# A SPECIFICATION OF THE XTP SERVICE IN LOTOS

YAOLIN ZHANG

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

APRIL 1994

ISBN 0-315-97588-1

# Abstract

**A Specification of the XTP Service in LOTOS**

Yaolin Zhang

The Xpress Transfer Protocol (XTP) is a transfer (transport- and network-level) protocol designed to meet the needs of distributed, real-time, and multi-media systems, in both unicast and multicast environments.

A service definition for XTP is proposed, covering both the unicast service and the multicast service, and reflecting the features of XTP.

To provide a precise and unambiguous specification of the service definition, it has been formally specified using LOTOS. The architecture of the specification is presented, along with the approaches used, the data type definitions, and the process structures.

The formal specification demonstrates the need for simpler interactions when closing a connection in XTP, and for better type-definition capabilities in LOTOS.

# Acknowledgements

I would like to take this opportunity to express my sincerest thanks to my supervisor, Dr. J.W. Atwood, for his advice, encouragement, and practical assistance during the course of this research.

I would also like to express my gratitude to Dr. L. Logrippo from theUniversity of Ottawa for sharing their LOTOS utility, ISLA.

I would especially like to thank Ted Ewanchyna for his many helpful comments on this thesis.

Finally, I would also like to thank my wife, Guiling Guo, whose consistent support and encouragement never waivered.

# Contents

# List of Figures

# Chapter 1

# Introduction

To simplify the complexity of data communication systems, the International Organization for Standardization (ISO) proposed a layere darchitectural model called the Reference Model for OpenSystem Interconnection (OSI) [ISO7498]. In the OSI model, communication systems are broken down into several layers. Each of the layers is designed to perform a well-defined function, and to provide services to users at the next higher layer.

Based on this model, each layer is described by two documents, a service definition and a protocol specification. Protocol specifications describe how the transfer occurs, how errors are detected, and how acknowledgments are passed, while service definitions specify what services are provided to the next higher layer and how services are used.

The importance of the protocol specification is widely accepted. However, a well-defined service definition is also necessary for characterizing a layer in communication systems, because it expresses the functions of a complex protocol in a simple way. In addition, the definition of service describes the behaviors of providing the service to the users, that is, the procedures to correctly use the services provided.

Besides the layered architectural model, the OSI system requires unambiguous, precise, implementation independent, and standardized descriptions of services and protocols. An informal description written in natural language can hardly avoid ambiguity, resulting in incompatible implementations. Additionally, it cannot be used to carry out extensive verification and validation at the design level, which is in an early stage of the development cycle.

To deal with this issue, various formal description techniques (FDTs) have been developed since the 1980's. FDTs are languages designed to formally define and analyze the structure and behavior of concurrent systems such as the OSI computer network architecture. LOTOS is one of standardized formal description techniques developed by the ISO. In recent years, some tools to support FDTs have been designed and are now available. These tools make it possible to automatically check and analyze specifications. Therefore, application of FDTs to the design of protocols has been given more and more attention.

In this thesis, we will propose a service definition for the Xpress Transfer Protocol (XTP). XTP is a high performance transfer layer protocol being developed by the XTP Forum to meet the needs of distributed, real time, and multi-media systems. XTP is intended to combine the facilities of the transport and network layers of the ISO reference model, and to provide for both unicasting and multicasting. The defining document for XTP, "XTP Protocol Definition, Revision 3.6"[XTP92], is mainly concerned with defining the protocol. Only an incomplete and implementation-dependent service definition is appended to the document. Our intention is to define a complete XTP service that is independent from implementations.

With this in mind, a formal specification of the XTP service, written in LOTOS, will be presented in this thesis.

The thesis is organized as follows. Chapter two gives a general overview of LOTOS, and introduces the main concepts such as abstract data typing and process algebra. In chapter three we outline the XTP Protocol Definition Revision 3.6. Since the work is concerned with the service interface of XTP, we introduce only the information that should be provided by users and operations that are related to defining the service. Based on the discussion in chapter three, we propose the definition of the XTP service in chapter four. The proposal gives an informal description including definition of service primitives, their parameters, and some typical time sequence diagrams. The LOTOS specification of the XTP service is divided into two parts, the unicast service and multicast service. These are discussed in chapters five and six, respectively. The discussion includes the architecture, approaches used in the specification, data type definition and process structures. Finally, the conclusions of this thesis and some suggestions for future work are presented in chapter seven.

# Chapter 2

# Overview of LOTOS

LOTOS (Language of Temporal Ordering Specification) is one of the standardized formal description techniques (FDTs) developed within the International Organization for Standardization (ISO). It is used to describe open distributed systems, and in particular those related to the Open Systems Interconnection (OSI) computer network architecture, in an abstract way.

The OSI system requires unambiguous, precise, implementation independent, and standardized descriptions of services and protocols. An informal description written in natural language can hardly avoid ambiguity, resulting in incompatible implementations. Additionally, it cannot be used to carry out extensive verification and validation at the design level, which is in an early stage of the development cycle. Therefore LOTOS, as one of formal description techniques for OSI systems, has been developed and became an ISO standard in 1989 [ISO8807].

LOTOS is a mathematically-defined formal description technique. It consists of two components: a process algebra that specifies the behaviour of the system, and abstract data types that deal with the description of data structures and value expressions. In the following sections of this chapter, the concepts of LOTOS used in this thesis will be summarized.

## 2.1 Abstract data types

The representations of values, value expressions and data structures in LOTOS are modelled as "abstract data types (ADTs)". An abstract data type does not describe how data value are actually represented and manipulated in memory, but only defines the essential properties of data and the operations that manipulate the data. Therefore abstract data types are independent of any concrete implementation. Most notions of abstract data types in LOTOS are derived from the algebraic specification language for abstract data types ACT ONE [Ehrig].

### 2.1.1 Basic data typing

The most basic form of data definition in LOTOS consists of defining the names of sorts, operations and possibly, equations. Each sort represents a set of data values of the data type. Every operation defines a total function that describes manipulating procedure on sorts. The declaration of an operation will include its domain, which consists of a list of zero or more sorts, and its range, which consists of exactly one sort. The equations are used to specify relationship between operations. As an example, the following data type definition is extracted from the specification of the XTP service.

```
type Bittype
is    OctDigit, Boolean
sorts bittype
opns
        <>, yes no : -> bittype
        h : bittype -> OctDigit
        Isyes : bittype -> Bool
        _eq_, _ne_: bittype, bittype -> Bool
eqns
        forall n1, n2, : bittype
        ofsort OctDigit
            h (no) = 0;
            h (yes) = 1;
            h (<>) = 2;
        ofsort Bool
            n1 eq n2 = h (n1) eq h (n2);
            n1 ne n2 = not (n1 eq n2);
            Isyes (n1) = h (n1) eq 1
endtype (* Bittype *)
```

### 2.1.2 Structured data types

In order to build new types with existing data types, LOTOS provides the following structuring mechanisms for abstract data type specifications.

1. *Combination* constructs a new data type by combining the types and adding new sorts, operation and equations. The data type Bittype in the above example is formed by combining pre-defined data types OctDigit and Boolean from the LOTOS standard library, and adding some new operations.

2. *Renaming* defines a new data type by changing the names of sorts and operations without any change in semantics. Sample 5.3 in chapter 5 is an example.

3. *Parameterization* allows one to add the formal sorts, operations and equations as formal parameters to the data type specifications. Then, a new data type can be defined by substituting the formal part of the type definition with actual parameters. The substitution is called actualization in LOTOS.

## 2.2 Process algebra

Process algebra is the essential part of LOTOS and is developed from the Calculus of Communicating Systems (CCS)[Milner]. The basic idea is that the system can be specified by defining the temporal relation of the interactions that compose the externally observable behaviour of a system.

In LOTOS a distributed and concurrent system is described by a set of hierarchical processes. A process is an abstract entity that is able to perform both internal, unobservable actions and observable actions (that are interactions with other processes, which form its environment). The LOTOS term for an interaction is event, and it occurs at interaction points called gates. An event is an elementary unit of synchronization among processes. The occurrence of an event at a gate of the process implies that the action is observed by the processes that participate in the synchronization.

To clarify the difference between unobservable actions and observable actions, a process can be seen as a black box with some gates for interaction with other processes. The mechanisms inside the box are not observable. Therefore, all actions occurring inside a process are unobservable.

In LOTOS interactions, i.e. events, have the following features:

- Atomic: Events occur instantaneously, without consuming time. So interactions are not able to be interrupted, and processes cannot carry out more than one interaction at a time.
- Synchronous: Processes that interact in an event participate in its execution at the same moment in time. In other words, it is necessary that all processes that take part in th^e event must be ready for an event to occur.
- Multi-way: Interaction may take place between two or more processes.

### 2.2.1 Process definition

In LOTOS, the process definition consists of two parts, static and dynamic. The static part includes declaration of its identifiers, formal gate parameters, formal value parameters and functionality. The following is a typical structure of a process definition.

**process** *process-identifier* [*gate-list*] (*parameter-list*): *functionality* :=

    *dynamic part*

**where**

    *local type definition*

*process definitions*

**endproc**

Formal gate parameters are used to define the interaction ports of a process that may interact with other processes, and will be substituted by actual ones when the process is instantiated. As with functions or procedures in other programming languages, a process in LOTOS may have a list of formal value parameters. When the process is instantiated these will be replaced by actual values. Each process has a functionality parameter to indicate its result. The three possible formats of this parameter are:

1. **noexit**. The process cannot terminate successfully.
2. **exit**. The process may terminate successfully.
3. **exit**$[t_1, \cdots, t_n]$, where $t_1, \cdots, t_n$ is a sort list. The process may terminate successfully, and offer a list of values that belong to the sort in the sort list.

The concept of successful termination will be discussed in detail later.

In the dynamic part of the definition, the dynamic behaviour of a process is defined by "behaviour expressions", which describe the temporal relation between observable actions. A behaviour expression is constructed by applying an operator to other behaviour expressions. For convenience, given a behaviour expression $B$, it is also called a process, even when $B$ is not explicitly associated with a process name.

### 2.2.2 Operators and syntax of behaviour expressions

In this section, operators used in LOTOS and forms of basic behaviour expression will be examined. As a convention we will use boldface for reserved LOTOS keywords, and italics for nonterminal symbols.

1. name: inaction

   denotation: **stop**

   syntax: **stop**

   > **stop** represents a completely inactive process. That is, it cannot execute internal actions, nor it can perform interactions.

2. name: action prefix

   denotation: ;

   syntax: $a$;$B$

   > The behavior of $a$;$B$ will be the action $a$ followed by the behaviour of $B$, where $a$ stands for either an internal action or event, and $B$ for any behaviour expression.

3. name: guarding

   denotation: [*guard*] ->

   syntax: [*guard*] -> $B$

   > The behavior of [*guard*] -> $B$ will be the conditional execution of the behaviour $B$, where *guard* is a condition to be evaluated. If the guard holds, then the behaviour $B$ is possible, otherwise the behaviour expression is equivalent to **stop**.

4. name: choice

   denotation: []

   syntax: $B_1$[]$B_2$

   > The behavior of $B_1$[]$B_2$ will be the behaviour of either $B_1$ or $B_2$, but not both, where $B_1$ and $B_2$ are any behaviour expressions.

5. name: parallel composition

   denotation: $|[g_1, g_2, \cdots, g_n]|$

   syntax: $B_1|[g_1, g_2, \cdots, g_n]|B_2$

   The behavior of $B_1|[g_1, g_2, \cdots, g_n]|B_2$ will be a composition in which $B_1$ and $B_2$ must synchronize with respect to the gates belonging to the gate set $[g_1, g_2, \cdots, g_n]$.

6. name: interleaving

   denotation: $|||$

   syntax: $B_1|||B_2$

   Interleaving is a special case of parallel composition, in which the gate set is empty. The expression $B_1|||B_2$ expresses any interleaving of the actions of $B_1$ with the actions of $B_2$. For example, if $B_1$ is ready for an action $a_1$ and $B_2$ for an action $a_2$, then both action orderings ($a_1$ before $a_2$, $a_2$ before $a_1$) are possible. Also, after an action (say $a_1$ or $a_2$), $B_1|||B_2$ transforms into an expression which still involves the operator '$|||$'.

