ESTELLE SPECIFICATION OF XTP: ANALYSIS OF DATA PROPAGATION WITHIN XTP

Luc Lajoie

A report
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For The Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

APRIL 2001

0-612-59331-2

Abstract

ESTELLE Specification of XTP: Analysis of Data Propagation within XTP

Luc Lajoie

The first version of the XTP protocol using the ESTELLE specification language was of little or no practical use. The goal of the exercise was to produce a specification of the XTP protocol, using a tool that was designed to build telecommunication protocols and benefit from language constructs designed to help the coding and thereafter the maintenance of the protocol. This benefit was clouded by poor performance. First analysis revealed that the data was manipulated character by character and copied several times throughout its existence within XTP.

At the same time, a C++ version of the XTP protocol was available, with acceptable performance. The C++ port of the protocol was tightly coded, using most of the C++ language features making the understanding of the association of code parts to protocol components a tedious task to master and an even greater challenge to maintain and improve to keep up with the protocol evolution.

Originated and motivated from an observation from Dr. William Atwood from Concordia University Canada, it was proposed that the ESTELLE code be improved by making use of a slight deviation from ESTELLE into the C language by making use of 'Qualifying Comments' in ESTELLE. Such comments allowed the specification of C constructs, namely pointer references, that are not a defined type for ESTELLE.

Further enhancements gave birth to a generation of the ESTELLE specification, where most of the data manipulation C operations had been taken out of the XTP protocol, and consolidated into a set of highly efficient and frequently executed parts of code. These 'code beans' are referred to as 'primitives' and make full usage of the C language. The thrust of this report is to review and document the topology of the most recent ESTELLE specification of the XTP protocol, propose improvements if such are possible, and finally compare the performance of both the ESTELLE specification and the SANDIA C++ specification of XTP.

Acknowledgements

I first want to express my most sincere thank you to Dr. J.W. Atwood for his infinite patience in bearing with me in my analysis of the ESTELLE specification of XTP, his infinite resourcefulness in providing the texts of the different versions of the ESTELLE specifications, coordinating the group discussion sessions, bringing together very talented students, making it possible to understand clearly all of the different specifications of XTP, regularly holding tutorial sessions by the protocol designers and giving the opportunity to each and everyone of us to share with one another recent progress in the part of the XTP protocol description that we are assigned.

Most importantly, I wish to acknowledge the infinite resourcefulness that we, all of his students, were continuously benefiting in times where progress seemed less likely to happen, Dr. Atwood's advice was truly inspiring, time after times. Finally, the uniqueness of his vision on how to improve the performance of crucial part of the ESTELLE specification was taken as a defacto directive, and became the thrust for the next implementation of the specification.

I have sometime heard the quotation: "A team will consistently achieve more than a group of individuals". Throughout the several years that I had the pleasure of working for Dr. Atwood, there was always the presence of a team of collaborating students. A team driven by a unique vision and a wealth of invaluable information that made each of us believe that our contribution as a team member was en-

hanced to a level that spirited our confidence and drove our productivity.

Contents

1 Initial Description and Thrust of this Work 1

1.1 Transmission Protocols: The Basics 1

1.1.1 Formal Description of XTP 1

1.1.2 A Communication Model 3

1.1.3 Notion of UNICAST Vs MULTICAST. 5

1.1.4 Flow Control 6

1.1.5 Rate Control 7

1.1.6 Error Control 8

1.2 SANDIA XTP Vs ESTELLE specification 9

1.2.1 Motivation for the SANDIA XTP 9

1.2.2 Motivation for the ESTELLE Specification of XTP 10

1.3 History and Improvement of ESTELLE XTP 11

1.3.1 Earlier versions of ESTELLE XTP 11

1.3.2 The Ring Buffer 13

1.3.3 Referencing the Data. 14

2 ESTELLE XTP Components Description 18

2.1 Components and Modules Overview 18

2.2 The User Interface 21

2.2.1 The User Interface Primitives 24

2.3 The Context Manager 26

2.3.1 The context send/receive buffers: the ring buffers 28

2.3.2 The Data Primitives 31

2.3.2.1 Concept 31

2.3.2.2 Description of the Data Primitives 31

2.3.3 ESTELLE XTP Calls to the Data Primitives 38

2.4 The Network Interface 54

3 The Channel Activity Trace 59

4 Data Transmission Performance Benchmark 63

4.1 Description and Credits of the Performance Data 63

4.2 Data Performance Result Diagrams 64

4.3 Data Performance Results Conclusion 66

A. Appendix 68

A.1 Glossary 68

A.2 References 69

List of Figures

Figure 1. General Network Communication Model 3
Figure 2. Typical Context Association Data Transfer Behavior 4
Figure 3. Kernel Implementation of a Transport Protocol 16
Figure 4. User Space Implementation of a Transport Protocol........... 17
Figure 5. The Processes and Code Layers of Estelle XTP 18
Figure 6. The USER, XTP and Network Interface Pictorial Design..... 20
Figure 7. User Process Unix Shared Memory XTP and Data Del. Services.. 27
Figure 8. The Context Ring Buffers 28
Figure 9: Modular representation of the ESTELLE XTP Specification 40

List of Tables

Table 1: ESTELLE Data Throughput Benchmarks 65

1 Initial Description and Thrust of this Work

1.1 Transmission Protocols: The Basics

This section is included to ensure the knowledge of the traditional functions and features of transmission protocols in general. Specifics are added to introduce the role and differences that XTP brings in contribution to the performance, reliability and innovation in this domain of the research. The material related in this section is inspired from a work in progress from a colleague student.

The following subjects are elaborated with goal of describing XTP while setting the stage for the description of the generic features of a transmission protocol.

- Formal description of XTP
- A communication model
- Unicast vs. Multicast
- Flow Control
- Rate Control
- Error Control

1.1.1 Formal Description of XTP

Compliant with the OSI-7 layer model, XTP addresses and contributes in the following fields as a transport layer protocol:

- Orthogonal protocol functions for separating paradigms from policies
- Separation of rate and flow control

- Explicit first-class support for reliable multicast
- Data delivery service independence

Other features of XTP, completing its formal description but not necessary unique to this protocol are: implicit fast connection setup for virtual circuit paradigm, key based addressing lookup, message priority and scheduling, support for encapsulation and convergence protocols, selective retransmission and acknowledgment, fixed size 64-bit frame design, 64-bit sequence and connection identifiers, parameterized traffic and finally quality of service.

1.1.2 A Communication Model

Next, a general communication model is included (see Figure 1). It is adapted from the SANDIA XTP User's Guide Release 1.5 and depicts a Unicast communication model. It attempts to locate the main communication components and place labels. This diagram is not a complete communication model, as such is included later.

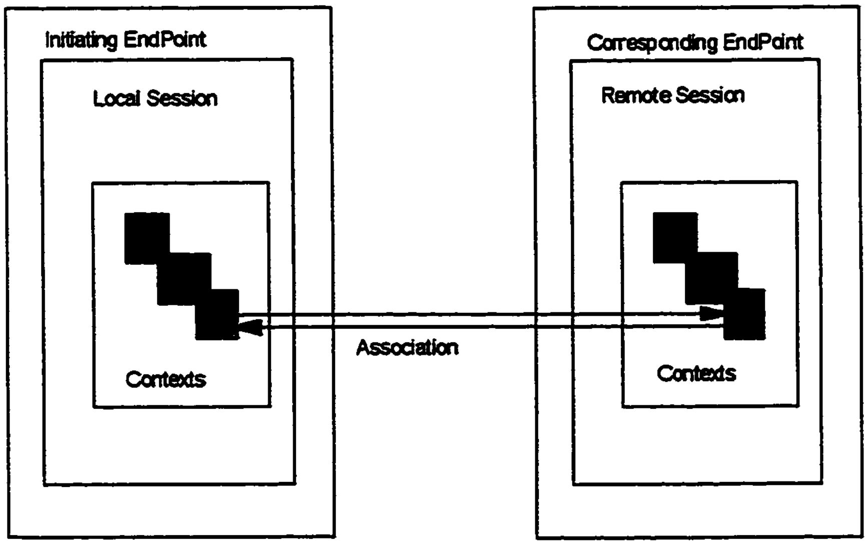


Figure 1. General Network Communication Model

Figure 2 below, is included to show additional information on how a context association is initiated, functions and ends. Again, it is general in description and pertains to the general network architecture of XTP.

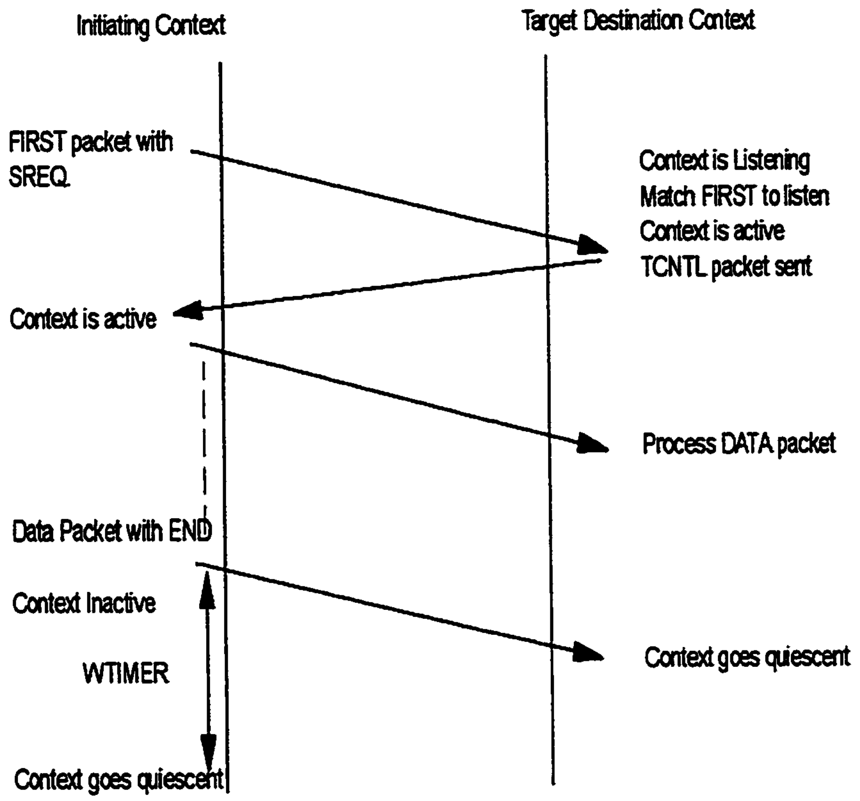


Figure 2. Typical Context Association Data Transfer Behavior

Referring to the association diagram above, we can find that:

The context is always in one of the following states: Quiescent, Listening, Active or Inactive. Based on a user request, the contexts change their states. Next, we can make a difference between a Unicast communication model, where one sender's context communicates with one and only one receiving context for a given session, and a Multicast session model, which engages a sending context in communication with several receivers. In a Unicast model, as depicted above in the diagram, the following state transitions take place:

- The receiver application always starts first, and the receiver goes to the LISTENING state, a transition from a QUIESCENT state, and waits for the sender's packet.
- The sender issues an output command (FIRST Packet), and the sender goes to the FIRST_SENT state.
- After the FIRST packet is accepted by the receiver, it enters the ACTIVE state.
- A TCNTL packet is returned by the receiver to the sender, which causes the sender to move to the ACTIVE state.
- At that time, the session is established and both the sender and the receiver can start exchanging well defined packets, freely.
- When the receiver receives a packet with the END flag set, the receiver reverts to the QUIESCENT state.
- The sender then enters the INACTIVE state, and finally goes to the QUIESCENT state.

