m by n multicast is called bulk4 and it is found under //mnt/grad/ma_tong/mysniff/team_work/bin. The application program is better to run in pine, loon, forest and fir four machines in Concordia university, since those four machines have more than ten shared memories which allows the context to be attached if you want to run all four machines. For those who don't have the right to access these machines, I also provide another application program which is called mnbulk permitting three machines to perform multicast so that you can run it in any three machines in the Computer Science Department of Concordia University.

In the bulk4 application program, type bulk4 –K –R 239.0.0.239 can be performed in one machine, and it performs receive, receive, send and receive in the group. Type bulk4 –Q –R 239.0.0.239 in one machine, performs receive, send, receive and receive in the group. Type bulk4 –M –R 239.0.0.239 in one machine, it does receive, receive, receive and send in the group, on the last machine, type bulk4 –H –T 239.0.0.239, performs send, receive, receive and receive in the group.

It is possible to run mnbulk application among three machines. In one machine, type mnbulk –Q –R 239.0.0.239, it performs receive, send and receive in the group. In one machine, type mnbulk –K –R 239.0.0.239, performs receive, receive and send in the group. In last machine, type mnbulk –H –T 239.0.0.239, performs send, receive and receive in the group.

From the above test, it is evident that each host can be a sender and receiver in the group. The xtpif class can be used to create more m by n multicast application programs which can be variously applied.

7. Conclusion and Future Work

A set of XTP requirements, including design and implementation, has been given for a generally applicable multicast transport platform. This report provides a general overview of XTP protocol. More specifically, the m by n multicast has been described in this report according to its implementation for the Sandia XTP. The more detailed design of the top level of m by n multicast—ECSL, has also been presented in this report. Certain strengths and limitations can be identified and will become more clearly manifest once the suggestions for future work outlined in this report have been implemented.

XTP owes a great deal to outstanding academic work conducted both before and after its inception. The XTP owes everything to this seminal work which will undoubtedly became stronger and more reliable in the future as it continues to evolve in the wake of this research.

7.1 Addressing the problem where the master sender is dead in multicast

If, in m by n multicast, after the association has been established, all the slave hosts send data packets one by one to the existing association which is created by the master host, then the association will be closed by the master host. During transmission, if the master host found to be dead, a problem emerges concerning who will send the packet to the slave hosts to turn off the association. There are at least 2 ways to handle this case. One solution is that each host can wait for the closing of the packet from the master host after a certain period of time, if this is not possible, the association will be terminated automatically. Another solution is that after a specified period of time, if the slave has not received the closing signal from the master host, then it will send a packet to the master host to query after it. If there is no immediate reply, then he will automatically leave the association.

If the master host is found to be dead after he has finishing sending the FIRST packet, the consensus of a new master host may be required to manage the association.

Further consideration of the problem of the dead master host will be necessary in any subsequent research.

Closing the association is not possible in one group, since the death of the master host affects all the groups.

7.2 Later joining in m by n multicast

Later joining frequently occurs in multicast. It sometimes happens that after the association has been created, another host asks to join the group as well by sending a JCNTL packet. If it gets a reply from the master host, it can join the group by sending and receiving packets from others. XTP one by n provides an opportunity for a host that wants to join later, however this option is not available in m by n multicast. In this scenario, if the host is permitted to join the group, it will begin to receive a packet. The problem concerns how the host can obtain the token to start sending a data packet. In ECSL, all the token lists are sent from the master host at the very beginning, and so the host that attempts to join later cannot send a data packet, because in each host's daemon, there is no context token number. Perhaps it is possible to choose another algorithm and send a new token number from the master host to the slave hosts so as to alert other hosts that there is a new member in the group. This design problem will undoubtedly be of interest in any future research.

8. Reference

1. Jim Krupp, ed., *Xpress Transport Protocol Specification XTP Revision 4.0B XTP*, XTP Forum, July, 1998.
2. W. Timothy Strayer, Bert J. Dempsey, Alfred C. Weaver, *The Xpress Transfer Protocol* 1992.
3. Andrew S. Tanenbaum, *Computer Networks*, Third Edition, 1996.
4. J. W. Atwood, *The Xpress Transport Protocol: Outline of a Tutorial.*
5. Douglas E. Comer, *Internetworking with TCP/IP Volume I Principles, Protocols, and Architecture*, Third Edition, 1995.
6. Ganesh Ramasivan, *Enhanced Communication Services for Many-To-Many Multicasting Using XTP*, Thesis in Department of Computer Science at Concordia University, March 2000.
7. Louis Harvey, *In Search of a Rate Control Policy for XTP: Unicast & Multicast*, Report in Department of Computer Science at Concordia University, March 1999.
8. Kamal Ghander, *Requirement Documentation for XTP Time Management Base Class*, Report for the fulfillment of a Computer Science Project at Concordia University, January 10, 2000.
9. Infrastructure and Networking Research Sandia National Laboratories, *Meta-Transport Library Reference Manual*, MTL version 1.5, December 1996.
10. Infrastructure and Networking Research Sandia National Laboratories, *SandiaXTP User's Guide*, Release 1.5.1, 1997.
11. Infrastructure and Networking Research Sandia National Laboratories, *Meta-Transport Library User's Guide, Release 1.5.1*, 1997.

12. Infrastructure and Networking Research Sandia National Laboratories, SandiaXTP Reference Manual, SandiaXTP version 1.4, February 1996.

13. S. Armstrong Xerox, A. Freier Apple, K. Marzullo Cornell, *Multicast Transport Protocol*, February 1992.

14. C. Bormann, J. Ott, H.-C.Gehrche, T.Kerschat, N.Seifert, MTP-2: Towards Achieving the S.E.R.O. Properties for Multicast Transport, Presented at the ICCCN'94, San Francisco, September 1994.

15. Carsten Bormann, Joerg Ott, Nils Seifert, MTP/SO: Self-Organizing Multicast draft-bormann-mtp-so-01.txt, Internet-draft, May 1998.