7. name: synchronization

   denotation: $||$

   syntax: $B_1||B_2$

   Synchronization is another special case of parallel composition, in which the gate set includes all user-definable gates. $B_1||B_2$ expresses that $B_1$ and $B_2$ are compelled to proceed in full synchronization except for possible internal actions.

8. name: hiding

   denotation: **hide**

syntax: **hide** $g_1, g_2, \cdots, g_n$ **in** $B$

Hiding transforms some observable actions of a process into unobservable ones. **hide** $g_1, g_2, \cdots, g_n$ **in** $B$ means that any action occurring at a gate in the set of hidden gates $g_1, g_2, \cdots, g_n$ is transformed into unobservable action.

9. name: process instantiation

   syntax: $P\ [g_1, g_2, \cdots, g_n\ ]$ or

   $P\ [g_1, g_2, \cdots, g_n\ ](p_1, p_2, \cdots, p_m)$

   where $P$ is a process identifier, $[g_1, g_2, \cdots, g_n\ ]$ is a list of actual gates and $(p_1, p_2, \cdots, p_m)$ is a list of actual parameters. The process instantiation is analogous to the invocation of a function or procedure in standard programming languages. It is useful for the specification of recursive behaviours.

10. name: successful termination

    denotation: **exit**

    syntax: **exit** or

    **exit** $(v_1, \cdots, v_n)$

    where $v_1, \cdots, v_n$ is a list of values to be offered. The behaviour expression **exit** is used to specify that a process terminates successfully.

    In order to distinguish between successful termination and unsuccessful termination, we need to introduce a special gate called $\delta$ and the successful termination action $\delta$. Both of them are unique and different from user-defined gates and actions. They cannot be used in a specification. Only **exit** performs the action $\delta$, and then it transforms into **stop**. Moreover, **exit** requires synchronization at gate $\delta$ to succeed, which is used in the sequential composition of behaviour as shown below.

11. name: enabling

    denotation: $\gg$

    syntax: $B_1 \gg B_2$ or

    $B_1 \gg$ **accept** $x_1 : t_1, \cdots, x_n : t_n$ **in** $B_2$

    where $x_1, \cdots, x_n$ are the variables used in $B_2$ and $t_1, \cdots, t_n$ are the sorts related to the variables. The behavior of $B_1 \gg B_2$ will be that if $B_1$ terminates successfully, then the execution of $B_2$ is enabled. Additionally, the second expression indicates that $B_2$ will accept the values offered by $B_1$.

12. name: disabling

    denotation: [>

    syntax: $B_1[> B_2$

    Disabling resembles an irreversible interrupt. The behavior of $B_1[> B_2$ will be one of three possible cases. One case is that the behaviour $B_1$ is interrupted by the first action of $B_2$, control is transferred from $B_1$ to $B_2$, and cannot go back to $B_1$. The second case is that $B_1$ performs a successful termination action. In this case, $B_2$ is terminated at the same time. The last case is that $B_1$ performs an action that is not a successful termination. $B_2$ is still expected to interrupt $B_1$'s successor.

### 2.2.3 Operational Semantics of expressions

In the above section, we discussed the syntax of behaviour expressions in LOTOS, and informally interpreted their semantics. More precise semantics will now be defined in terms of a formal system of inference rules.

**Definition:** Let $B_1$ and $B_2$ be behaviour expressions, $a$ be an action. A *labelled transition* is a triple,

$$B_1 - a \longrightarrow B_2$$

in which $a$ is called a label.

Labelled transitions are used to define the following inference rules of the operational semantics of LOTOS. In these inference rules, the interpretation of a labelled transition of the form $B_1 - a \longrightarrow B_2$ is that $B_1$ can perform action $a$ and transform into $B_2$ after the occurrence of $a$. For the given behaviour expression, the actions that the expression may perform can be systematically derived from its structure by using the inference rules.

For defining the semantics, the following notations will be used.

| | |
|---|---|
| $G$ | denotes the set of user-definable gates; |
| $g, g_1, \cdots, g_n$ | range over G; |
| $\mathbf{i}$ | denotes the unobservable action; |
| $A$ | denotes the set $G \cup \{\mathbf{i}\}$ of user-definable actions; |
| $a$ | range over A; |
| $\delta$ | denotes the successful termination action; |
| $E$ | denotes the value expression as a guard; |
| $G^+$ | denotes the set $G \cup \{\delta\}$ of observable actions; |
| $g^+$ | range over $G^+$; |
| $A^+$ | denotes the set $A \cup \{\delta\}$ of actions; |
| $a^+$ | range over $A^+$. |

**Operational Semantics:**

1. inaction

   **stop** has no associated inference rules.

2. Action Prefix

$$a; B - a \longrightarrow B \tag{2.1}$$

3. Guarding

$$\frac{E = \mathbf{true}}{[E]\text{->}\ B \quad \longrightarrow B} \tag{2.2}$$

$$\frac{E = \mathbf{false}}{[E]\text{->}\ B \quad \longrightarrow \mathbf{stop}} \tag{2.3}$$

4. Choice

$$\frac{B_1 - a^+ \longrightarrow B_1'}{B_1[]B_2 - a^+ \longrightarrow B_1'} \tag{2.4}$$

$$\frac{B_2 - a^+ \longrightarrow B_2'}{B_1[]B_2 - a^+ \longrightarrow B_2'} \tag{2.5}$$

5. Parallel composition

$$\frac{B_1 - a \longrightarrow B_1',\ a \notin \{g_1,\ \cdots,\ g_n\}}{B_1|[g_1,\ \cdots,\ g_n]|B_2 - a \longrightarrow B_1'|[g_1,\ \cdots,\ g_n]|B_2} \tag{2.6}$$

$$\frac{B_2 - a \longrightarrow B_2',\ a \notin \{g_1,\ \cdots,\ g_n\}}{B_1|[g_1,\ \cdots,\ g_n]|B_2 - a \longrightarrow B_1|[g_1,\ \cdots,\ g_n]|B_2'} \tag{2.7}$$

$$\frac{B_1 - g^+ \longrightarrow B_1',\ B_2 - g^+ \longrightarrow B_2',\ g^+ \in \{g_1,\ \cdots,\ g_n\} \cup \{\delta\}}{B_1|[g_1,\ \cdots,\ g_n]|B_2 - g^+ \longrightarrow B_1'|[g_1,\ \cdots,\ g_n]|B_2'} \tag{2.8}$$

6. Interleaving

$$\frac{B_1 - a^+ \longrightarrow B_1'}{B_1\ |||\ B_2 - a^+ \longrightarrow B_1'\ |||\ B_2} \tag{2.9}$$

$$\frac{B_2 - a^+ \longrightarrow B_2'}{B_1\ |||\ B_2 - a^+ \longrightarrow B_1\ |||\ B_2'} \tag{2.10}$$

$$\frac{B_1 - \delta \longrightarrow B_1',\ B_2 - \delta \longrightarrow B_2'}{B_1\ |||\ B_2 - \delta \longrightarrow B_1'\ |||\ B_2'} \tag{2.11}$$

7. Synchronization

$$\frac{B_1 - \mathbf{i} \longrightarrow B_1'}{B_1 \parallel B_2 - \mathbf{i} \longrightarrow B_1' \parallel B_2} \tag{2.12}$$

$$\frac{B_2 - \mathbf{i} \longrightarrow B_2'}{B_1 \parallel B_2 - \mathbf{i} \longrightarrow B_1 \parallel B_2'} \tag{2.13}$$

$$\frac{B_1 - g^+ \longrightarrow B_1',\ B_2 - g^+ \longrightarrow B_2'}{B_1 \parallel B_2 - g^+ \longrightarrow B_1' \parallel B_2'} \tag{2.14}$$

8. Hiding

$$\frac{B - a^+ \longrightarrow B',\ a^+ \notin \{g_1,\ \cdots,\ g_n\}}{\textbf{hide}\ g_1,\ \cdots,\ g_n\ \textbf{in}\ B - a^+ \longrightarrow \textbf{hide}\ g_1,\ \cdots,\ g_n\ \textbf{in}\ B'} \tag{2.15}$$

$$\frac{B - g \longrightarrow B',\ g \in \{g_1,\ \cdots,\ g_n\}}{\textbf{hide}\ g_1,\ \cdots,\ g_n\ \textbf{in}\ B - i \longrightarrow \textbf{hide}\ g_1,\ \cdots,\ g_n\ \textbf{in}\ B'} \tag{2.16}$$

9. Process Instantiation

If '$P[g_1', \cdots, g_n'] := B_p$' is a process definition with the body $B_p$, and $P[g_1, \cdots, g_n]$ is an instantiation by replacing formal gates $g_i'$ with actual gates $g_i (i = 1, \cdots, n)$, then

$$\frac{B_p[g_1/g_1',\ \cdots,\ g_n/g_n'] - a^+ \longrightarrow B'}{P[g_1, \cdots, g_n] - a^+ \longrightarrow B'} \tag{2.17}$$

where $B_p[g_1/g_1',\ \cdots,\ g_n/g_n']$ is the body after replacing gates.

10. Successful Termination

$$\textbf{exit} - \delta \longrightarrow \textbf{stop} \tag{2.18}$$

11. Sequential Composition

$$\frac{B_1 - a \longrightarrow B_1'}{B_1 \gg B_2 - a \longrightarrow B_1' \gg B_2} \tag{2.19}$$

$$\frac{B_1 - \delta \longrightarrow B_1'}{B_1 \gg B_2 - \mathbf{i} \longrightarrow B_2} \tag{2.20}$$

12. Disabling

$$\frac{B_2 - a^+ \longrightarrow B_2'}{B_1[> B_2 - a^+ \longrightarrow B_2'} \tag{2.21}$$

$$\frac{B_1 - \delta \longrightarrow B_1'}{B_1[> B_2 - \delta \longrightarrow B_1'} \tag{2.22}$$

$$\frac{B_1 - a \longrightarrow B_1'}{B_1[> B_2 - a \longrightarrow B_1'[> B_2} \tag{2.23}$$

## 2.3 Interprocess communication

From the discussion in the last section, we know that the value passing between two processes may occur when a process instantiation or enabling operation is performed. However, the exchange of data is mainly achieved when an interaction among processes takes place at a gate. LOTOS therefore allows interactions (i.e., events) at a gate to be associated with variable declarations ('input') and value declaration ('output'). In LOTOS, input and output of data are represented by '?' and '!'. For example,

$g?x : t$ means that the process is prepared to accept a value of sort $t$ at gate $g$; after the event has occurred the received value is stored in the variable $x$.

$g!e$ means that the process is prepared to offer the value of $e$ at gate $g$, where $e$ is an arbitrary value expression.

LOTOS supports three types of interaction between two processes. We illustrate them by simple examples as follows.

1. value passing, $g!e$ and $g?x : t$

   If $value(e) \in$ sort $t$ then $x = value(e)$ after the synchronization.

2. value matching, $g!e_1$ and $g!e_2$

   If $value(e_1) = value(e_2)$ then the synchronization is achieved.

3. value negotiation, $g?x : t_1$ and $g?y : t_2$

   If $t_1 = t_2$ then $x = y = v$ after the synchronization, where $v \in$ sort $t$.

# Chapter 3

# An introduction to XTP

## 3.1 General

The Xpress Transfer Protocol (XTP) is a high performance transfer layer protocol being developed by the XTP Forum. Unlike traditional transport layer protocols, such as DoD's Transmission Control Protocol (TCP) and ISO's Transport Protocol (TP), XTP combines the functionalities of both the network and transport layers of the ISO OSI model into a single layer.

The design goals of XTP are to meet the needs of distributed, real time, and multi-media systems. In order to support all of these kinds of systems, XTP provides the following functionalities:

- implicit connection setup
- simple frame format
- message boundary preservation

- compatibility with multiple addressing schemes
- flow/rate/error control
- bulk transport service
- real-time reliable datagram service
- traditional stream service
- both unicast and multicast mechanisms.

XTP is a connection-oriented protocol, which sets up full-duplex virtual circuit connection between end systems. In XTP, a connection is called an association, and considered as a pair of active contexts, with one context at each end system. The context denotes state information representing one instance of an active communication between two (or more for multicast mode) XTP endpoints. It is important to note that a context must be created, or instantiated, before sending and receiving XTP packets. Once an association is shut down, the related contexts revert to the "null" state.