The association is terminated.

1.1.3 Notion of UNICAST Vs MULTICAST.

Unicast support, which XTP provides, is the support of a communication session between one sender endpoint and one receiver endpoint. The XTP

Unicast provides a high degree of functionality, through orthogonal protocol mechanisms. These mechanisms are in the form of fields and bit flags, used during packet exchange, over the lifetime of an association. The association management procedures define how the fields and bit flags are used during the lifecycle of the association.

Multicast is a major distinctive feature of XTP. XTP implements Multicast while ensuring that there is no duplicate data from a single context, to a set of XTP receivers, hence saving bandwidth and increasing throughput performance. Similarly to Unicast, the Multicast feature of XTP provides a powerful mechanism for group communication, supporting a data service of 'one to many' and 'many to many' associations. The XTP Multicast uses the same control algorithms and mechanism in flow control and rate control, defined below, except for the association management.

1.1.4 Flow Control

Flow control, as implemented by XTP, has the following goal: Like TCP, XTP Flow Control aims to prevent swamping of a slow receiver, with too much data, too quickly, by transferring control to the receiver, who then issues credits to the sender. Specifically to XTP, the following applies to Flow Control:

- XTP reserves two fields, *alloc* and *rseq* to negotiate the flow control window value.

- XTP also implements two additional control variables, the *RES* bit and the *NOFLOW* bit, providing the user with the following configuration options:
  - By setting the RES bit, the sender instructs the receiver to advertise only the actual buffer allocated by the user for the context. This is called the 'reservation mode'. In this mode, the receiver is forced to adopt a conservative policy, making sure that no packets are lost due to a lack of buffer space.
  - The NOFLOW bit indicates to the receiver that the transmitter does not wish to adhere to flow control constraints, hence flow control in the forward direction will be disabled.

This short introduction to flow control gave insights to what is flow control and how it applies to XTP.

1.1.5 Rate Control

Unlike flow control, rate control focuses on the producer/consumer relationship, as it applies to XTP between the endpoints, and considers processor speed and congestion. XTP performs rate control by monitoring the following factors: Rate, Credit and Burst, and additionally with a refresh timer called RTIMER. The relationship among these factors is as follows:

- RATE is the user expected data output rate. By default, rate control is disabled.
- BURST is another user-input parameter. It is the maximum number of data bytes that can be sent at once.

- RTIMER is equal to the value of BURST divided by the value of RATE. It is a period timer.

- CREDIT is evaluated (verified) each time the context wants to send data out. The credit is decreased until it reaches the value of zero, while data is sent. If there is no more credit available, the data packet is put on a FIFO packet queue and no more data is sent. Basically, the context will be assigned the credit value (equal to burst) at each RTIMER time period.

1.1.6 Error Control

Error control is the operation of maintaining a checksum field in each packet. XTP implements a 16 bit one's complement sum over all octet pairs. XTP provides the choice of performing a whole packet checksum with the NOCHECK bit setting turned off, or to perform only a packet header checksum when the NOCHECK bit is turned on.

If a check error is found, the packet is simply dropped. A receiver will detect the lost packet by checking the incoming packet stream for gap in the sequence space. At that time, an ECNTL packet is sent back to the sender to request a retransmission. If the sender fails to receive the retransmission request, the sender will request a synchronization handshake. The two timer values WTIMER and CTIMEOUT are used to help the sender in detecting if the association can be recovered to normal. If the handshake fails, the association is terminated. The communication between the pair of receiver and sender is aborted.

1.2 SANDIA XTP Vs ESTELLE specification

1.2.1 Motivation for the SANDIA XTP

The origins of XTP go back to the drive of putting together a "PROTOCOL ENGINE" on a chip, that would amalgamate the network and transport layer into a transfer layer. Although such a hardware implementation is capable of very high performance, a need was also seen for a software implementation that was portable across a wide range of Operating Systems.

SANDIA XTP was designed with exactly this goal. Its C++ code base was chosen to enhance its portability across platforms and to provide a high performance protocol with the above goal of real-time packet forwarding using selective re-transmission as required instead of a go-back-n simple algorithm.

The down-side of this approach stems from the choice made for its performance and portability: its language that allows the developers to relax the representation of the protocol structure through its coding and thus making the protocol modules and inter module information communication a more tedious task to improve on and enhance the protocol with new features.

There are a number of copy operations that are imbedded into the C++ code representation. Each data copy operation comes from forwarding data from one component of the protocol to another. The thrust of this work will not focus on the SANDIA version of XTP but rather the ESTELLE described in the next section. An unpublished document is available that traces data movement within the SANDIA port of the XTP protocol. [Lajoie-2]

A better method of forwarding data within the XTP protocol is to pass a pointer to the data to the next component for reference. Thus a single (or minimal set of) repository is required to completely transfer data from the user interface right to the network interface medium.

1.2.2 Motivation for the ESTELLE Specification of XTP

The ESTELLE version of the protocol was designed with the goal of using a programming language that has network protocol constructs, thus expressing the protocol specifics (channels interaction points and modular component representation) and implementing the operation of the protocol through changes of an extended state machine. The underlying language of the ESTELLE higher level language is a C base.

The strength of this approach is the immediate comprehension of the representation, using the ESTELLE syntax, the separation of the lower level functions in the form of C primitives, but not at the price of coding protocol specific parts within the primitives. Rather these primitives are used to manipulate sub-components of information that ESTELLE, being a protocol representation language, cannot represent elegantly.

As for the SANDIA implementation of the XTP protocol, the aim and the thrust of this work are to identify, document and suggest improvements, as applicable, so that data forwarding within the XTP protocol is done with a minimal number of copy operations and data reference through pointer reference becomes the choice for data forwarding.

To see the scope of this work, and what has been done to this point, we first describe the interfaces through which data must be propagated. In

the next section, we can present some development history and progress suggesting how the protocol was improved to this point and what may be left to implement.
The major protocol components through which data must be forwarded are:

- The USER Interface
- The XTP Context Manager
- The Packet Management Modules
- The Network Interface

1.3 History and Improvement of ESTELLE XTP

1.3.1 Earlier versions of ESTELLE XTP

Driven by the motivation of expressing XTP in a protocol language, the protocol performance took a second plan. The ESTELLE compiler generates C code, but ESTELLE does not include the possibility of addressing data through reference, via pointers. Therefore data copy operations were very much in evidence across each module section of the protocol, and the result was a sluggish, and non-performance competitive implementation of XTP.

The concept of a "RING BUFFER" was used to hold data within the protocol, in the context module. Character by character, copy operations took place. In the following paragraphs, we relate changes according to the XTP version and tag the time period it applies to.

1995-1996: XTP 3.6 Corresponding ESTELLE Version

- Closer to the performance problem are the "in_ring_buff" and "out_ring_buff" procedures. These two procedures are the low-level character by character buffer update procedures.

- The two procedures are called by a higher level procedure, itself dealing with string messages. The procedure calls, for every character, consumes CPU cycles. The send and receive procedures were the higher level procedures.

- Making the two procedures C inline procedures, should improve the performance substantially, while maintaining the two discrete levels of abstraction.

The protocol, at this implementation level (V3.6) was not performing in a comparable magnitude to the SANDIA implementation, and its use was more a ESTELLE syntax protocol generation endeavor than a performance implementation.

A complete documentation of the Data Movements, Transitions and Generating Signals, Interaction Points and module structures and variables with goal of documenting the protocol while tracing the data copy operations is available through an unpublished work [Lajoie-1].

1996: XTP 4.0, ESTELLE Version (Prototype Version)

The major improvements are:

- Data transfer improvements
- Increased code reusability within modules.
- Introduction of multicasting and the use of new packet types.

Data transfer in this Prototype version was hindered by two major factors:

1. The use of a ring buffer for storing incoming messages or dequeing outgoing data
2. The data replication inherent to the Estelle code specification.

We examine each of these two points separately.

1.3.2 The Ring Buffer

The ring buffer as used in the specification is a temporary repository of the data, acting as a "buffer" between the user and the medium. Raw data coming from the user process is enqueued into the ring buffer before it can be sent into one, or more likely several, packets. Such transmitted data is found in the XTP First and Data packets. This is in the sender process.

Data coming from the medium as packets were also inserted into the ring buffer, to later be sent to the user process. This is in the receiver module.

Along with these ring buffer enqueue/dequeue activities, the concept of "spans" is used to keep tab of what fraction of the buffer has been transmitted or received as XTP packets. This technique implements the concept of selective retransmission.

The version 3.6 of XTP invoked the ring buffer enqueue/dequeue procedure for each character stored or extracted. Consequently, the protocol was of no practical use.

The version 4.0 of the Estelle specification of XTP made substantial improvements to the above findings. The ring buffer enqueue/dequeue procedures we called once for each transmission unit to be added or removed.

Still the ring buffer is a transitory stage and our goal here is to eliminate it, by making the data transfer more direct.

Typically in the Estelle specification, forwarding of information between modules leads to the copying of the information. It is noted, in earlier versions of the specification, that from one end of the process of forwarding a message, to the other end, the same message is copied several times, upward of five times. This copying of the message represents, more or less, the overhead of a ring buffer operation. The problem was still present. The following paragraphs suggest improvements that helped in eliminating the repetitive copying of data within the protocol.

1.3.3 Referencing the Data.

As explained above, the forwarding of information in Estelle, from one module to another, copied the data. What we need to do then is to send information about the whereabouts of the data, when invoking another module. The pointer data type, used in referencing the data, is defined using the "Estelle qualifying comment" statement. Using this technique, we can propagate the information, from say, the user module, to, finally the sender module, using only a few bytes of information to point to the user buffer area.

Finally, the user buffer is dropped into "XTP Packet Shells", in a piecemeal fashion, and a header is added to the packet(s) which are sent on the on the medium. The receiving of information is very similar using a reverse ordering of the above procedure.

The packet images are kept for as long it takes to get a confirmation of successful reception by the receiver, unless the "NOERR" option is used. In this case, the packets can be reused whenever they have been transmitted on the medium.