In the following sections of this chapter, we will briefly introduce the packet formats and operations of XTP based on Revision 3.6 of the Protocol Definition. Because the work reported in this thesis is confined to the service interface of XTP, we are only going to discuss the information that should be provided by users and operations that are related to defining the service. The details of some issues, such as flow/rate/error control, will be ignored. The complete definition of the protocol can be found in [XTP92].

## 3.2 XTP packet formats

XTP contains two general types of packets: control packets and information packets. Control packets are used to communicate the state of an XTP machine between protocol entities, while information packets contain transport layer messages and user data within an information segment. The information packets that can carry user data are also referred to as data packets.

As shown in Figure 3.1, an XTP packet consists of three parts, a 40-byte header, a 4-byte trailer, and a middle segment, which is of variable length and can be a control segment or an information segment depending on the type of packet. The XTP trailer only contains a checksum of the middle segment.

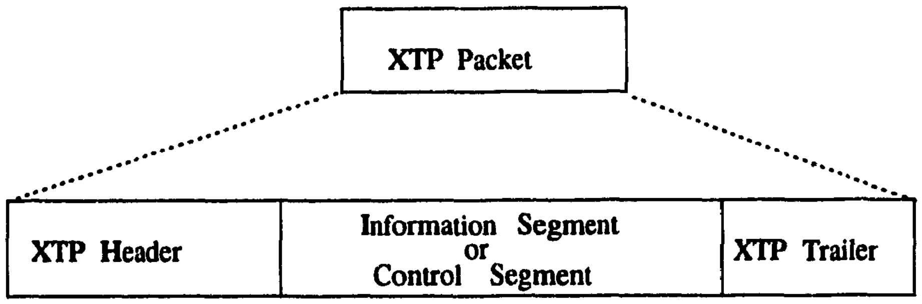

Figure 3.1: XTP packet: three major segments

### 3.2.1 XTP header

An XTP header has ten fields, namely *route*, *ttl* (time to live), *cmd* (command), *sync*, *key*, *seq* (sequence), *dseq* (delivery sequence), *sort*, *dlen* (data length), and *hcheck* (header checksum).

The field *cmd* contains the following information:

1. the type of XTP packet. The *ptype* field in the field *cmd* sub-classifies the control packet and information packet as nine types, that is,

   - *DATA*: user data packet.
   - *FIRST*: initial packet of an association, it may contain user data.
   - *PATH*: reconnection packet, which is used to change an existing association, or to join an in-progress multicast conversation.
   - *DIAG*: diagnostic packet that contains peer-to-peer diagnostic messages.
   - *MAINT*: network maintenance packet.
   - *MGMT*: network management packet.
   - *ROUTE*: route management packet used to exchange information with XTP switches and routers.
   - *CNTL*: control packet.
   - *RCNTL*: intermediate system control packet.

   The last two types of packets belong to control packets that contain a control segment. The others are information packets because they each contain an information segment.

2. the offset used to identify the beginning of actual data in an information segment.

3. the XTP version.

4. the service options used to select XTP operating mode, mechanisms, and operations. XTP provides two groups of options. The first group is used to specify

operating mode and mechanisms. Once the options of this group are selected, they will never be changed for the lifetime of a connection. The second group of options, called control flags in the previous version(3.5) of XTP, are used to indicate operations. They could be different on every packet. All the options are defined as follows.

The first group:

- NOCHECK: disable checksum function
- NOERR: disable error control
- MULTI: multicast mode
- RES: enable reservation mode
- SORT: enable sorting
- NOFLOW: disable flow control
- FASTNAK: enable aggressive error control

The second group:

- SREQ: status requested
- DREQ: deliver status requested
- RCLOSE: reader closed, indicates that no further incoming packets are expected
- WCLOSE: writer closed, indicates that all user data have been transmitted
- EOM: end of message
- END: end of association, indicates that the packet is the last one for a connection
- BTAG: beginning data tag

### 3.2.2 Control and information segment

The control segments are used to exchange state information of a context between endpoints. A control segment contains values of the flow, rate and error control parameters. It also contains information used to synchronize the sender and receiver. These are represented as fields: *rate, burst, echo, time, techo, xkey, xroute, alloc, rseq, nspan*, and *spans.*

As mentioned in section 3.2 above, information segments are used in the following of types of packets: *FIRST, DATA, PATH, DIAG, MAINT, MGMT*, and *ROUTE*. The first two types are also referred to as data packets because these can contain user data. The others contain network layer messages only, and will be omitted here.

The *FIRST* packet is used for initiating an association. Its information segment contains an address segment and an optional data segment. In the address segment the following fields are defined.

1. *alen* indicates the address segment length.
2. *service type* is used to indicate the type of traffic expected on the association. The possible services are:
   - Connection
   - Transaction
   - Unacknowledged datagram
   - Acknowledged datagram
   - Isochronous stream
   - Bulk data

This information is provided by the user and transmitted without change to the destination host. The receiving context will use it to select an interface for the duration of the association.

3. *aformat* (address format) and *address* identify the network address syntax and addresses of two endpoints, i.e., source and destination for supporting multiple addressing schemes.

4. *id* contains the MAC address of the originator of the *FIRST* packet.

5. *rate_req*, *burst_req* and *maxdata* are used to indicate the sender's expected data flow rate, burst size, and length of maximum information segment, respectively.

The information segment of *DATA* packets consists only of a variable-length data segment. If *BTAG* is set in the header, the data segment contains an user-defined *btag* field for identifying message boundaries as a part of user data.

As a summary, the structures of XTP packets are shown in Fig. 3.2.

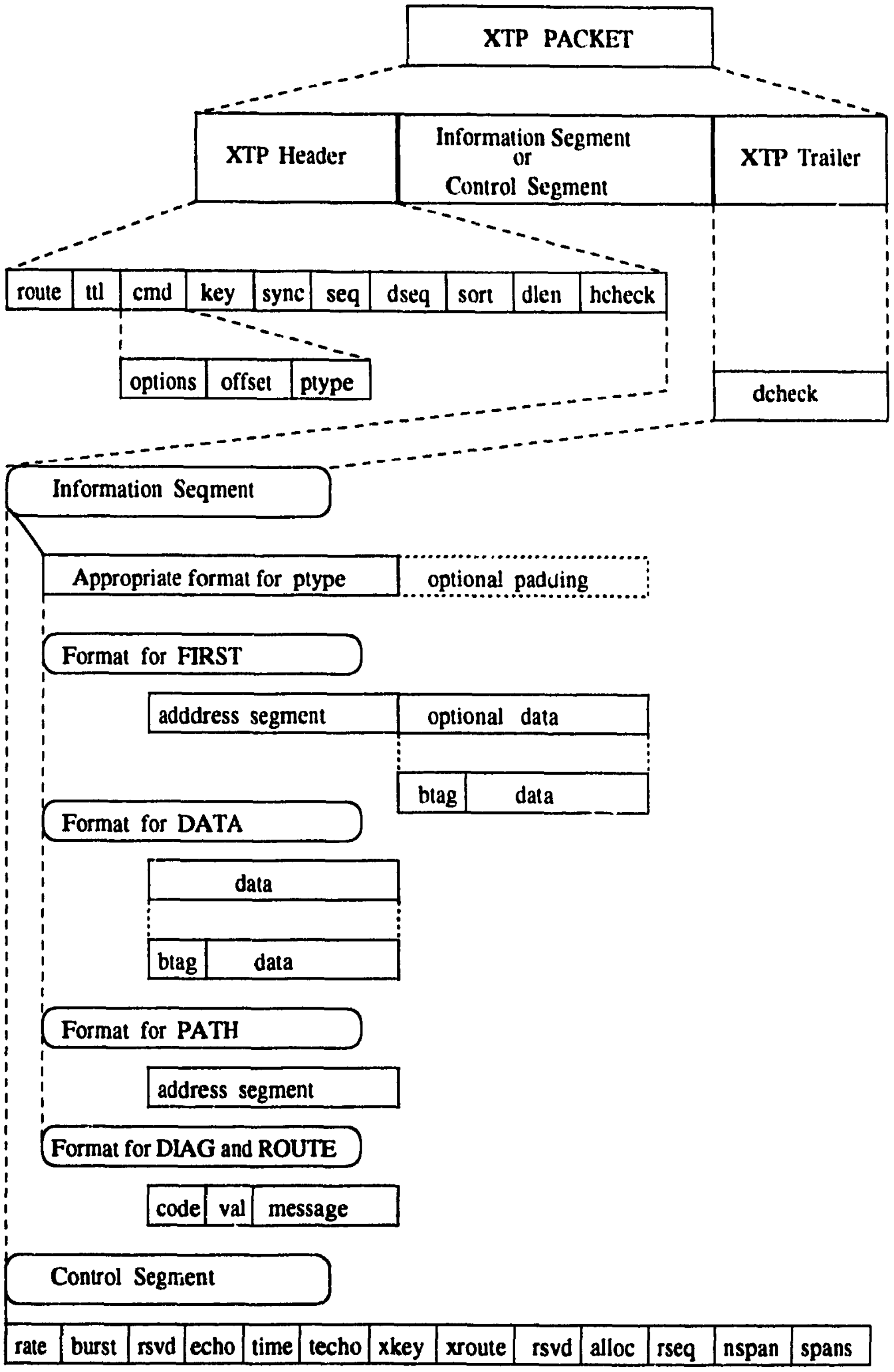

Figure 3.2: XTP packet structures

## 3.3 XTP operation

In this section we will discuss procedures for managing connections and exchanging packets between endpoints. The lifetime of a connection in protocols such as TCP and TP4 can be divided into three phases: connection phase, data transfer phase, and termination phase. Although the boundaries between these phases in XTP are not as clear as in the protocols mentioned above, we will still introduce the operations of XTP in this way.

### 3.3.1 Connection phase

In XTP, transmitting a *FIRST* packet starts to establish an association between contexts on the source and destination endpoints. For convenience, the source side sending a *FIRST* packet is called an initiator, and the destination side receiving the *FIRST* packet is called a responder.

Before a *FIRST* packet is received, a context in *listen* state must be waiting for it on the responder. Otherwise the attempt to initiate an association will be rejected. Upon receipt of a *FIRST* packet with matching expected options, the listening context is transformed into the *active* state, and an association is established. Then the responder may respond to the initiator by sending a *CNTL* or *DATA* packet.

Figure 3.3 depicts two styles of successful association establishment. In the examples of Figure 3.3, Host A is the initiator and Host B is the responder in listen state.

Figure 3.3 (a) shows host A sending a *FIRST* packet with the *SREQ* bit requesting an immediate acknowledgment, and waiting for a positive response. After the *FIRST* packet arrives at host B, a *CNTL* packet is sent as a response. This style, as in TCP and TP4, is called explicit connection set up.

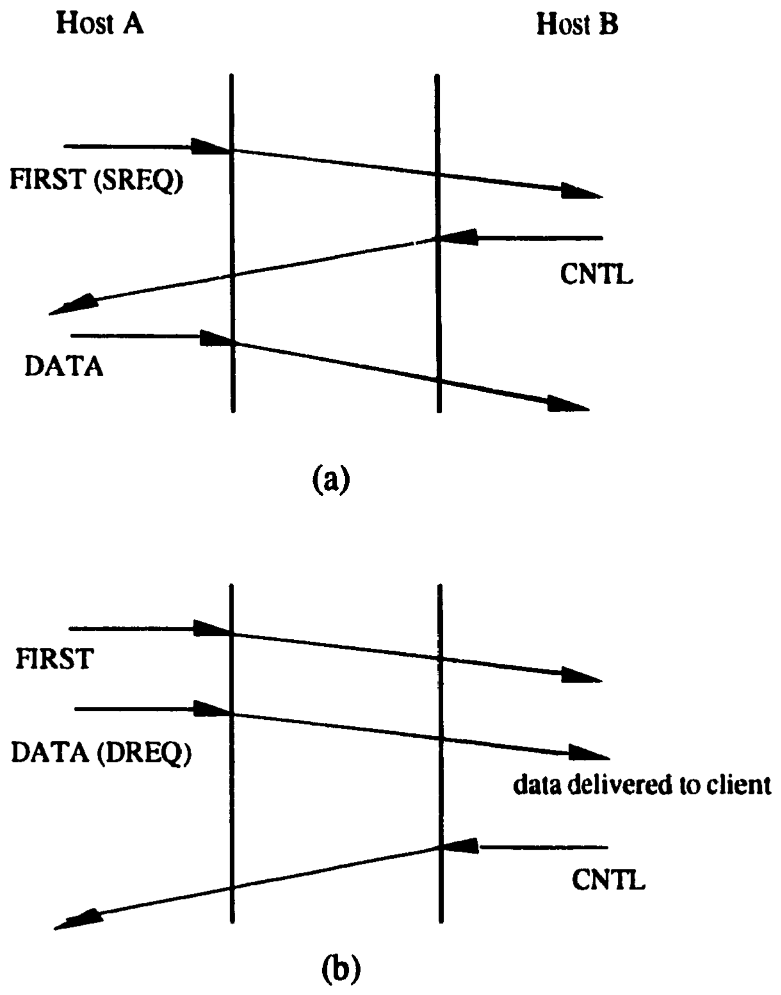

Figure 3.3: Connection handshakes

In order to eliminate the latency of a preliminary end-to-end message exchange, XTP also provides another style, that is, implicit connection setup, as shown in Figure 3.3 (b). In this style, host A may transmit data immediately following the *FIRST* packet, without waiting for any response from host B. Host B can also start to transmit its outgoing data stream as soon as its context becomes active. This is why the boundary between phases is said to be unclear.

### 3.3.2 Data transfer phase

After an association is set up, it provides for two separate data streams, one in each direction. In fact, sending a *FIRST* packet implies that the data transfer phase starts at the initiator side. We call an association at this point a half open association, because the responder can not transmit data until it receives a *FIRST* packet. When an association becomes fully open, both endpoints can perform sending and receiving behaviors.

To send a user message, the XTP sender side fragments the message into one or more *DATA* packets and sends to another endpoint. After getting the acknowledgment that indicates that the data have been delivered to the user space at the receiver side, XTP will inform the user that the user buffers may be released.

For receiving messages, the XTP receiver will check the received *DATA* packets, and deliver the data to user buffers provided by the receiving client. Then, it may respond with an acknowledgment (a *CNTL* packet) or an outgoing *DATA* packet in the reverse direction. When the receiver receives a *CNTL* packet with the SREQ bit on, it immediately sends back a *CNTL* packet to indicate its current receiving status.

In addition to the basic operations discussed above, XTP provides mechanisms for dealing with flow, rate and error control. The details of these issues will not be discussed here.

### 3.3.3 Termination phase

An association between a pair of contexts, say A and B, includes two simplex streams: A-to-B and B-to-A. Each context is the sender for one of the data streams and the receiver for the other one. For terminating an association, each data stream must be closed, and both contexts must be released. The close procedure involves a handshake with the following close control bits in the packet header:

- *WCLOSE*: a data sender wants to close its output stream.
- *RCLOSE*: a data receiver has closed its input data stream and will accept no further data.
- *END*: the context is being released.

Unlike the procedure for connection setup, XTP allows either endpoint to initiate the closure with *WCLOSE*, *RCLOSE*, both *WCLOSE* and *RCLOSE* or all of the close control bits. The other end has several possible ways to respond.

The closure procedures of a simplex stream can be divided into two categories, graceful and forced, by the effect on transmitting user data. A graceful close of a data stream means that the sender initiates the close by setting the *WCLOSE* bit in an output packet, and then the receiver responds to it with the *RCLOSE* bit after all received data have been delivered to user space. Before responding to the sender, the receiver may ask for retransmissions. A forced close of a data stream results when the receiver's close request with *RCLOSE* or/and *END* precedes the sender's *WCLOSE*. In this case, data transmission is forced to terminate. A close procedure of an association can also be indicated by the number of handshakes between the two endpoints. Figure 3.4 shows a typical 4-way handshake for a graceful close of an association.

XTP provides flexible ways to terminate an association. As shown in Figure 3.5, a 3-way handshake also achieves two graceful closes. In this example, the endpoint

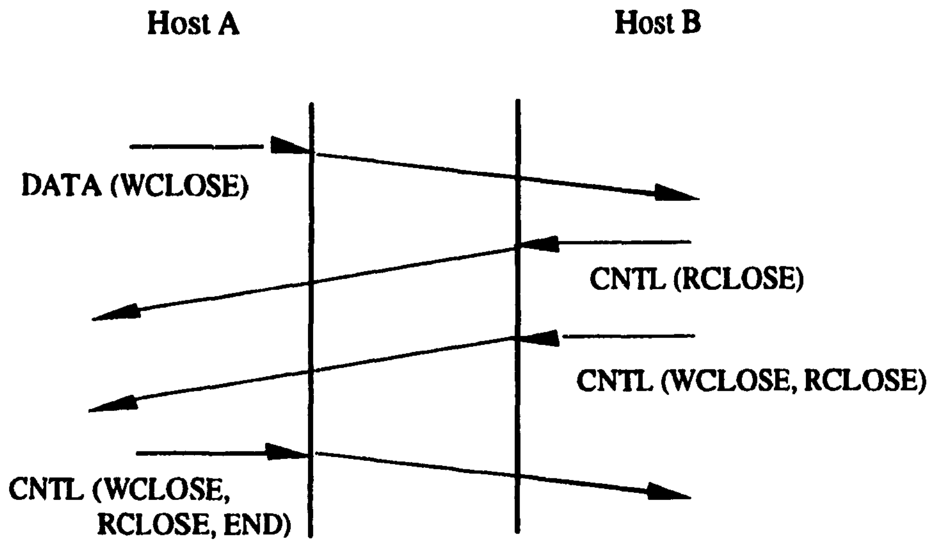

Figure 3.4: A 4-way handshake closure

B responds with *RCLOSE* and initiates another graceful close with *WCLOSE* at the same time.

The number of handshakes can be further reduced as shown in Figure 3.6 and Figure 3.7. Figure 3.6 illustrates a 2-way closure called a foreshortened close procedure, in which A initiates a graceful close for the A-to-B data stream and a forced close for the B-to-A data stream. B then responds to A with *WCLOSE, RCLOSE* and *END.*

Figure 3.7 also shows an example of the unacknowledged datagram style of operations. The endpoint A sends a *FIRST* packet that includes *WCLOSE, RCLOSE* and *END.* The endpoint B received both the *FIRST* packet for setting up an association, and control bits initiating closure of two data streams and indicating that A's context has been released. The endpoint B will therefore not need to respond to A with any further message, and releases its context immediately. This example also shows that the connection phase, data transfer phase and termination phase can be combined in XTP.

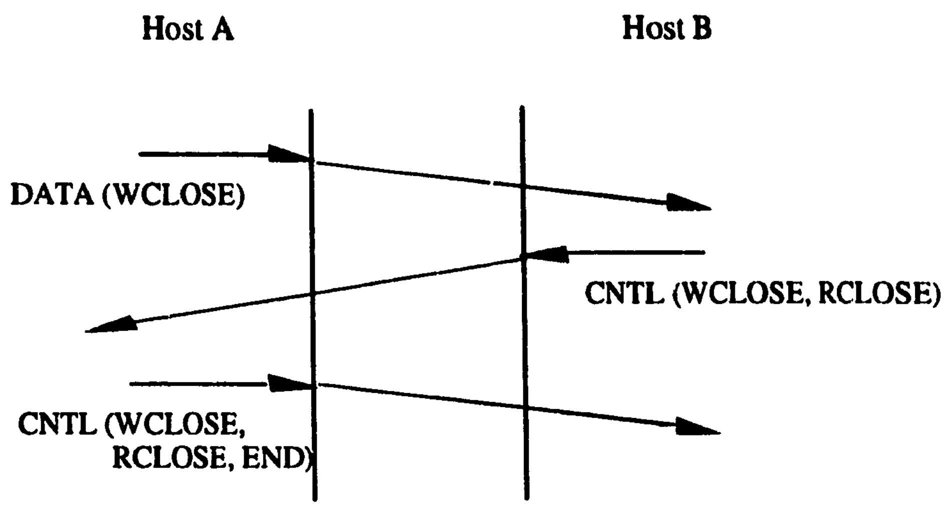

Figure 3.5: A 3-way handshake closure

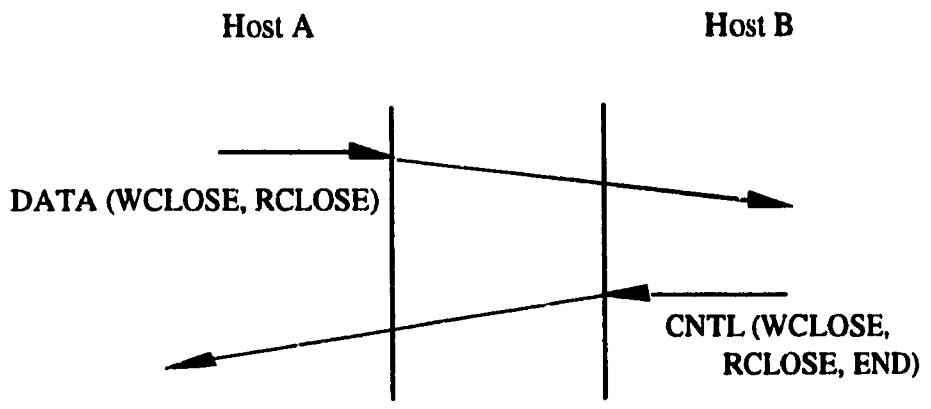

Figure 3.6: A foreshortened close

In the real world, two interacting XTP entities act concurrently. Besides the normal closures discussed above, several other cases may take place. As discussed in [Atwood92], both endpoints may issue a *WCLOSE* or/and *RCLOSE* simultaneously, which leads to a close procedure called a crossed close. Figure 3.8 shows a typical crossed close, in which each side initiates a graceful close at the same time, responds to its partner, and then releases its own context simultaneously. In this 6-way close procedure, the last one or two messages may be ignored, because the destination context may have been released.