This improvement has been implemented in the following version of the ESTELLE specification, and we describe it next.

1997: XTP 4.0, ESTELLE Version (2.5.3)

This current version of the ESTELLE specification of XTP 4.0 added the following improvements:

- C Primitive (procedures) replacing the often-referenced ESTELLE ring buffer manipulation procedures.
- Support structures that implement pointer references in most cases.
- All of the Channel Interaction Points are passing data references, not the data itself.
- Somewhat implicit but still worth mentioning, the data is transferred by string manipulation and not on a character by character basis.

Before we start exploring the protocol components that characterize the structure of the Estelle specification of XTP, we first introduce a series of figures to give progressive insight at understanding the nature and role of the components found in the modular description of XTP. Figure 3 below shows a simple transport protocol design, imbedded in the operating system's kernel:

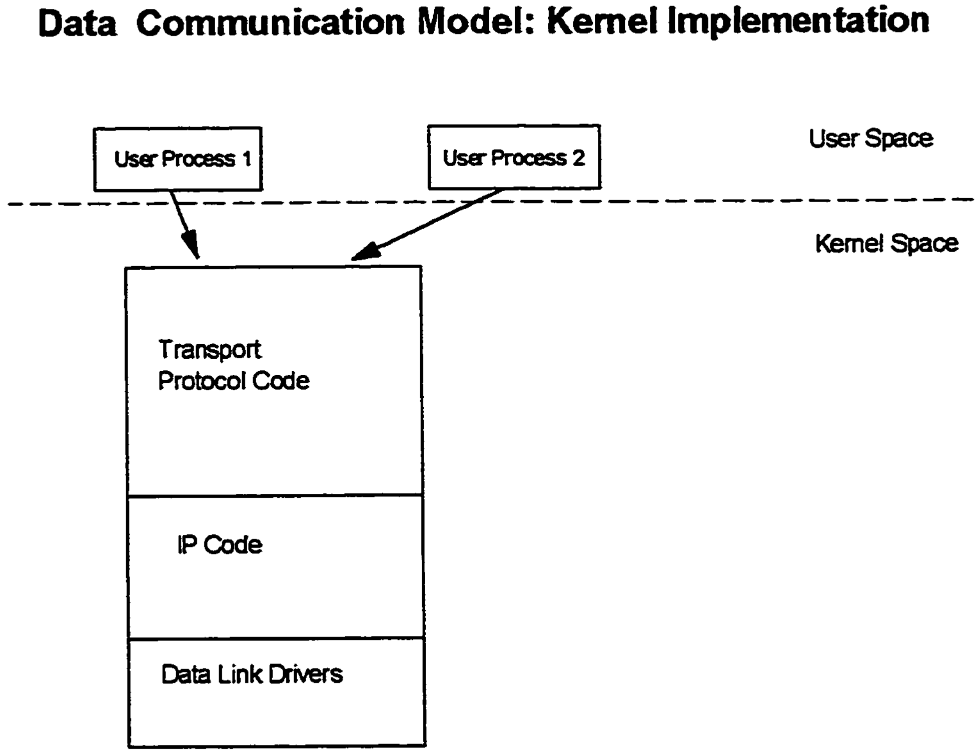


Figure 3. Kernel Implementation of a Transport Protocol

When a transport protocol implementation is hidden in the kernel, it is very awkward to provide enhancements to the transport protocol, having to update the kernel code for any changes. The privilege that is required to effect any development modification to the transport code would have to be "root" or "superuser", a privilege usually not granted to everyone, making any protocol update a system wide risk.

Figure 4 below shows an improvement in segregating the transport code outside of the kernel. A daemon is required to link the interactions of the transmission protocol with the ones of the operating system.

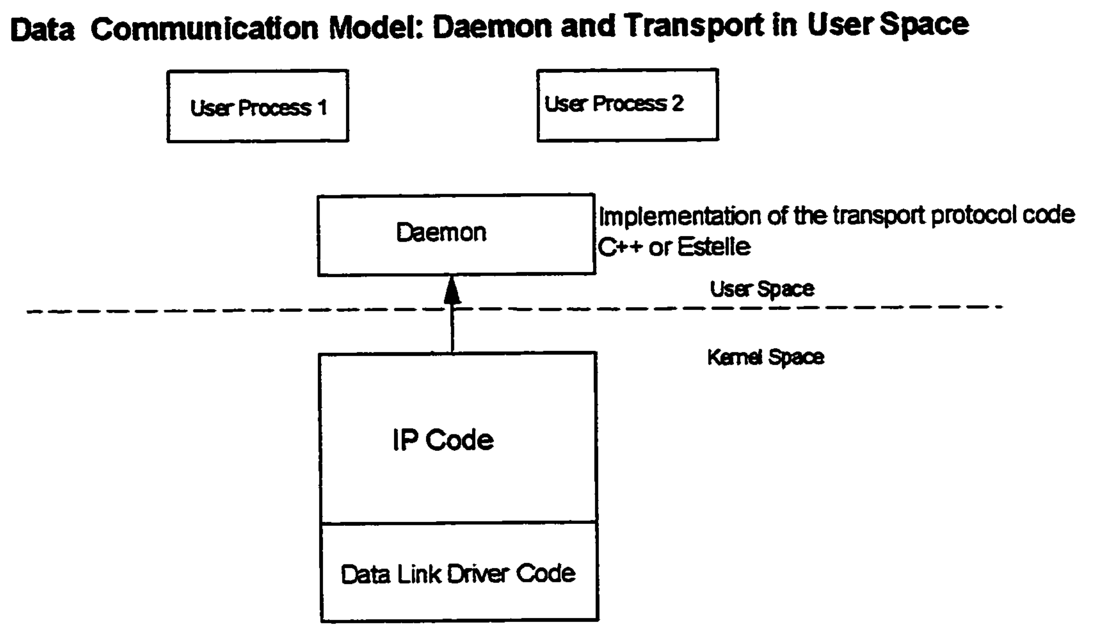

Data Communication Model: Daemon and Transport in User Space
User Process 1
User Process 2
Daemon
Implementation of the transport protocol code
C++ or Estelle
User Space
Kernel Space
IP Code
Data Link Driver Code

Figure 4. User Space Implementation of a Transport Protocol

In the next figure below, Figure 5, we describe the code layers that make up XTP. The C primitives serve to provide a more flexible programming environment for the user processes and the network interface, than what the PASCAL underlying code of the Estelle specification can provide. Again, a daemon is required to effect the communication link between the Kernel space layer and the User space layer.

Data Communication Model: The Processes and Code Layers

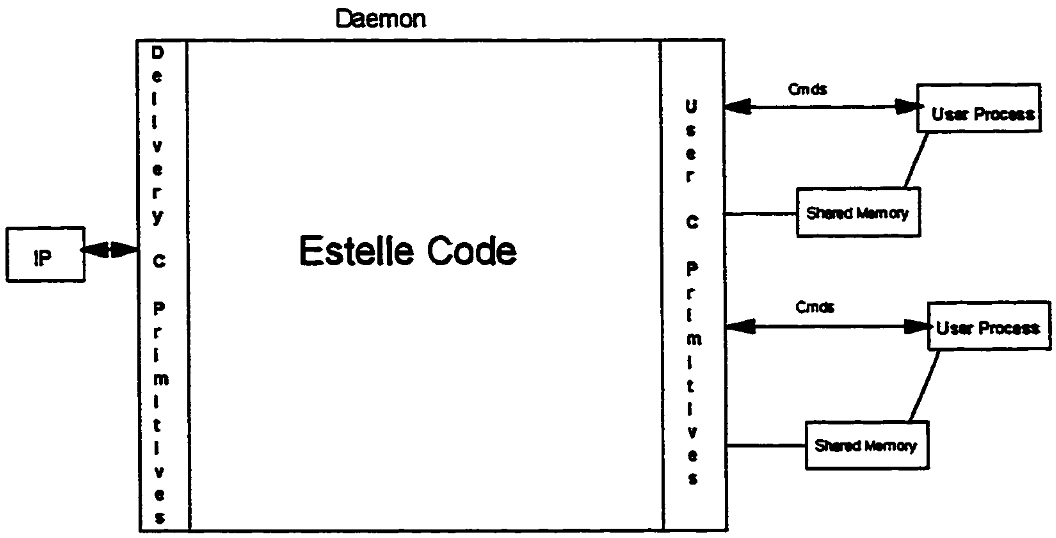


Figure 5. The Processes and Code Layers of Estelle XTP

Having introduced the conceptual components of the transport protocol with the above Figures 3, 4 and 5, the next figure below will start describing the unique structure of Estelle specification of XTP.
Figure 6 below is a picture diagram of the XTP protocol. The following references to modules and ESTELLE channel names can be located from this diagram.

2 ESTELLE XTP COMPONENTS DESCRIPTION

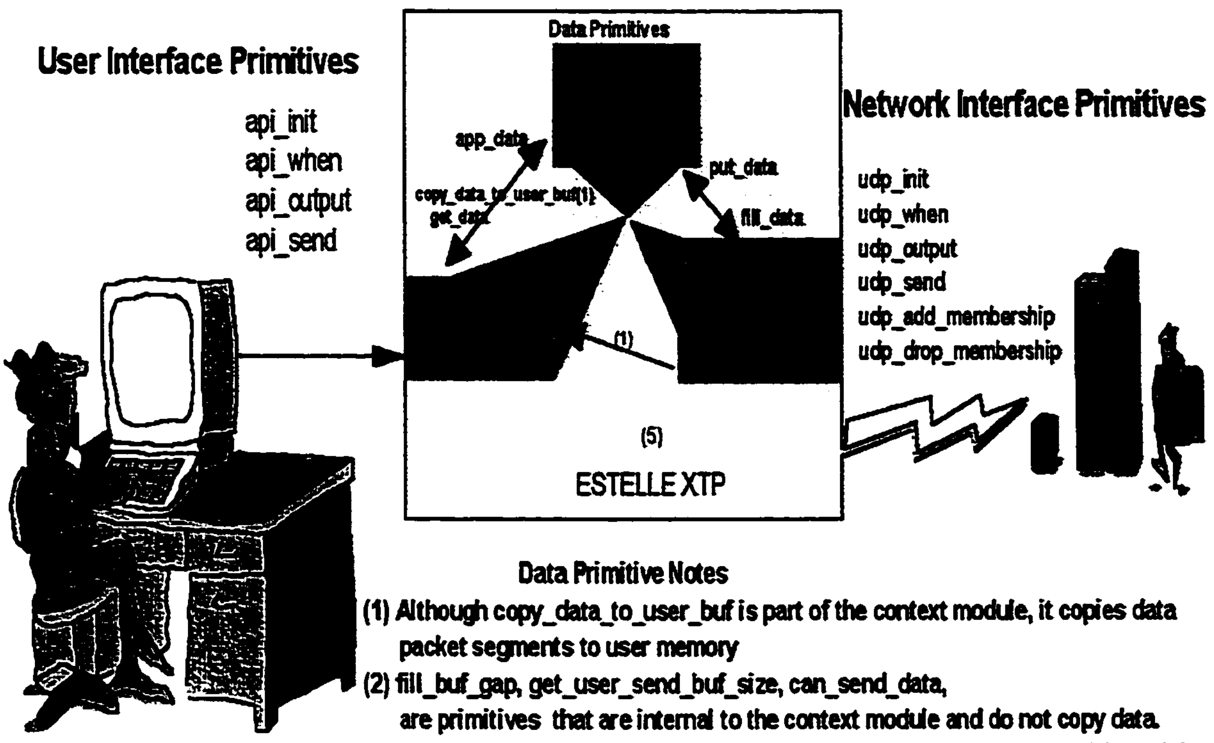
2.1 Components and Modules Overview

In the following sections, we will describe the major discrete components of ESTELLE SPECIFICATION of XTP 4.0 at the latest version described above. Before we proceed to list these components, Figure 6 below will help to locate the main interfaces that come into play, from the User to XTP and finally to the Network interface. It introduces the presence of copy operations, within the protocol, as well as the internal components of the context module and the packet management module, namely the ring buffers and the packet pool, respectively.