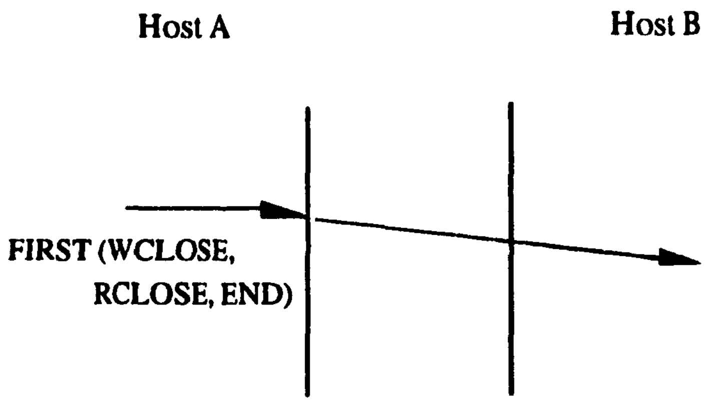

Figure 3.7: A foreshortened close

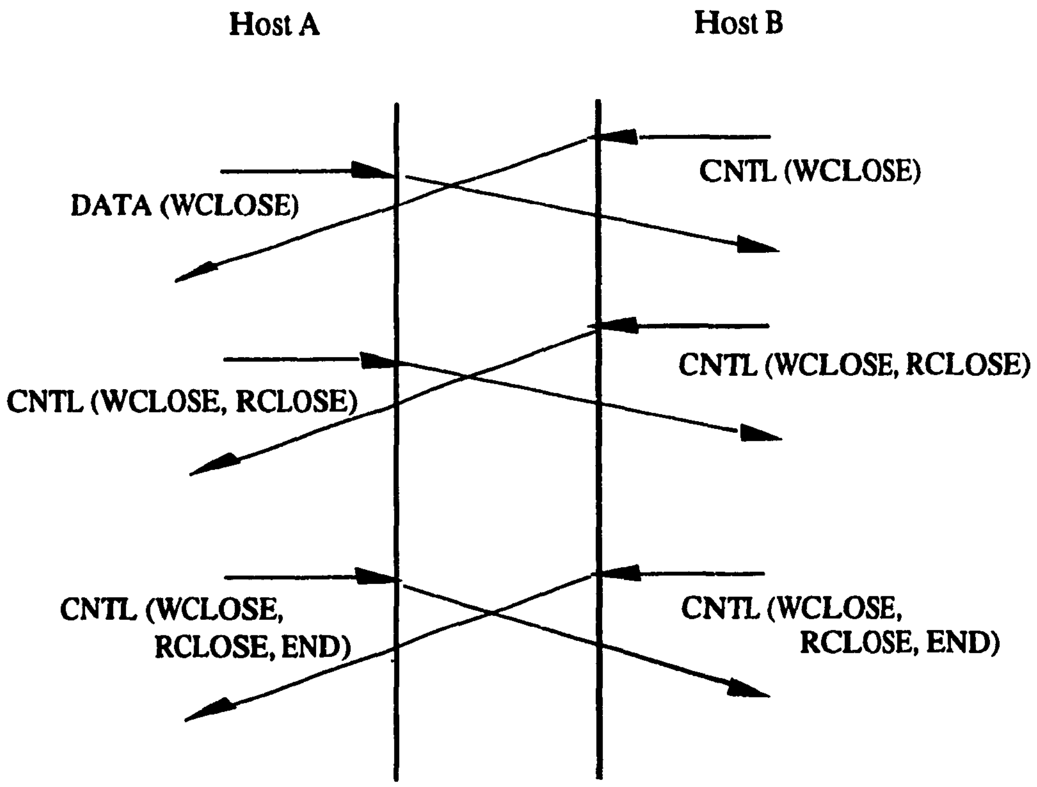

Figure 3.8: A 6-way crossed close

### 3.3.4 Multicast mode

Multicast is a new functional capability for a high performance transport layer. It is designed for transferring messages from one sender to multiple receivers simultaneously. This functionality is not provided in many existing standard protocols. XTP supports a multicast mode, which achieves a multicasting service.

In XTP, the format of packets in multicast mode is same as in unicast mode except that the *MULTI* flag in the header must be set and *DREQ* is never used. Due to there being more than one receiver in multicast mode, some issues, such as addressing and operations related to setting up and managing associations, differ from the unicast mode discussed in the previous section.

In the multicast mode, the sender's context will associate with multiple receiver contexts. Each association between the sender and one receiver is only a simplex stream, that is, sender-to-receiver. This means that receivers never send data to the sender.

#### Setting up a multicast association

For the multicasting sender, the operation of setting up an association is similar to unicast mode. The sender issues a *FIRST* packet, which utilizes group addresses rather than point-to-point address, and then may send *DATA* packets.

In the other side, there may be many receiving contexts, which are in the listening state. For any receiver belonging to a multicast group, its context becomes active after receiving the *FIRST* packet. Only one association is set up. In the unicast mode, a receiver's context must be listening state before receiving a *FIRST* packet, otherwise the association cannot be established. However, in multicast mode, it is possible that a receiver who did not catch the *FIRST* packet can join an in-progress association by sending a *PATH* packet.

## Data broadcasting

In multicast mode, because the receivers are not allowed to send any data back to the sender, it seems that mechanisms in the data transfer phase can be simplified. The multicast sender obeys the same rules for flow control and rate control as a unicast sender. However, the error control is more complex due to the fact that a received *DSEQ* in multicast mode only indicates correct reception by one receiver. This issue also relates to when the sender's data buffers can be released.

XTP provides two classes of multicast services: unacknowledged and acknowledged. For unacknowledged multicasting, the sender sets the *NOERR* bit, thus its buffers can be released as soon as the data have been transmitted. Acknowledged multicasting requires special consideration, because partial positive acknowledgements do not mean that all expected receivers correctly received the data. XTP proposes a "bucket algorithm" to deal with this issue. However, [Santoso] pointed out that the bucket algorithm is inefficient and introduced an enhanced bucket mechanism. We do not discuss this error control algorithm in detail here because we are only concerned with the condition of releasing a user's buffer. In the specification of multicast service, we adopt a "time-stamp" mechanism to specify acknowledged multicast service. This will be discussed in Chapter 6.

In addition to the complex error control mechanism, XTP allows the receiver to drop-out, and rejoin an in-progress multicasting association with a special *PATH* packet.

## Termination of multicast

A multicast conversation is supported by multiple multicast associations. Terminating multicast implies closing all of them. Each receiver can only close or drop-out of its own association by using control bits *RCLOSE* and *END*, as for a unicast receiver. Note that a multicast association is simplex, thus *RCLOSE* will lead to completely closing the association and control bit *WCLOSE* used by receivers is meaningless.

Multicasting cannot be terminated by one receiver, even all receivers, because some receivers may rejoin the conversation. Only the sender can terminate multicasting by issuing *WCLOSE* or/and *END*, which will shutdown handshake with all receivers. Certainly, *RCLOSE* is not necessary for the sender.

# Chapter 4

# Informal description of XTP service

## 4.1 Concept of service

In the late 1970's, the ISO proposed a layered architectural model [ISO7498] that is called the Reference Model for Open System Interconnection (OSI). In the OSI model, communication systems are broken down into a number of layers, each of which performs a well-defined function. Associated with a layer, say layer *N*, are two interfaces. As shown in Figure 4.1, layer *N* provides at the interface to layer *N+1* a well-defined set of services. Layer *N*'s protocol entities establishing the protocol of the layer *N* are in turn built using the services provided through interfaces with layer $N-1$.

In order to characterize each layer and the linkages between adjacent layers of the OSI model, some new concepts and terms were introduced in [ISO82]. Each

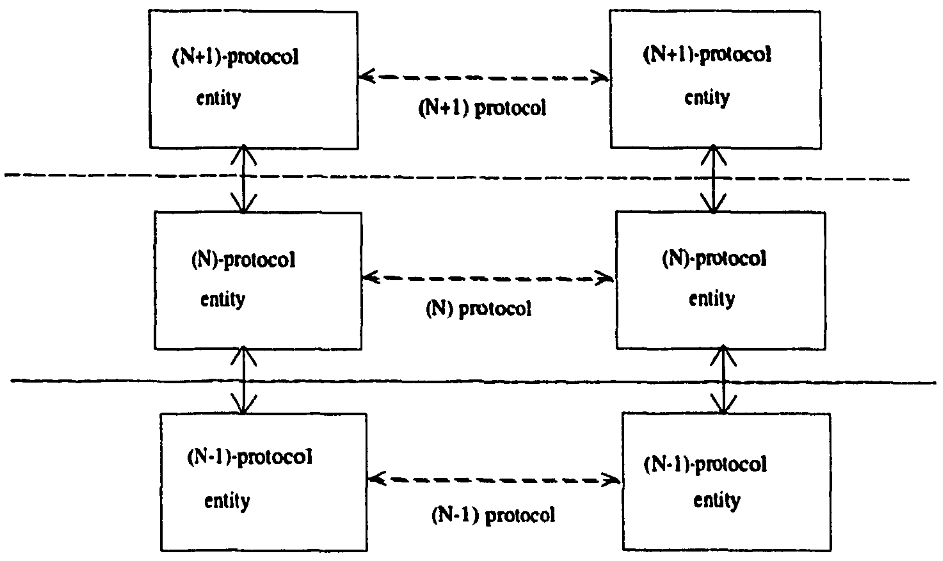

Figure 4.1: Layer model service and protocols

layer is described in terms of two documents: a service definition and a protocol specification. The service definition defines the services provided by this layer to the next higher layer. The protocol specification defines the set of communication conventions followed by the protocol entities in this layer.

For adjacent layers, say layers *N+1* and *N*, the *N+1* layer's protocol entities, as the *service users*, communicate with each other via the *N* layer's entities, which serve as the *service provider.* In order to communicate via the service provider, the service user utilizes called *service primitives.* A service primitive is an abstract, implementation-independent element of an interaction between a service user and the service provider. The interactions take place at the common boundary of a service user and the service provider, called a *Service Access Point (SAP).* Based on the concepts above, the example shown in Figure 4.1 can be simplified as shown in Figure 4.2.

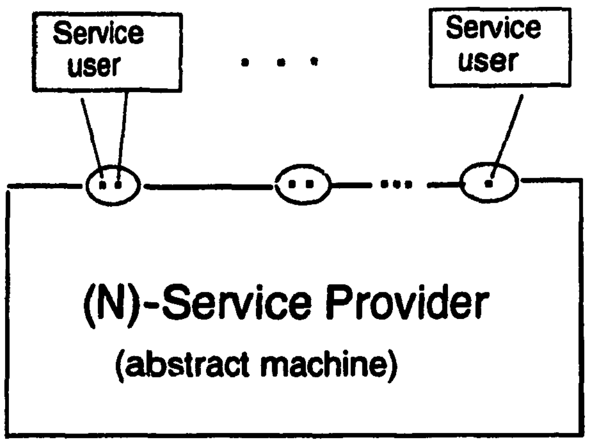

Figure 4.2: Model of service

A service provider is seen as an abstract machine accessible from a number of service access points. The abstract machine hides its internal structures, such as the protocol entities and underlying service provider. Therefore the definition of the service defines the behavior of the provider rendering the service to the next higher layer, which is expressed by the service primitives and the possible ordering of service primitives.

The definition of service is indispensable to the design of a protocol for various reasons. As a simple substitution for the protocol entities of a layer and the underlying service, the abstract machine can be used to express in simple terms the functions accomplished in the layer and to formally verify the operations of the protocol. Additionally, the service is the user's real concern. Most users are only interested in the service provided to them. For more reasons for the importance of service definitions see [Vissers], in which the authors point out that the specification of service should precede or accompany, but definitely not follow, the specification of the protocol.

An informal service definition for XTP is presented in Appendix D of [XTP92], which is taken from an XTP implementation. However it is incomplete and described in an implementation-dependent form. In the rest of this chapter, we will propose the definition of the XTP service. It includes the definition of service primitives, their parameters, and the time sequence of primitives.

## 4.2 Service primitives

### 4.2.1 General considerations

The service primitives are implementation-independent representations of interactions between the service user and the service provider. They express both the user's needs and the capabilities of the service provider. From the user's point of view, primitives should possess characteristics of conciseness and generality in order to be suitable for various protocols in the same layer. However our purpose is to propose the definition of service for XTP rather than a general transport service. Our definition therefore will reflect the features of XTP, for example, multicast service, flexible operation, etc., as discussed in Chapter 3.

The ISO document [ISO82] recommends four kinds of primitives. They are request, indication, response, and confirm in time order. We will adopt this kind of syntax, but in some cases the semantics are varied to meet the requirements of XTP. The parameters of the service primitives are defined in such way that it is easy to map them into protocol data units.

### 4.2.2 Service primitives for unicast

There are 23 service primitives for unicast mode. They are divided into 4 groups by their functions as follows.

**Group 1:** Context creation

1. OPEN.request (*address-segment, options, service, flags, user-buffer*)

   This is used to request the creation a context, and to indicate that the user wishes to initiate an association. It also serves as the request to establish an association, similar to the "connection-request" in TCP/IP service. It will be mapped into a "*FIRST*" packet, if the context is created successfully.

2. OPEN.confirm (*key-index, confirm-code*)

   This is used to confirm or reject the initiating user's request for creation of a context. If the *confirm-code* indicates success, the provider returns an end-point identifier, *key-index*. Otherwise, the user's attempt to initiate an association has failed locally.

3. LISTEN.request (*address-segment, options, service*)

   This is used to request creation of a context for the responder of an association. It will place the context in "listen" state upon successful creation.

4. LISTEN.confirm (*key-index, confirm-code*)

   This is used to confirm the responding user's request for creation of a context. If the *confirm-code* indicates successful, the provider returns an end-point identifier, *key-index*. Otherwise, the user's attempt to respond to an association has failed locally.

**Group 2:** Connection

1. ASSOCIATION.indication (*address-segment, options*)

   This is used to indicate that a *FIRST* packet has been received, thus establishing the association.

2. ASSOCIATION.response (address-segment, options, confirm-code)

   This is used to allow the Responder to acknowledge receipt of the above indication. This primitive may not be necessary if "confirm" in parameter "*options*" of primitive "LISTEN.request" was not set. Details will be discussed in the next section.