Below is a list of the components to be described in the following sections of this work:

- The USER Interface
- The Context Manager
- The Data Primitives
  - Concept
  - Description
- The Packet Manager
- The Network Interface
- The DAEMON

Before we start describing and analyzing the role of each component, we look at Figure 6 below, a pictorial representation of the whole user/XTP/network interaction is included following this paragraph. It should be used as a reference for data flow and components of the protocol.



Data Primitive Notes

(1) Although copy_data_to_user_buf is part of the context module, it copies data packet segments to user memory

(2) fill_buf_gap, get_user_send_buf_size, can_send_data, are primitives that are internal to the context module and do not copy data.

(3) rmv_rcvd_data_seg is called from both the packet_management module and the context module; there is no data movement.

(4) reg_rcvd_data_seg is called from the packet_management module; no data copy takes place

(5) clean_pak_pool is called from the XTP body transitions; this not a data copy operation

Figure 6. The USER, XTP and Network Interface Pictorial Design

2.2 The User Interface

The user interface uses two data structures to effect data movement between the interface and XTP. They are:

- A receive and a transmit buffer (rbuf and tbuf) also referred as shared memory buffer.

- A data structure (dt_rcv) that holds pointers. It is a temporary send/receive data descriptor.

The xtp_initialize user API must be called before any other API invocation, because it provides a structure definition for the shared memory buffers and initializes pointers to the allocated user environment.

Both of the above data structures are part of a higher level structure that is described below:

```
struct xtp_if_struct  {
....
....
struct shm_buf_type  tbuf, rbuf;
...
struct  data_param_type dt_snd,  dt_rcv;
...
...
}  *xtpif;
```

We will now describe the data structure used in the user interface to better understand how data is passed from the user environment to XTP. We will note the structures that do data copy versus data pointer reference, and this, with a spirit of reducing the data copying and speeding up the data overall data transfer.

We start with the temporary send/receive data descriptor structure: dt_snd and dt_rcv.

•dt_snd and dt_rcv are of type: data_param_type

•data_param_type is used in the TDAT*, TCON* and TCLS* primitives and is:

```
 RECORD
   (* data buffer *)
    dtbuf  :  Data_type;
     (* EOM, BTAG flags  *)
    eom    : BOOLEAN;
    btag   : BOOLEAN;
    (*  Flags used for tests - interpreted in TCONreq, TDATreq  *)
    sreq   : BOOLEAN; { SREQ value used in the initial packet  }
    dreq   :  BOOLEAN; { DREQ value in the last packet  }
 END;
```

•Data_type is the data holding definition type and is defined as:

```
 Data_type =
   RECORD
            nb_data  : 0..Max_user_data;
            data_ptr: INTEGER;
       { replaces:   data : User_data; }

       END;
```

The structure implements a pointer (data_ptr) to the user data area, an improvement over the last implementation that had User_data defined as a character string.

We now describe the shared memory buffer structure. After the description we will show how the data is copied, or referenced from one structure to the other. The shared memory buffer structure is defined as:

```
/* Shared memory buffer */

     struct shm_buf_type {
          int shmid;
          int len;
          char * buf;
        };
```

The shared memory buffer is attached by the xtp_reg function primitive, using the *shmat* UNIX function, to register the user with the XTP daemon. At the end of the association, the shared memory is detached from the XTP context using the *smhdt* UNIX function. Because the shared memory buffer is attached and detached as explained here, there is one copy of the shared memory per user process. This is represented as such in Figure 5 from the section 1.3.3.

When data is referenced in the shared memory buffer, no data movement happens, only a pointer to the data is set. Of course, if a string is copied to the referenced data pointer, as in "get_data", described later, a data copy will happen.

For send and receive requests, the shm data pointer is assigned using the following statements:

```
p->dt_snd.dtbuf.data_ptr = p->user_index;
p->dt_rcv.dtbuf.data_ptr = p->user_index;
```

Taken from the xtp_send and xtp_receive primitives respectively.

Other control variables are initialized in a similar way.

2.2.1 The User Interface Primitives

In this section, we briefly describe the primitives that are used by the User Interface. These primitives are not primarily concerned with data manipulation in the sense of copying and forwarding, but play a role in setting up the parts of the protocol that will be used to flow the data.

The format adopted to describe the primitive will be in describing the parts of the XTP specification requiring this support, and the transitions that are triggered to execute the calling of the such primitive.

- api_init

Initializes the datagram socket used for the communication between XTP and the application. It does so by binding the local address so that user can send information to this destination.

From the ESTELLE specification, the call is performed as part of the protocol initialization when the user channel is connected to the protocol, in a state referred to as S0, purely in naming convention.

- api_when

It checks for input datagrams from an application. It returns a Boolean value of true if data is present to be entered into the USER INTERFACE part of XTP, as detected by the socket call EDTFD_ISSET.

It is used as an XTP module transition Boolean condition, part of the trigger.

- api_output

Gets and checks the message from the socket and outputs the corresponding Estelle interaction. All of the types of Estelle interactions are represented by a case statement. Of interest to this work, when a transition involving data is encountered, a pointer operation is used to reference the data, and no data is copied through the user channel. The shared memory is mapped into structures that XTP uses, thereafter, to process the transition and enqueue the data, if any, in the context buffers.

The data structure used to map the data is data_param_type as described above, where Data_type is a 'pointer to data' structure. It is called from the XTP transitions responding, triggered by the api_when API.

- api_send

Sends the message corresponding to an output interaction to the application process as a UNIX domain datagram. It does the same work as api_output, but in the reverse direction. It is called to transfer data, when data transfer is involved, and all of the non-data related messages to the application. It uses the same code type as api_output, swapping the source and destination operands of the memcpy C function.

Contrarily to api_output, it is called for each type of interaction possible, and sets the interaction type within the datagram so that the primitive can handle the call according to the information passed to the application. No data copy is performed; the same structures as in api_output are used.

2.3 The Context Manager

The context manager is the most crucial part of the protocol, and it is where a large part of the features, both usage and performance, reside. It is the link with the user interface and the packet management modules, and as such communicates with both of these components. Because it is an important link between the functionally independent parts of the protocol, this introduces a requirement to store data within the protocol. Data storing and copying being the main thrust of this work, the context manager is covered in length in this section (2.3), and its subsections.

Within a transmission protocol, events happen asynchronously: A data packet is received from the delivery services, much at the same time as a new user requests to send some data. For the protocol to be able to service all of the simultaneous requests seamlessly, the protocol must be able to dedicate a small amount of processing to each of the requests at a given time. By doing this, it has to be able to jump from a state to another state that is completely irrelevant to the first one. The key to understanding how this is achieved is in the multiplexing of contexts. Each context describes the state of a given association that can be initiated, continued or terminated independently from one another.

To achieve the switch from one context to another without any loss of data, the data must be copied to the location where it will continue to be fetched when the processing resumes to the one context it switched from. To give an example, the action of filling up a series of packet shells with some data to be sent should not depend on the shared user memory between XTP and the user; rather this data should reside within the context ring buffers, simply because the filling of data packets is not a user process, it is an XTP context manager process. By keeping data locally to the process that handles this data gives XTP the ability to manage its module processes independently (see figure 7 and 8 below).

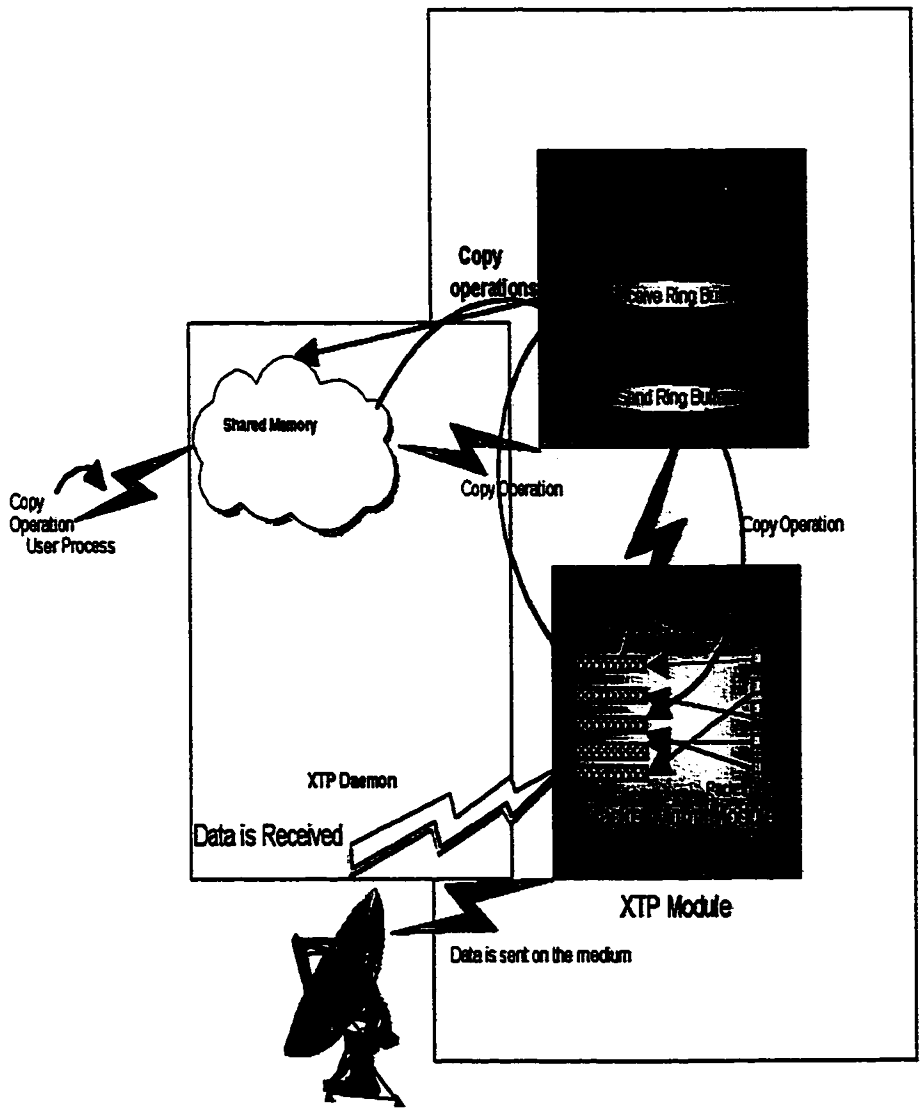


Figure 7. User Process, Unix Shared Memory, XTP and Data Delivery Services

The context manager's role is also to associate context and users. Such association will introduce fast lookup for already existing conversations and context instantiation when a FIRST packet is received. In the next two sections, we document the most important data structures of the context manager, and then we document the procedures of the context module that call the data primitives.