3. ASSOCIATION.confirm (*address-segment, options, confirm-code*)

   This is used to confirm the success (or failure) of the association establishment.

**Group 3:** Data transfer

1. SEND.request (*user-buffer, flags*)

   This allows the user to provide a descriptor of an outgoing data buffer. In the Initiator, this primitive is legal immediately after receipt of the OPEN.confirm. In the Responder, this primitive may only be used after receipt of ASSOCIATION.indication (and issue of ASSOCIATION.response, if adopted).

2. SEND.confirm (*user-buffer, confirm-code*)

   This allows the user to release the outgoing data buffer provided in a previous SEND.request. Depending on the semantics of the type of Service chosen, this may also imply successful delivery.

3. RECEIVE.request (*user-buffer, flags*)

   This allows the user to provide a descriptor for an incoming data buffer.

In the responder, this primitive is legal immediately after receipt of ASSOCIATION.indication (and issue of ASSOCIATION.response, if adopted). In the Initiator, this primitive may only be used after receipt of ASSOCIATION.confirm.

4. RECEIVE.confirm (*user-buffer, confirm-code*)

   This indicates that the incoming data buffer provided in a previous RECEIVE.request has been filled on successful "*confirm-code*", and control of the buffer has been returned to the user.

**Group 4:** Disconnection

1. CLOSE-SEND.request

   This is used to close the SEND path in an XTP association.

2. CLOSE-SEND.indication

   This indicates that the remote peer has (gracefully) terminated sending data.

3. CLOSE-SEND.response

   This is used to acknowledge receipt of the CLOSE-SEND.indication

4. CLOSE-SEND.confirm

   This indicates that the SEND path has been closed.

5. CLOSE-RECEIVE.request

   This is used to close the RECEIVE path in an XTP association.

6. CLOSE-RECEIVE.indication

   This indicates that the remote peer has (forcibly) terminated the reception of data.

7. CLOSE-RECEIVE.response

   This is used to acknowledge receipt of the CLOSE-RECEIVE.indication

8. CLOSE-RECEIVE.confirm

   This indicates that the RECEIVE path has been closed.

9. CLOSE.request

   This is used to close both the RECEIVE path and SEND path in an XTP association.

10. CLOSE.indication

    This indicates that the remote peer has terminated sending and receiving data.

11. CLOSE.response

    This is used to acknowledge receipt of the CLOSE.indication

12. CLOSE.confirm

    This indicates that the association has been closed completely.

### 4.2.3 Parameters of service primitives

XTP primitives have to have more parameters than most other transport layer service primitives, because XTP can implement association establishment, data transfer, and association termination in a single (*FIRST*) packet, implying that a single primitive was invoked (i.e., OPEN.request). Association termination can be performed in company with data transfer, and two data streams of an association can be closed separately, so primitives SEND.request and RECEIVE.request must have parameters to indicate this. In addition to the quality of service, XTP provides the user with a variety of modes of operation (NOCHECK, NOERR, NOFLOW, MULTI), which must be indicated as parameters.

In more detail, the parameters and their semantics are as follows:

**address-segment** indicates the source address and destination address. It consists of three parts as follows:

- address-family, which addressing domain is being used.
- destination-address, in one of the XTP address formats.
- source-address, in the same XTP addressing format.

**options** indicates the mode of operations. It includes *Nocheck, Noerr, Noflow, Fastnak, Rcs, Multi, Sort*, and *Confirm*. The meanings of these options are the same as definitions in XTP except the last one. "Confirm" in OPEN.request is like "SREQ" in the *FIRST* packet. While "Confirm" in LISTEN.request is used to indicate the response mode, manual or automatic. For the manual mode selected by setting "Confirm" on, the user must issue ASSOCIATION.response after receipt of ASSOCIATION.indication. For the automatic mode, the service provider will automatically generate ASSOCIATION.response.

The "options" in OPEN.request, LISTEN.request and ASSOCIATION.response are from the local user, while "options" in ASSOCIATION.indicate and ASSOCIATION.confirm are from the partner.

**service** indicates one of the six types of services as defined in XTP.

**flags** indicates one or more operations, that is, Rclose, Wclose, Eom, and Btag as defined in XTP.

**user-buffer** indicates a descriptor of a user data buffer. It consists of a pointer to a user buffer and the size of the buffer.

The parameter "user-buffer" in OPEN.request and SEND.request is used to specify the location and length of data to be sent. The length may be zero, which means there are no user data to be sent, so the pointer is meaningless.

In this case, the primitive will be mapped into the *FIRST* packet without user data for the OPEN.request, or a *DATA* packet with a specially coded "btag" field for the SEND.request.

The "user-buffer" in SEND.confirm is used to report which user buffer may be released and how much data in it have been delivered to the partner.

The "user-buffer" in RECEIVE.request indicates the location and size of the buffer used to put data received from the partner.

The "user-buffer" in RECEIVE.confirm is used to report which user buffer is filled and how much data from the partner are in it.

**key-index** indicates an association endpoint identifier in order to differ from one installation of a given service to another. It corresponds to an "index" of a context in XTP.

**confirm-code** is used to report the result of relative service primitives, *success* or *failure*.

### 4.2.4 Service primitives for multicast

All the primitives for unicast are available for multicast. Note that the use of these primitives in multicast mode must obey the following rules:

1. The *Multi* of "options" must be set on and the destination-address in both OPEN.request and LISTEN.request should be a group-address.
2. Only one ASSOCIATION.confirm is issued for received multiple ASSOCIATION.responses with successful "confirm-code".

3. The Initiator determines the "options" of the association if the automatic mode is adopted.

4. The primitive SEND.request is not available to the listeners, and RECEIVE.request is not available to the broadcaster.

5. The issued SEND.confirm with successful "confirm-code" means that the data have been delivered to at least one listener.

6. For the listeners, the primitives that close the SEND path are not available, and the primitives that close the RECEIVE path are identical to the primitives that close the association.

7. For the broadcaster, the primitives that close the RECEIVE path are meaningless, and the primitives that close the SEND path are identical to the primitives that close the association.

8. In the listener side, a CLOSE.confirm is sent back to the listener immediately after the listener issues a CLOSE.request. It is not necessary to wait for a CLOSE.response.

9. In the broadcaster side, the issue of CLOSE.confirm means that all associations with listeners are closed.

The extra primitives for multicast are the following:

1. JOIN.indication

   This is used to indicate that a *FIRST* packet has not been received in a certain period.

2. JOIN.response (confirm-code)

   This indicates that the user desires either to join an in-progress multicast association (set "confirm-code" as *success*), or not (set "confirm-code" as *failure*).

3. DROP.indication

   This is used to indicate failures, for example, unrecoverable loss of data.

4. DROP.response (confirm-code)

   This allows the user to indicate whether or not to continue the conversation. If the user desires to insist on the conversation (set "confirm-code" as *failure*), the provider will provide the multicast rejoin service. Otherwise, the association will be terminated.

## 4.3 Time sequence of service primitives

The possible ordering of service primitives is an important part of a service definition. It defines both the allowed sequences of primitives at the provider/user interface (i.e., service access point) and the allowed sequences of primitives between peer users. It also shows users how to correctly use the service.

### 4.3.1 Time sequence diagram

The ISO document [ISO82] introduces a notation called the time sequence diagram to illustrate how sequences of primitives are related in time. In a time sequence diagram, the two vertical lines separate the users and the provider. The service provider is between the vertical lines, and on the right and left side of the provider are the service users. The lines represent the service access points between the users and the provider.

Arrows in the user areas indicate that primitives are to or from the user. Sequences of primitives at each service access point are positioned along lines representing the passage of time, increasing downwards. Necessary sequence relations between peer users are represented by a diagonal dashed arrow between the lines representing service access points. If one primitive is no consequence of another, the dashed arrow is omitted.

### 4.3.2 Time sequence diagrams for the XTP service

In this section, we will discuss the ordering of XTP service primitives with time sequence diagrams. The following figures will illustrate the typical orderings and events involved in unicast mode.

Figure 4.3 and Figure 4.4 show the possible sequences of primitives for creating contexts. In Figure 4.3, the attempt to create a context at the initiator side fails. Figure 4.4 shows an example where the contexts are successfully created at both sides, and that the creation of the context at the responder side must precede the arrival of the OPEN.request from the initiator side.

Figure 4.5 and Figure 4.6, as the successors to Figure 4.4, show the possible sequences of primitives for establishing an association. The responder sets the response mode by the parameter in LISTEN.request. In case of manual response mode, as shown in Figure 4.5, the user must respond to the received ASSOCIATION.indication. Figure 4.6 illustrates the automatic response mode, in which the ASSOCIATION.response from the user is no longer required.

Figure 4.7 to Figure 4.10 show some cases related to data transfer. Figure 4.7 and Figure 4.8 show when the initiator and responder are able to send data in the manual response mode and automatic response mode, respectively. Figure 4.9 shows that the initiator must wait for ASSOCIATION.confirm to start receiving data, while the responder is able to use the primitive RECEIVE.request after a successful creation of the context. Figure 4.10 is used to show the relation between sending data and receiving data in the normal case. Only when data have been received from the remote peer, is it possible to confirm a previous RECEIVE.request. For the data sender, the previous SEND.request cannot be confirmed until the data have been delivered to the partner.

The rest of the figures show some examples of closing an association. Figure 4.11 shows a graceful close of one path of an association, that is, the path is closed by using close send primitives. Figure 4.12 also shows a case of closing a path, but in this case the receiver side of the path is closed first. Figure 4.13 illustrates that an association is closed by using close primitives resulting in a graceful close and a forced close. A graceful close of an association is given in Figure 4.14. Figure 4.15 is another example of a graceful close of an association, in which only six primitives are used. In Figure 4.16, the number of used primitives is also six, but the association is forcibly closed. Figure 4.17 is used to illustrate a crossed close of a path, that is, both sides of the path want to close at the same time. In this case, the send end of the path treats the CLOSE-RECEIVE.request from the other end as a CLOSE-SEND.response. Similarly, the receive end treats the CLOSE-SEND.request from the peer as a CLOSE-RECEIVE.response. Finally, Figure 4.18 shows a crossed close of an association, in which both ends of the association treat the CLOSE.request from the peer as a CLOSE.response.

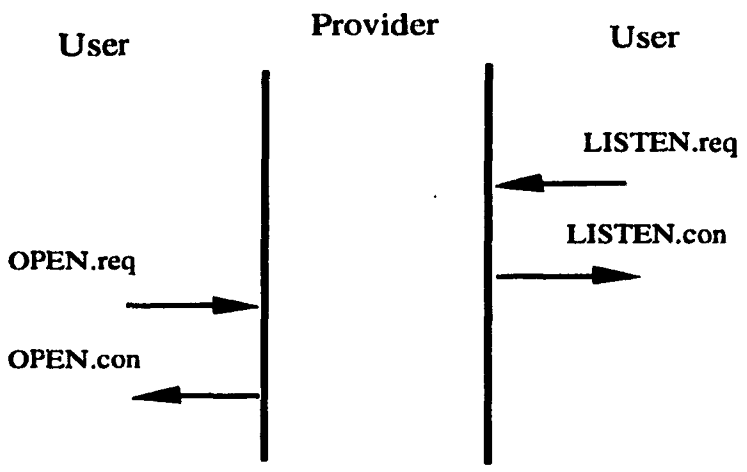

Figure 4.3: Creation of contexts (Open.request fails)