2.3.1 The context send/receive buffers: the ring buffers

The ring buffers are the main data store of the XTP Context module. Figure 8 below is a picture representation of the ring buffers, part of the Context module:

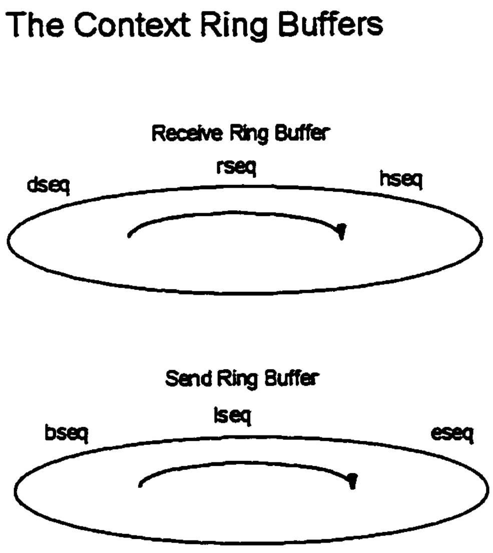


Figure 8. The Context Ring Buffers

Referring the Figure 8 above, the buffers flow pointers are defined as follows:

The receive buffer has the following three position flow variables:

- dseq: delivered (to the user) sequence
- rseq: contiguously received sequence not yet delivered to the user
- hseq: high received sequence that includes receive sequence gaps

The send buffer also has the following three position flow variables

- bseq: begin sequence of data sent but not yet acknowledged
- lseq: last sequence of data sent but not yet acknowledged
- eseq: end sequence that is available to be sent but is not sent yet

It is now appropriate to mention that the current version of the Estelle specification of XTP uses a block transfer method to read and write to the ring buffers. This feature has enhanced the protocol's performance dramatically over the original implementation that performed ring buffer updates on a one by one character basis.

Following is the definition of the ring buffer according to the data structure of the C primitives:

```
{Send/receive data buffer}

   ring_buffer = RECORD
         data : ARRAY [1..Max_buf_lng] OF CHAR;
```

The ring buffer size is currently set to the arbitrary value of 24576 bytes.

Some additional control variables are also defined to map and use the ring buffer:

```
 Max_buf_lng; {Pointer to first data byte in buffer}
     cnt  : 0..Max_buf_lng; {Counter of data bytes in buffer}
     lng  : 1..Max_buf_lng; {Actual buffer size}

   END;
```

and the buffers are instantiated as:

```
(** Send buffer **)
   sbuf : ring_buffer;     { Data buffer }
```

and...

```
(** Receive buffer **)
   rbuf : ring_buffer;       { Data buffer }
```

within the BODY of the Context module: Context_body .

These structures define a character array, and we will next show the data copying that takes place from the shared memory buffers to the ring buffers.

2.3.2 The Data Primitives

For a quick reference about the placement and role of the data primitives, the reader is urged to refer to the Figure 6, above.

2.3.2.1 Concept

To enhance the performance of the protocol, the copy operations have been consolidated in common routines coded in C. This consolidation made it possible to free the ESTELLE specification from low level formatting of the data packets and also streamlined the data copy operations to the reusable C coded procedures: the Data Primitives. This concept was a dramatic improvement over the spread copy operations that were found throughout the specification in the earlier version of ESTELLE XTP.

2.3.2.2 Description of the Data Primitives

In this section, we describe each data primitive, by name enumeration, each followed with a functional description. To bring substance to the enumeration, pertinent data structure is added when appropriate and a brief mention of the calling procedure, from the ESTELLE text, is mentioned.

- **app_data**

Data Transfer: From Shared Memory to Context Buffer

Copies data from user's send buffer to the context's send buffer and assumes there is enough room in the buffer to store the user data.

This is a data copy operation, not a pointer propagation. It is called from XTP's context module procedures. Once the data is copied into the ring buffer, a TDATcnf message is output through the context-user channel to acknowledge the copied data. This is done using a data pointer to the shared memory buffer; thus, no additional data copy is done.

- **put_data**

Writes data from the data segment in a received packet to the context buffer. This is a data copy operation, not a pointer propagation. It is called from XTP's context module procedures.

- **fill_data**

Transfers data from the Context Buffers to the Packet Pool. This is done by copying data from the context's send buffer to a packet in the packet pool.

This is a second copy operation for data originating from user shared memory buffer. The function get_dist64 evaluates the length of data to be transferred to the packet pool by inspecting low_seq and high_seq. This operation is part of the fill_data_packet procedure from the context module. The primitive is also called from another procedure in the context module's body when a FIRST packet is received, containing data; this happens when an idle context receives a connection request.

Once the packet has been copied into from the context buffer, the channel interaction specifies a pointer to the copied data. This is expressed as follows, the channel interaction is reported first, followed by the corresponding data structure definitions:

```
    OUTPUT cl.first(first);     and
       OUTPUT cl.data(dp);
```

The channel definition:

```
    CHANNEL Context_Channel (context, xtp);

        BY context, xtp:
        first  (p : Xtp_first_packet);
        data   (p : Xtp_data_packet);
        cntl   (p : Xtp_cntl_packets; id : lptr_type);
        diag   (p : Xtp_diag_packet; id : lptr_type);
```

The data structure passed through the channel is:

```
  Xtp_First_packet =

  RECORD
     header: Xtp_header_type;
     address_seg: Address_seg_type;
     traffic_seg: Traffic_seg_type;
     data_seg: Data_seg_type;
  END;

  Xtp_Data_packet =

  RECORD
     header: Xtp_header_type;
     data_seg: Data_seg_type;
  END;
```

...where Data_seg_type is defined as:

```
 Data_seg_type = { Modif. for implem. }

  RECORD
     nb_data: 0..Data_seg_max_lng;
     data_ptr: INTEGER;
  END;
```

- **get_data**

Copies data from the context's receive buffer to the user's receive buffer. Again, this is a data copy operation. It is the opposite of put_data but essentially uses the same techniques and the same data structure. The data structure is not included here because it was described in put_data.

The copy operation is:

```
        memcpy(to_ptr, b->data + head, amount);
```

Where b is the ring buffer, to_ptr was assigned from the shared memory with:

```
        char* to_ptr = user_info[d.data_ptr].rbuf.buf;
```

The get_data primitive is referenced from the context module in the "context_delivers_rcvd_data_when_input_active" procedure.

The channel output only delivers a pointer to the copied data:

```
        OUTPUT cu.TDATind(data_param)
and
        OUTPUT cu.TCLSind(data_param)
```

- **copy_data_to_user_buf**

Copies data directly to shm buffer from the data segment of an XTP packet

```
memcpy(user_info[get_key_index(kv)].rbuf.buf, data_buf->data_seg, len)
```

Where rbuf is as described before and data_buf is of type xtp_packet_buf defined as:

```
        struct xtp_packet_buf {
            /* number of owners <= max number of multicast receivers */
            int count;
        /* owners flags: context index -> bit position */
        word32 owners;
        char data_seg[PAYLOAD_SIZE];
        };
```

and:

```
    #define PAK_POOL_SIZE 64    /* number of elements in pak_pool */
    #define PAYLOAD_SIZE 4096   /* size of each data seg in pak_pool */
```

and "len" represents the number of bytes to be read from the packet buffer expressed as nb_data defined here:

```
rseqv := p.header.dlen;
dseq := p.traffic_seg.tlen + p.address_seg.alen;
hseq := dseq;
cntl_to_send := p.header.cmd.sreq;
nb_data := p.header.dlen - (p.traffic_seg.tlen + p.address_seg.alen)
```

It is invoked from the procedure: context_listening_rcvd_first_packet.

- **put_data vs. copy_data_to_user_buf: which one to use and when:**

Copy_data_to_user_buf is a means to spare one data copy. It is a performance option to improve the throughput of the protocol. Put_data is the regular way about the protocol structure, copying the data segment portion of a received packet to the context ring buffer, before it gets copied to the user shared memory buffer or shm buffer. We will trace put_data and report on when the alternative copy_data_to_user_buf is used, in the same procedures as put_data, since it is never used in a procedure where put_data is not used

- Store_data_with_error_control

If this is new data or a span within the existing sequence space, put_data is used. Copy_data_to_user_buf is not used here.

- Context_listening_rcvd_first_packet

Both methods are used here and put_data is used when we must delay the delivery of the data because:

- ❖ there is no shm receive buffer yet

  or

- ❖ some qos options are not set yet and...

  The packet header does not specify "noerr or wclose"

If the above conditions are met, then copy_data_to_user_buf is used, provided all of the data can fit in a smh receive buffer. If the data to be received is larger than a receive shm, then a call is made to copy_data_to_user_buf with the shm length, followed by a call to put_data to store the rest of the data into a context receive ring buffer.

- context_joining_rcvd_initial_first_packet

  Only put_data is used here.

- context_received_data_packet

  Only put_data is used here with the following transitions and channel condition:

  ```
  FROM first_sent, first_rcvd, active
  WHEN cl.data(p)
  PROVIDED (in_state <> in_inactive)
  ```

- **fill_buf_gap**

Fill with '*' character a gap in the receive buffer (user specified character, in future versions) Used in noerr mode, to mark lost data segments. Fails if gap is bigger than buffer size!!!

- **get_user_send_buf_size(word64 kv)**

This function is self-explanatory and returns the user buffer size as the kv 64 bits integer variable.

- **can_send_data**

This boolean function verifies that we have not consumed all of the packet shells from the packet pool.

- **rmv_rcvd_data_seg**

This primitive decrements the count of data segments to be received; it frees a packet shell.

- **reg_rcvd_data_seg**

This primitive is called to assist in registering a new owner: it gets its index key and increments the number of owners.

- **clean_pak_pool**

Release all data buffers from pak_pool allocated to a context, when the context is released. The normal solution would be to have, for each context, a linked list of the allocated buffers. The current simplified solution has been implemented for the preliminary tests only.

2.3.3 ESTELLE XTP Calls to the Data Primitives

At this point, it is proper to first, level set some Estelle terminology that will be used here, in relating the state transitions, and also to describe how data communication is done when using Estelle.

Estelle is an extended finite state automata language. Some of its coding constructs are:

- State Transitions using the keywords: FROM and TO. A transition condition checking is possible using the keywords PROVIDED and WHEN.

- The pipes providing communications are called CHANNELS. CHANNELS link Estelle modules to one another by creating interaction points (IP). Each interaction points are tagged with roles, each role defining the type of data that can flow within a channel. This arrangement makes it possible to anticipate state transitions resulting from data that is exchanged through the CHANNELS.

- The OUTPUT statement triggers the data flow through the named CHANNEL.

- The concept of a module, a module body and the instantiation of a module is also part of Estelle. It provides a way to switch the body of a module, according to its function requirements and to reuse code by instantiating modules as required (e.g. a context) with a process or an activity.

This description is summary. A definition reference for Estelle is included in the Glossary appendix, under Estelle.

Before we start going through data primitives and describe their functions, the figure 9 below relates the complete Estelle specification structure, with the essential data components that are copied across modules. All CHANNELS are named and are also tagged with their role at the named IP (Interaction Points) represented by the large dots.

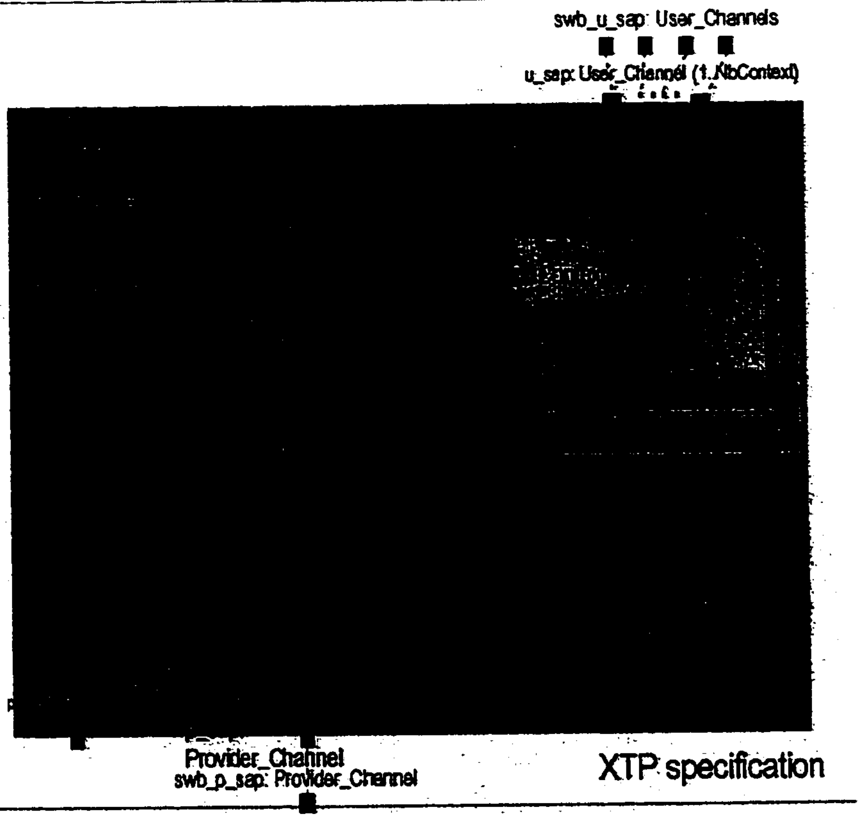

swb_u_sap: User_Channels
u_sap: User_Channel (1..NbContext)
Provider_Channel
swb_p_sap: Provider_Channel
XTP specification

Figure 9: Modular representation of the ESTELLE XTP Specification

Following is the enumeration of the data primitive calls as traced from within the Estelle specification

- app_data

Writes (copies) data from user' (shm) send buffer to the context's send buffer.

It is called from:

- context_idle_rcvd_con_req

Following the transition:

FROM idle

TO first_sent

On a FIRST packet type, if there is user data in the packet, the procedure stores user data in sbuf. Data segment should fit in sbuf because it is the first sequence of data for that user context.

- ❖ context_stores_user_data

When the following transition occurs:

```
FROM active
WHEN cu.TDATreq(data_param)

PROVIDED (out_state = out_active)
```

In this calling sequence, the sender stores user data in the context's ring buffer: app_data(sbuf, dtbuf);   where sbuf is the context send ring buffer and data_param is a pointer (to shm) type of data structure that is used in channel I/O operations to avoid data copying.

- ❖ context_received_close_request

When the following transition happens:

```
FROM active
WHEN cu.TCLSreq(data_param)
PROVIDED (out_state = out_active) { Else Error_ind }
```

The call is as follows:

app_data(sbuf, dtbuf);

After verifying that there is data present in the shm that has not been added to the context send ring buffer (sbuf), app_data is called to add this data to the context buffers. Again, a pointer type of data structure is used in the channel operation to avoid propagating strings of data internally within the components of XTP.

- put_data

Writes (copies) data from the data segment in a received packet, to the context (ring) buffer. This primitive is called from the ESTELLE specification by the following procedures:

❖ store_data_with_error_control

Received data processing in error control mode. For a received data segment at sequence 'seq' and with length 'lng', the procedure updates the variables 'rseqv', 'in_nspan' and 'in_spans' and stores the data - if possible. 'Start' gives the index of the first byte of interesting data in the received segment. More specifically, the function performed by put_data in this procedure is as follows:
There are two situations where new data can arrive, hence two calls to put_data:

- The data is included in the sequence of received data or added (appended) to the data.
- The new data is beyond the sequence of received data, and thus constitute a new span, with a gap between the previously end of received sequence and the beginning of this new data.

For the first case, the segment is added to or included in the complete sequence. If new data exists, store new data and advance rseqv.
In the second case, a new span is added provided we did not exceed the maximum number of possible spans.

Most of the variables mentioned above are intuitive by their names. The ESTELLE text may be consulted for a complete description.

❖ context_listening_rcvd_first_packet

This procedure is called as the result of the following transition:

FROM listening
TO first_rcvd
WHEN cl.first(p)

The process of this procedure is self-explanatory as its name suggests. At this point, there is no questions of spans, and the data is moved from the packet buffer to the context receive ring buffer, the put_data thus does this copy operation. This operation is referred to as storing the data, as opposed to delivering the data. Delivering the data consists of copying the data to the user buffer (shm). The storing operation is done in the following cases:

- The delivering of the data must be delayed because it is part of a reliable multicast operation.
- The data to be received is longer than the user buffer; in this case a complete user buffer is copied and the remainder of the data is inserted into the context ring receive buffer.

❖ context_joining_rcvd_initial_first_packet

Typically, this procedure involves the storing of data in a multicast operation. As mentioned above data is stored instead of delivered for a reliable multicast operation.

The procedure is called from the following commented transition:

```
{ Joining multicast receiver: initial First received }
FROM join_sent
TO first_rcvd
WHEN cl.first(p)
PROVIDED NOT remote_xtp_alive
```

The data is copied from the packet buffer to the context receive ring buffer.

❖ context_received_data_packet

As the name suggests, the packet buffer is copied into the context receive ring buffer. In this instance, we are dealing with a data packet as previously we had been describing the logic of processing a FIRST packet.

The procedure is triggered from the transition:

```
(* Incoming packets *)
 { New: Incoming Data packet }
FROM first_sent, first_rcvd, active
WHEN cl.data(p)
PROVIDED (in_state <> in_inactive)
```

No updates to the user are done in this operation. A data packet updates the context buffers.

- fill_data

Transfers (copies) the context buffer to the packet buffer.

This primitive is called from:

- ❖ PROCEDURE fill_data_packet

The data from the context send ring buffer is copied to a packet shell buffer, and the data is removed from the context buffer if the 'noerr' option is in effect, or is left in the context ring buffer if the 'noerr' option is not used.

- ❖ context_idle_rcvd_con_req

Triggered from the transition:
FROM idle
TO first_sent
WHEN cu.TCONreq(opt_param, cls_param, data_param,
    cntx_param, src_addr, dst_addr)

The same observations as for 'fill_data_packet' apply as for the 'noerr' option.

- get_data

This primitive delivers data from the context receive ring buffers to the user, via the shared memory (shm). The primitive is called from the following procedures in the ESTELLE specification:

- ❖ context_delivers_rcvd_data_when_input_active

  {Deliver received data to the user when input is active;}
  It is called as a trigger result from the transition:

  FROM active
  PROVIDED (in_state = in_active)

```
        AND (cmp64(rseqv, dseq) > 0)
        AND (user_recv_buffers > 0)
```

The data is copied from the context receive ring buffer (rbuf) to the user space or user data.

- context_delivers_data_after_getting_wclose_or_end

  {Deliver data after rcvd (wclose or (end and rclose)).

  End of graceful close or abort with output stream closed. }

The procedure is triggered as the result of the following transition:

```
        FROM active
        PROVIDED (in_state = in_deliver)
         AND (cmp64 (rseqv, dseq) > 0)
         AND (user_recv_buffers > 0)
         AND (NOT rcv_cls.endf OR rcv_cls.rclose)
```

The last data segment is delivered from the context receive ring buffer to the user data:

get_data(rbuf, dtbuf);

- context_flushes_receiver_buffer_when_end_rcvd

This call to get_data is made because:

{Association aborted by remote user.

Receiver buffer not empty, flush buffer}

The process of flushing the buffer is triggered by the following transition:

```
        FROM active
        PROVIDED (in_state = in_deliver)
```

AND rcv_cls.endf AND NOT rcv_cls.rclose

This is another case of processing the last data segment, in delivering it to the user, from the context receive ring buffer. The calling sequence is the same as the prior call: get_data(rbuf, dtbuf);

- copy_data_to_user_buf

This primitive is used when we will deliver the data directly, bypassing the storing of the data within the XTP protocol. Delivering the data consists of copying the data from the XTP data segment from a received packet, directly to user data (or shared memory: shm).

This primitive is called from the following procedures within XTP.