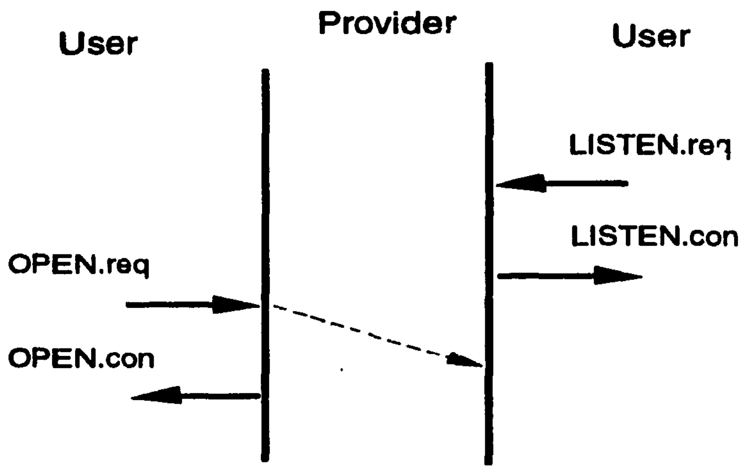

Figure 4.4: Successful creation of contexts in both sides

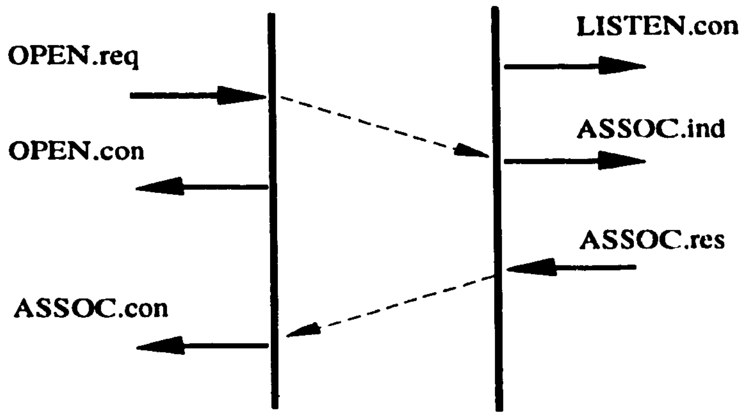

Figure 4.5: Establishment of an association (manual response mode)

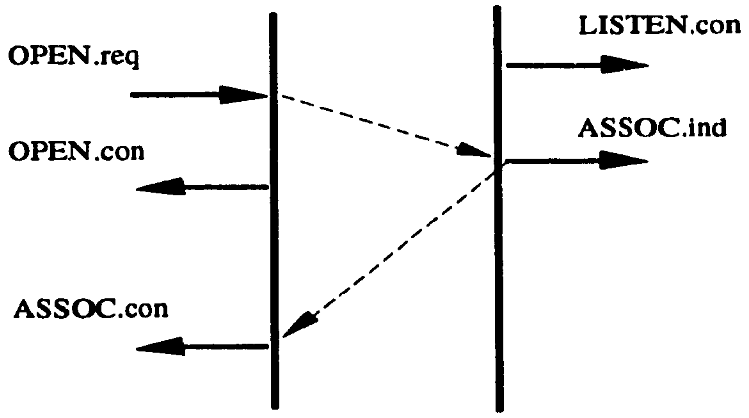

Figure 4.6: Establishment of an association (automatic response mode)

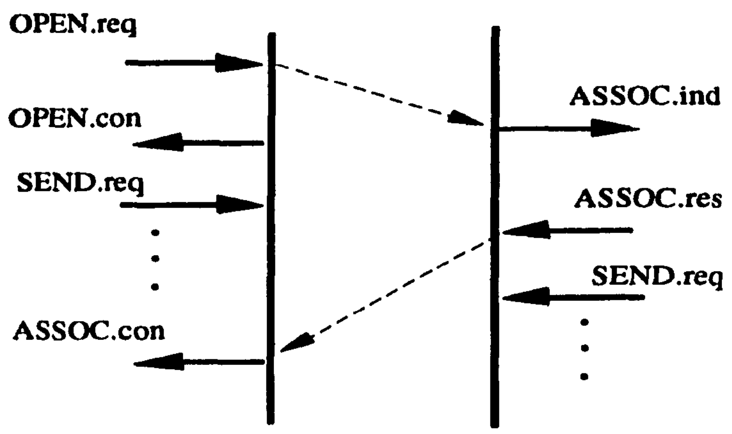

Figure 4.7: The legal start to send data (manual response mode)

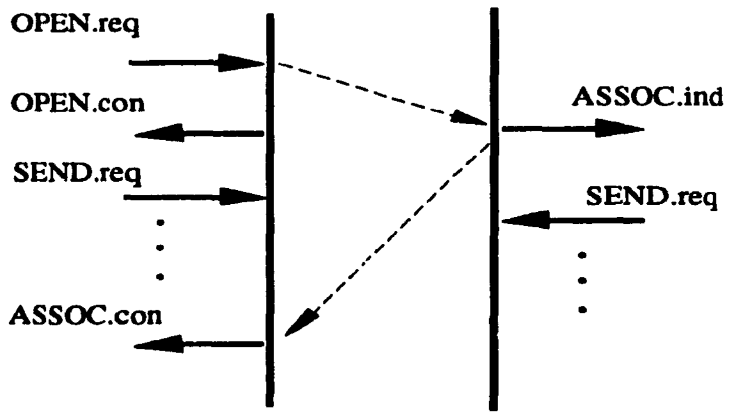

Figure 4.8: The legal start to send data (automatic response mode)