❖ context_listening_rcvd_first_packet

Is triggered by the transition:

FROM listening
TO first_rcvd
WHEN cl.first(p)

The logic that drives the delivery of data to user memory instead of storing the data has already been mentioned in the description of the 'put_data' primitive. Data is delivered if for the transmission:

- There are free user buffers
- We are not involved in multicast at this instance
- The options are NOT (noerr or write close)

The instantiation of the call is as follows:
copy_data_to_user_buf(loc_key.value,

p.data_seg,nb_data,TRUE);

There are two ways, here, in which the primitive is called:

- ❖ The data to deliver fits in user data space: all of the data is copied.
- ❖ The data to deliver is larger that the user buffer size and one buffer is copied. The remaining data is stored and delivered in a separate process.

• get_user_send_buf_size

This primitive, unlike the other primitives described to this point, does not copy data. It just returns a size value and it is used while initializing a new context.
It is called from the following XTP procedures:

- ❖ init_context

The role of this procedure is to initialize the context variable:
user_send_buf_size := get_user_send_buf_size(loc_key.value);

• can_send_data

This primitive is a Boolean function that returns the boolean value of the ability of adding new data into a packet from the packet pool, to be able to send data.

This primitive is called from the following procedures in the ESTELLE specification:

- ❖ context_idle_rcvd_con_req

Triggered from the following transition:

```
FROM idle
TO first_sent
WHEN cu.TCONreq(opt_param, cls_param, data_param,
          cntx_param, src_addr, dst_addr)
```

This is the instance where the context transfers a data segment into a FIRST packet. A test is done to verify that there are free packet shells.

❖ context_sends_data_packet

This reference to the Boolean function is part of the trigger for the transition of this procedure; following is the transition:

```
FROM active
PROVIDED (out_state = out_active)
      AND (cmp64(eseq, lseq) > 0)
      AND NOT sync_cond
      AND not_flow_ctrl_stop AND not_rate_ctrl_stop
      AND (out_nspan = 0)
      AND can_send_data { implem. }
```

❖ context_retransmits_data_packet

Similarly, to the procedure above, again the call is part of the trigger for the transition as reported here:

```
FROM active
 PROVIDED ((out_state = out_active) OR (out_state = out_close))
      AND NOT sync_cond
      AND not_rate_ctrl_stop
      AND (out_nspan > 0)
      AND can_send_data { implem. }
```

- rmv_rcvd_data_seg

This primitive removes a received packet from the packet buffer pool. It calls put_pak_buf, a UDP support function that adjusts the top of the packet pool downward, reflecting the de-allocation of a packet buffer.

It is called by the following procedures from the ESTELLE specification:

- ❖ context_rcvd_duplicate_first

It is triggered by the transition:
FROM first_rcvd, active
WHEN cl.first(p)
PROVIDED NOT initiator
The data segment is removed from the packet pool by this primitive in the case where we received a duplicate FIRST packet.

- ❖ context_joining_rcvd_initial_first_packet

The procedure is triggered by the transition:

FROM join_sent
TO first_rcvd
WHEN cl.first(p)
PROVIDED NOT remote_xtp_alive

Once the data is stored, the packet buffer is removed from the packet pool.

- ❖ context_joining_rcvd_duplicate_first_packet

The procedure is triggered by the transition:

FROM join_sent
WHEN cl.first(p)

PROVIDED remote_xtp_alive

The same observation as the previous procedure dealing with a duplicate FIRST packet applies.

- ❖ context_received_data_packet

The procedure is triggered by the transition:

```
FROM first_sent, first_rcvd, active
WHEN cl.data(p)
PROVIDED (in_state <> in_inactive)
```

If no new data is received, the packet is removed from the pool.

- ❖ context_discards_data_packet_when_input_closed

The procedure is triggered by the transition:

```
FROM active, inactive
WHEN cl.data(p)
PROVIDED (in_state = in_inactive)
```

Because the data is discarded, the packet pool must reflect this action.

- ❖ context_inactive_discards_first_packet

The procedure is triggered by the transition:

```
FROM active, inactive
WHEN cl.first(p)
```

PROVIDED (in_state = in_inactive)

❖ pack_mngm_rcvd_first_packet

The procedure is triggered by the transition:

```
TRANS { First }
WHEN net.NDATreq(src_addr, dst_addr, packet)
PROVIDED packet.ptype = First_pak
```

This instance of the call is done to reject with DIAG a received FIRST packet.

❖ pack_mngm_rcvd_data_packet

The procedure is triggered by the transition:

```
TRANS { Data }
WHEN net.NDATreq(src_addr, dst_addr, packet)
PROVIDED packet.ptype = Data_pak
```

Here we are rejecting a packet that is non-XTP.

- reg_rcvd_data_seg

This primitive registers a new user by getting a user key and incrementing the number of packet buffer users from the packet pool. This is part of setting the context flags.

It is called from the following procedures from the ESTELLE XTP modules; we also report the transitions triggering the activation of the procedures.

❖ pack_mngm_rcvd_first_packet

The procedure is triggered by the transition:

```
{Incoming packets}
TRANS {First}
WHEN net.NDATreq(src_addr, dst_addr, packet)
PROVIDED packet.ptype = First_pak
```

The context key is built and the new packet owner is added by incrementing the number of packet owners by one.

❖ pack_mngm_rcvd_data_packet

The procedure is triggered by the transition:

```
TRANS { Data }
WHEN net.NDATreq(src_addr, dst_addr, packet)
PROVIDED packet.ptype = Data_pak
```

The data flows from the packet management module to the context. Verification is made that the context exists, and if so, the data is delivered and the received data packet is registered.

- clean_pak_pool

This primitive de-allocates the packets from the packet pool. It resets the pointers to represent the removal of the packer buffers. It is called from the following ESTELLE XTP procedures.

❖ xtp_removes_inactive_context

The procedure is triggered by the transition:

```
{Remove inactive context}
TRANS
  ANY k : Context_range DO
  PROVIDED pmngm.cntx_rec[k].status = inactive
```

A FALSE indicator is sent on the usap (user) channel to indicate that the context is removed, and the clean_pak_pool primitive de-allocates the buffers.

This concludes the tracing and documentation of the data primitives from the ESTELLE XTP specification.

2.4 The Network Interface

The calls to the network interface, from ESTELLE XTP, is supported by the following primitives:

- udp_init
- udp_when
- udp_output
- udp_send
- udp_add_membership
- udp_drop_membership

Next, the primitives are described in the same style as the user interface and the context primitives, starting with a brief description of its function, followed by the part of the ESTELLE text where it is activated and finally, the transitions triggering the use of the primitive is mentioned. UDP is used in this case, because it requires less system privileges to run and gather statistics for evaluation and comparison.

- udp_init

The primitive opens a UDP socket. It sets a buffer size for both the send and receive buffers. The socket address is bound and finally the packet pool is initialized (the packets links are numbered).

It is part of the XTP module and is called at system state initialization (S0) at the just following api_init.

- udp_when

This Boolean function tests if a UDP datagram has arrived, provided there is free space in the packet pool. It is used as part of the transition triggering condition for udp_output described next.

- udp_output

This primitive outputs an XTP packet, received in a UDP datagram, as an Estelle interaction (OUTPUT ip.packet).
The XTP packet is built from the received UDP datagram, the corresponding header is constructed according to the determined packet type. In the case of a data datagram, our prime interest, a data pointer is used instead of copying the whole data sequence to the packet body. This is an improvement over the prior versions of the specification. Upon completion, the packet is available to XTP for further processing by the context module.

The primitive is triggered from the udp_when Boolean primitive, signaling the arrival of a datagram.

- udp_send

This primitive sends an XTP packet from an Estelle interaction in a UDP datagram it essentially does the reverse function of udp_output.

As for udp_output, no data copy operation takes place. A data pointer is used to locate the data to be sent on the network, in the case of a data packet or a first packet in which data is present. The datagram header is built in this primitive.

The primitive call, from XTP, is triggered from the XTP module transition:

```
WHEN swb_p_sap.ndatreq
```

- udp_add_membership

This primitive does the UNIX socket calls to add a member in a multicast scenario, using the supplied multicast address. It returns error conditions. It is triggered by the following transition from the XTP Body transitions from the following procedures:

  - **xtp_creates_listening_context**

```
(* Context management *)
TRANS
 { Create listening context }
ANY user_id : Context_range DO
WHEN u_sap[user_id].TLSTreq(opt_param, accept_param, cntx_param,
src_addr)
```

❖ **xtp_creates_initiating_context**

From the triggering transition:

```
{ Create initiating context }
ANY user_id : Context_range DO
WHEN u_sap[user_id].TCONreq(opt_param, cls_param, data_param,
                cntx_param, src_addr, dst_addr)
```

❖ **xtp_creates_joining_context**

From the triggering transition:

```
{ Create joining context }

 ANY user_id : Context_range DO
 WHEN u_sap[user_id].TMJNreq(opt_param, cntx_param, uct_addr,
mct_addr)
```

- udp_drop_membership

This primitive drops multicast membership by issuing the UNIX socket call IP_DROP_MEMBERSHIP. Like the ADD MEMBERSHIP primitive, it is part of the XTP BODY transitions, and is not involved in any data copying. It is reported here, like the 'ADD MEMBERSHIP' primitive for completeness of the UDP primitives.

It is called by the following triggering transition:

```
{ Remove inactive context }
 TRANS
   ANY k : Context_range DO
   PROVIDED pmngm.cntx_rec[k].status = inactive
```

from the procedure: **xtp_removes_inactive_context**

The Network Interface Summary Statement

As described above, no additional procedures copy data to the packet pool. The packet pool being the network interface repository structure. Rather, pointers are maintained, and these pointers are passed through the ESTELLE channels via the interaction points. The data is copied to the packet shell buffers, by the context data primitives. These copy operations, along with the copies to the ring buffer, are the only copy operations within XTP and constitute the XTP data required to be able to maintain a suite of received sequences for reliable transport of information.

3 The Channel Activity Trace

In this section each channel activity, as traced by the OUTPUT statement in the ESTELLE code is produced here and examined for data copying. The required data structure is also reported for reference. For the relative function of the mentioned channels throughout this section, the reader is encouraged to refer to the Figure 1 in the previous chapter: ESTELLE XTP V5 DIAGRAMS.

We now trace the "OUTPUT" statements, relevant to data manipulation, and report if data copying happens, or if data referencing using pointers is done.

First, we view again the data structure used for shared memory referencing:

```
   Data_type =
   RECORD
            nb_data  : 0..Max_user_data;
            data_ptr: INTEGER;
       { replaces:   data : User_data; }
END;
```

In addition, dtbuf has type of Data_type.

We will skip OUTPUT statements that deal with CNTL, DIAG... and report the statements dealing with data:

- OUTPUT cu.TDATcnf(dtbuf);

  from: context_idle_rcvd_con_req

- OUTPUT cl.first(first);

Where first also has the same data structure as a data packet for the data part of the packet. It is called from: context_idle_rcvd_con_req .

- OUTPUT cu.TCONind(rem_opt, cls_param, dt, rem_addr, loc_addr);

Where dt has data_param_type which also has a type of data_type using a pointer to data reference instead of being a character array. It is called from: context_listening_rcvd_first_packet

- OUTPUT cu.TDATcnf(dtbuf);
  from: context_stores_user_data

- OUTPUT cu.TRDYind(dtbuf)
  from: context_allows_user_to_resume_sending_data

- OUTPUT cu.TDATind(dt_par)

Where dt_par is of type data_param_type, again a pointer structure of derived from the same data structure as the previous OUTPUT statements. It is called from: context_gets_recv_req_from_user

- OUTPUT cu.TDATind(data_param);

Where data_param is derived from data_param_type again a pointer reference structure. It is called from:
context_delivers_rcvd_data_when_input_active.

- OUTPUT cu.TCLSind(data_param);

from: context_not_multicast_sender_rcvd_Cntl_or_Ecntl_packet and also from context_not_multicast_sender_rcvd_Tcntl_packet

- OUTPUT cl.first(first);

from: context_starts_timeout_recovery

and: context_retries_when_wtimer_expires_during_recovery

and: context_tries_recovery_when_ctimer_expires

- OUTPUT cu.TDATcnf(dtbuf)

from: context_received_close_request

- OUTPUT cu.TCLSind(data_param)

and: OUTPUT cu.TDATind(data_param)

from: context_delivers_data_after_getting_wclose_or_end

- OUTPUT cu.TDATind(data_param)

from: context_flushes_receiver_buffer_when_end_rcvd

- OUTPUT cl.first(first)

from: context_initiator_treats_diag_invalid_context_during_setup

- OUTPUT cnx[k].first(packet.first)

from: pack_mngm_rcvd_first_packet and first has a data part of type data_seg that has already been defined as a data pointer structure type.

- OUTPUT net.NDATreq(host_addr, src_addr, pak)

also from: pack_mngm_rcvd_first_packet. Pak has type of: packet_type which has pointer to data for packet types First and Data, and has no (pointer to) data reference for the other type of packets.

- OUTPUT net.NDATreq(host_addr, src_addr, pak) is also done from the pack_mngm_rcvd_jcntl_packet procedure.

- OUTPUT cnx[k].data(packet.data) where the data field of the packet record is of type data_seg, a data pointer structure, called from:
  pack_mngm_rcvd_data_packet

- OUTPUT net.NDATreq(dst_addr, src_addr, pak) again a data pointer structure from the procedure:
  pack_mngm_rcvd_cntl_ecntl_packet
  and the pack_mngm_rcvd_tcntl_packet procedure

- OUTPUT net.NDATreq(host_addr, cntx_rec[k].remote_addr.host, pak) from the pack_mngm_sends_first_packet procedure

- OUTPUT net.NDATreq(host_addr, cntx_rec[k].remote_addr.host, pak) from the pack_mngm_sends_data_packet procedure

- OUTPUT iu.TCONreq(opt_param, cls_param, data_param,cntx_param, src_addr, dst_addr)
  where data_param has been defined as a data pointer structure.
  It is called from: xtp_creates_initiating_context

We conclude that the channel output data interaction has been optimized, and that no data copying takes place.

4 Data Transmission Performance Benchmark

At this point, in this section, it is appropriate to get some transmission performance numbers together, some from the SANDIA XTP and compare these results with simple data transmission results using the ESTELLE XTP Specification.

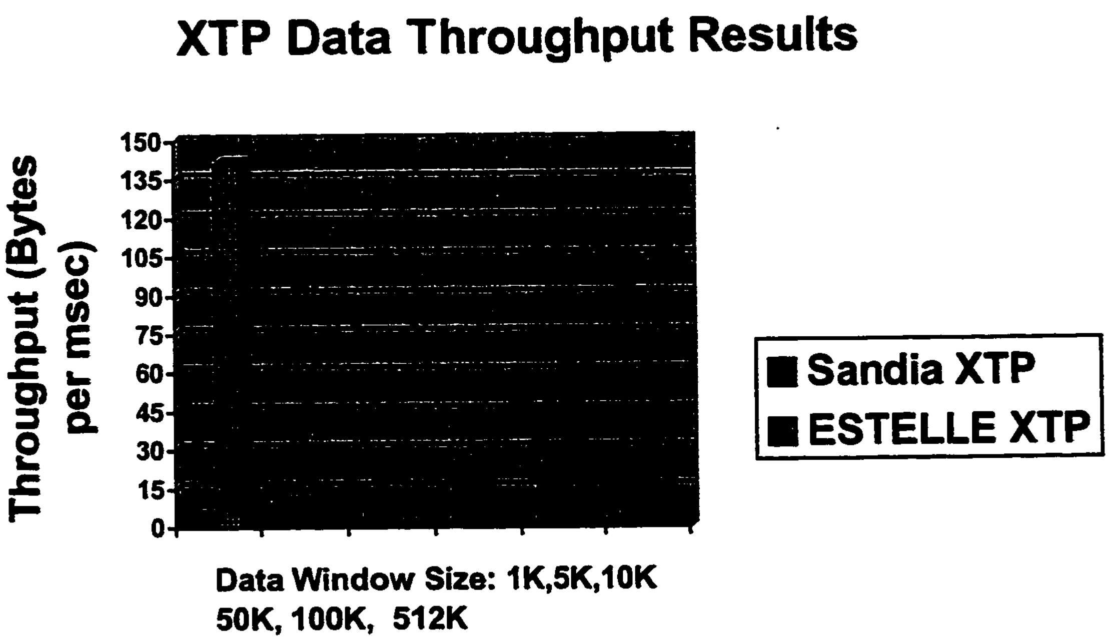
4.1 Description and Credits of the Performance Data

The SANDIA XTP performance data is extracted from the work "In Search of Rate Control Policy for XTP" by Louis Harvey [Harvey], in which prior data analysis was conducted in the Computer Science Laboratories, from Concordia University, Montreal Canada, from a team of research students.

The ESTELLE Specification performance data was acquired as part of this work, using the same facilities as for gathering the SANDIA XTP performance data. The scope of the ESTELLE data is not as complete in diversity of measurement scenario, and is aimed at showing the performance range in which ESTELLE XTP is ranked. The thrust of the ESTELLE specification is to express the XTP protocol using the supplied semantics of ESTELLE and encapsulate reusable parts in a C 'primitive' package. The SANDIA XTP is coded in pure C++, using all of the performance features of the C++ language and adjusting the code to exchange the data through the protocol, as fast and efficiently as possible, this, at the expense of the protocol specification simplicity. Thus, it is expected that the SANDIA XTP results show a superior data transmission throughput.

Both performance analysis benchmarks were run under the supervision and revised with the assistance of Dr. William Atwood.

4.2 Data Performance Result Diagrams



The above data has several window sizes for the Sandia XTP data throughput analysis. The window size is taken as the amount of data sent in one packet. For ESTELLE, and the sample user program, it is 1K, hence the data is reported accordingly. The load was also set at around 1K Bpms.

- **ESTELLE XTP THROUGHPUT ANALYSIS OF A DATA EXCHANGE BETWEEN TWO HOSTS**

Using a supplied sample program to exchange data between two hosts where XTP was installed, and defining one as the sender and the other one as the receiving host the following data rate was achieved:

Lower limit: 1100 Kbits / sec

Higher limit: 1160 Kbits / sec

Average throughput: 1135 Kbits / sec

The transmission unit was 1K (1024). The number of bytes sent/received was 204800 divided into 2000 send counts with transmission duration from 14111 ms to 14827 ms. The experiment was conducted 9 times with communication between the hosts DAFFODIL and FOREST. There was no more than a 1% difference between the throughput of a sender and a receiver, for any given pair of collected data. Following is the table of the load and throughput data as gathered:

143.5	143.8
142.5	141.6
138.1	137.6
141.8	140.4
145.1	144.6
141.9	141.6
142.3	142.0
140.4	140.1
144.1	143.3

Table 1: ESTELLE Data Throughput Benchmarks

4.3 Data Performance Results Conclusion

The performance data reported here reflect on the complexity of the protocol design. As traced in Chapter 2, only two copy operation happens, when the data is entered into the ring buffer and the packet pool shells. Similarly, two data copy operations take place when data is forwarded from user shm, to a data packet shell: One to the context buffer and one to the packet shell. No additional copy operations are performed in getting data from the user, or dropping the data packets on the medium. Copy operations have been reduced to a minimum.

More to the point of structuring the components of the protocol, with data copy (and replication) in mind, we analyzed that only the context primitives perform data copy. The user interface and the network interface do not move data. These procedures reference the data that the data primitives, from the context module, have placed to the correct data structures, ready to be forwarded by pointer reference.

Compared to a similar data analysis from the SANDIA XTP, data is copied several times as it is forwarded through the various parts of the protocol. No channel definitions exist and data is copied to the module data stores that process the sent or received data. This translates into a lower throughput for a given data window size and load.

Can the ESTELLE specification be improved, remains a valid question. Given that data is copied twice in both flow directions suggests that copy operations can still be improved. We consider the improvement of the receive process where by data is copied directly from the packet shell to the user shm. Still, when selective retransmission must be performed, and the NOERR option is used, a context is required to ensure the data transmission integrity. Shared memory suggests that there is a joint managing of the content that is stored; if the user process dies, integrity must be maintained, by XTP, and this validates the existence of the context buffer data store. Similarly for the network interface. Keeping the context buffer and eliminating the shared memory suggests that the external process must have addressability to the data buffered within the protocol. Hence, we converge to the idea that this specification of the XTP protocol has been highly optimized.

A. Appendix

A.1 Glossary

ESTELLE:

An Extended Finite State Machine with Pascal Data Types based Protocol Specification Language. It is defined in ISO 9074: 1989/Amendment 1 as a formal description technique (FDT) for specifying distributed, concurrent information processing systems with a particular application in mind, namely that of communication protocols and services of the layers of Open System Interconnection (OSI) architecture defined by ISO.

XTP:

eXpress Transfer Protocol:

A telecommunication protocol aimed at enabling high performance reliable communication using unicast and multicast.

API:

Application Programming Interface. Sometime used instead of the word Primitives.

SHM:

Shared Memory: Unix shared memory across process boundaries.

UDP:

User Datagram Protocol

Ring Buffer:

Data buffer addressed in a circular mode, to effect data transfer within XTP.

SAP:

Service Access Point

Bpms:

Bytes per millisecond (A load and throughput measure)

A.2 References

[Harvey]: In Search of a Rate Control Policy for XTP: Unicast and Multicast. Louis Harvey

[Lajoie-1]: ESTELLE XTP Specification V3.6 (Unpublished)

[Lajoie-2]: SANDIA XTP Data Movement and Replication (Unpublished)