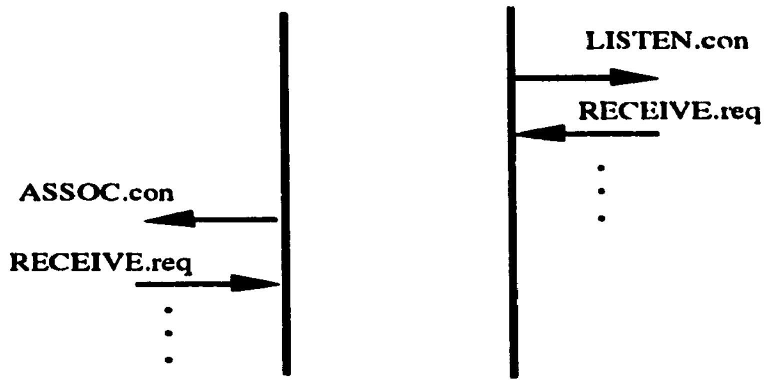

Figure 4.9: The legal start to receive data

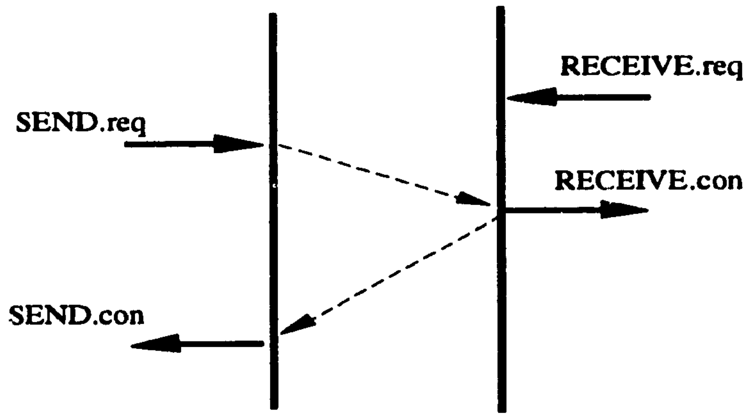

Figure 4.10: Data transfer procedure

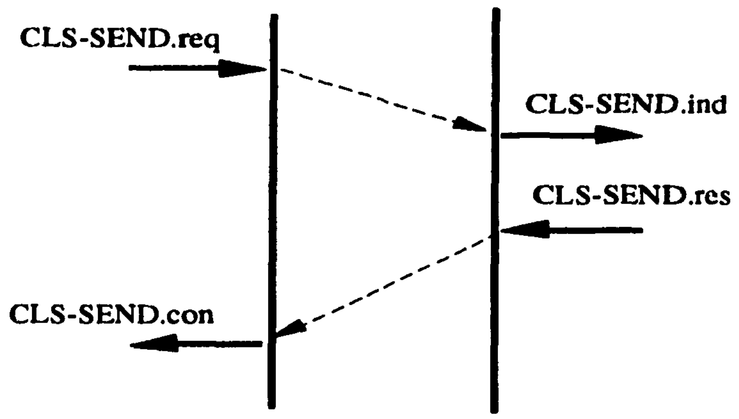

Figure 4.11: A graceful close of one path of an association

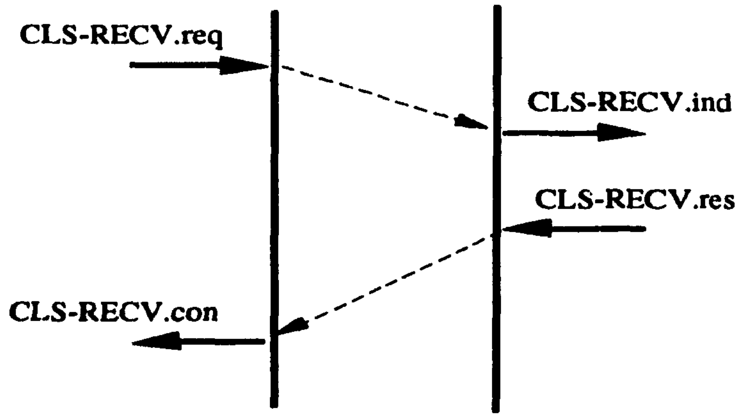

Figure 4.12: A forced close of one path of an association

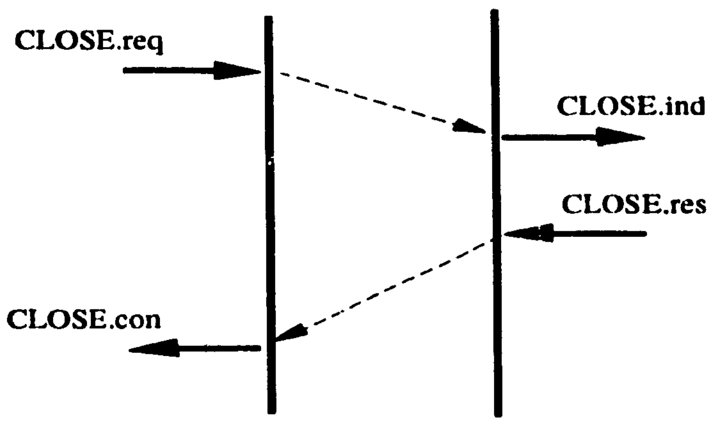

Figure 4.13: A close of an association with 4 primitives

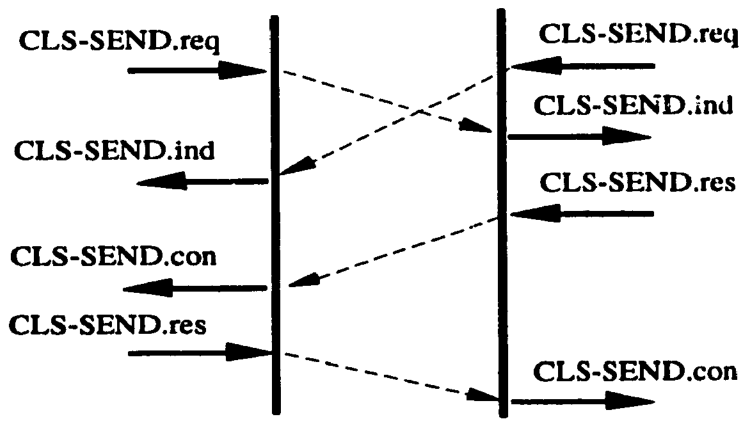

Figure 4.14: A graceful close of an association with 8 primitives

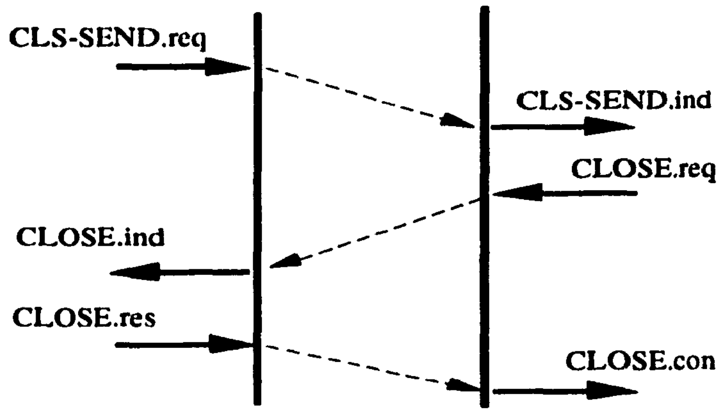

Figure 4.15: A graceful close of an association with 6 primitives

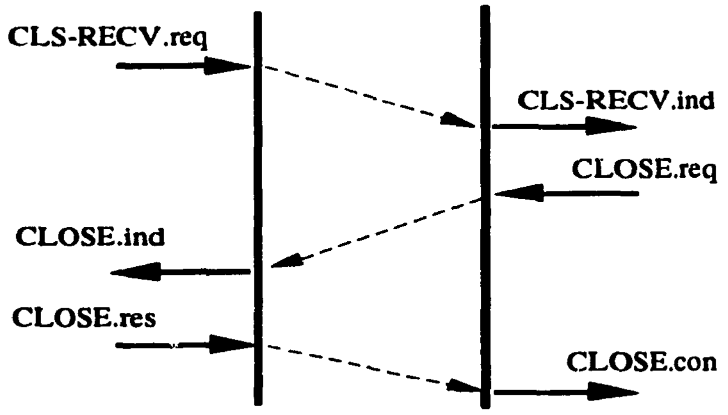

Figure 4.16: A forced close of an association with 6 primitives

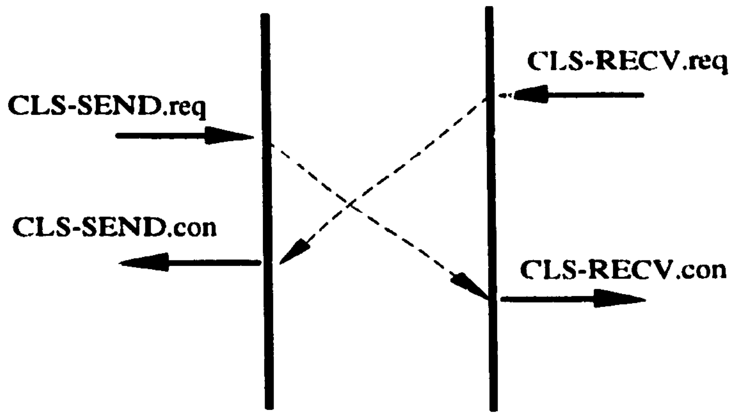

Figure 4.17: A crossed close of one path of an association

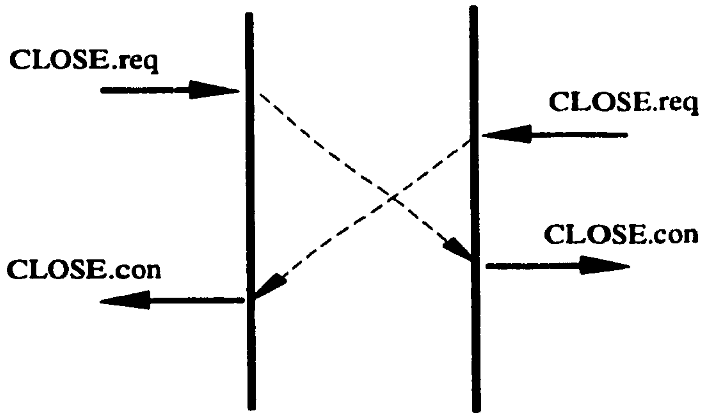

Figure 4.18: A crossed close of an association

## 4.4 A summary

In this chapter, we have proposed the definition of XTP service primitives, and informally described some typical orderings of primitives with time sequence diagrams. These diagrams only give us an overview of the basic features. However, it is very difficult to show all possible sequences of service primitives by using time sequence diagrams. This is why various formal description techniques have been developed, and used by more and more people. In the next chapters, we will present a formal specification of the XTP service in LOTOS.

# Chapter 5

# Unicast service specification in LOTOS

The formal specification of the XTP service defined in the last chapter was divided into two parts, the unicast service and the multicast service. This chapter will only present the formal specification of the unicast service.

## 5.1 Specification design principles

A LOTOS specification can be organized in many ways. In this specification, the approach called the constraint-oriented style will be used. In the constraint-oriented approach, the service is decomposed into a set of concurrent processes that focus on a collection of constraints. These constraints, as requirements, filter out only those interactions that may take place at a given gate at each moment. Thus, the allowed sequences of service primitives are specified. So a constraint-oriented style is appropriate for the specification of the service.

The constraints consist of local and end-to-end constraints. Local constraints assure the proper contents of service primitives used by service users and the permissible sequences of events at one service access point. For example, OPEN.request must be the first primitive issued by the user who wants to initiate a connection.

End-to-end constraints for the unicast service describe the relationships between events at different service access points. For instance, a SEND.request in the local end may be responded to with a successful SEND.confirm only if one or more RECEIVE.confirm primitives occur at the far end service access point, that is, the user data contained in that SEND.request have been delivered to the far end service user (see Figure 4.10).

## 5.2 Architecture of the specification

In the section 4.1, we described the model of service with Figure 4.2. Similarly, Figure 5.1 shows the global structure of the unicast service specification. In this specification, there are no parameters, and the whole service boundary is represented by a single interaction point, that is, gate XTPS. The interaction point is further decomposed into service access points and connection endpoint identifiers. Events at the gate XTPS consist of three values: the service access point, the connection endpoint identifier and the service primitive.

The unicast service is decomposed into two kinds of servers, initiator and responder, corresponding to the role of the user associated with the local endpoint. Additionally, the service underlying XTP is described by the Network-medium, which

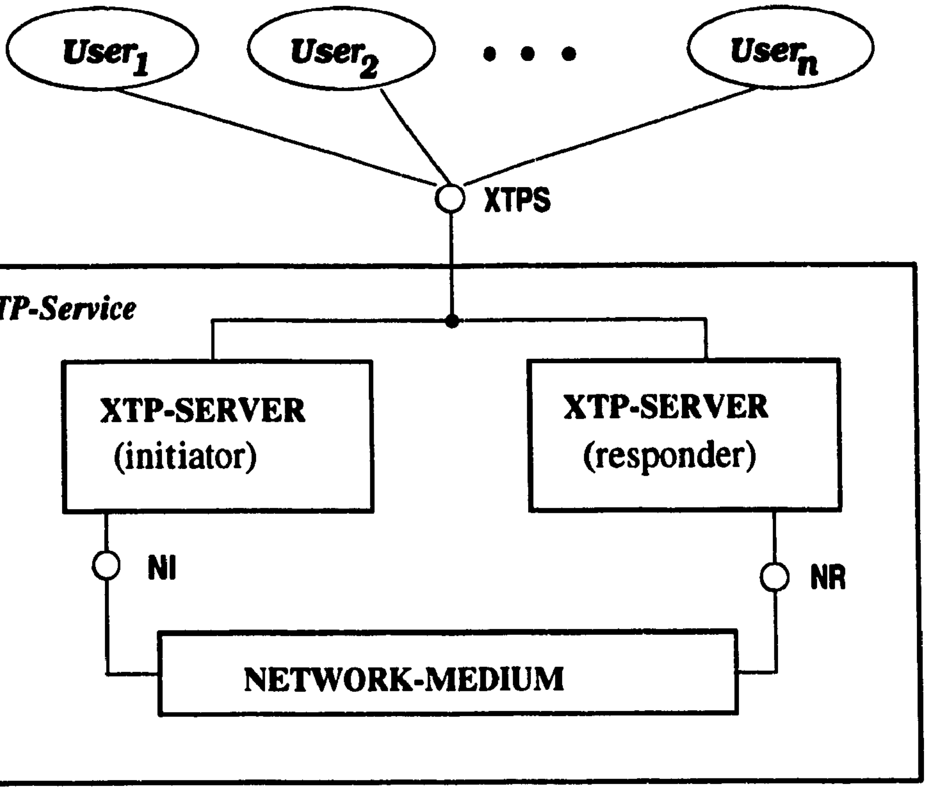

Figure 5.1: Architecture of XTP service (unicast)

interacts with the initiator and the responder at the internal gates NI and NR respectively, and provides FIFO queue mechanisms between them.

The skeleton of the specification text is as follows:

```
specification XTP-Service [XTPS] : noexit
   (* global type definitions *)
behaviour
   hide NI, NR in
   ( (  Server [XTPS, NI] (NA0, initiator)
        |||
        Server [XTPS, NR] (NA1, responder)
     )
     |[NI,NR]|
        Network-medium [NI, NR,] (NA0, NA1)
    )
where
    (* subordinate processes and local data types *)
endspec
```

Sample 5.1: Specification of the unicast service

## 5.3 Global data types

The global data type definitions relate to the representations of service access points, connection endpoints and service primitives.

### 5.3.1 Service access point identification

Service access points are an abstract means of interaction between service users and a service provider, and are identified by addresses. In the XTP document, the addresses are defined to support multiple addressing schemes. From abstractness of specification point of view, the most important requirement for defining service access points is to be able to distinguish any number of service access points. In this specification, each service access point address, called a user-index, consists of two parts: "Network-Address" and "User-Number". The first part is used to locate the site of service users and the service provider. The second part is used to distinguish the users within a site. The definition of the sort "user-index" is shown in the following sample.

```
type  User_Index
is    Network_Address, NaturalNumber, Boolean
sorts user_index
opns
        <> : -> user_index
        uid : Network_Address, Nat -> user_index
        hostaddr : user_index -> Network_Address
        userno : user_index -> NaturalNumber
        _eq_, _ne_: user_index, user_index -> Bool
eqns
        forall a : Network_Address, n : Nat, i1, i2 : user_index
        ofsort Network_Address
           hostaddr (uid (a, n)) = a;
        ofsort Nat
           userno (uid (a, n)) = n;
        ofsort Bool
           i1 eq i2 = ((hostaddr (i1) eq hostaddr (i2)) and
                 (userno (i1) eq userno (i2));
           i1 ne i2 = not (i1 eq i2);
endtype (* User_Index *)
```

Sample 5.2: Definition of the data type User-Index

On the basis of the sort "user-index", the set of user-indexes is defined as a sort of "user-indexes" by actualizing type Set in the library definition. The complete definition can be found in the specification text.

## 5.3.2 Connection endpoint identification

In the XTP service, several XTP associations can be established by a given user throug! ı given service access point. To distinguish different connections at one service ess point, we employ the concept of key index in XTP. Each connection endpoint identifier is defined as a key-index, which consists of two parts as the definition of "user-index". The definition of sort "key-index" is shown as follows:

```
type  Key_Index
is    User_Index renamedby
        sortnames
           key_index for user_index
        opnnames
           keyind for uid
           keyno for userno
endtype (* Key_Index *)
```

Sample 5.3: Definition of the data type Key-Index

Similarly, the set of key-indexes is defined as a sort of "key-indexes". With the definitions above, events at the gate XTPS will have the form:

*XTPS ? ui: user_index ? key: key_index ? sp: xtpsp*

where the sort "xtpsp" will discussed in the next section.

### 5.3.3 Service primitives

Service primitives are the main elements of interaction at service access points. The specification is complex because of the large number of primitives, some of which have many parameters. The specification includes:

- type definitions for each parameter;
- XTPSPtype (XTP Service Primitive type), which defines the sort of service primitive, "xtpsp", and functions for recognizing service primitives;
- XTPSPPS (XTP Service Primitive Parameter Selectors type), which is an enrichment of the type XTPSPtype with functions that pick out parameters from service primitives;
- ObjectType (Object Type), which defines the sort "Object" based on the XTP-SPtype. The objects are used to represent the information in transit on the network medium. The term "an object" referred in the rest of this chapter and the next chapter means a primitive received from the remote end via the network medium. For instance, a primitive RECEIVE.confirm to confirm the previous RECEIVE.request primitive will also be transferred to the other end as a data acknowledgment, which is called an object of RECEIVE.confirm.

For the sake of brevity, we simplify here the specification, show the definition of the primitive "OPEN.request" only, and omit the definitions of parameters and objects. Note that the primitive OPEN.request is presented as "open_request" in the specification. Similarly, this convention is used for other primitives.

```
type XTPSPtype
is   AddressSegment, ···, Boolean
```

```
sorts xtpsp
opns
    open_request : address_segment, options, service,
                   flags, user_buffer -> xtpsp
    ...
    Is_open_request : xtpsp -> Bool
    ...
endtype (* XTPSPType *)
```

Sample 5.4: Definition of service primitives

The following simplified example shows the partial definition of the function "s-address", which is used to select the parameter "address-segment" from service primitives.

```
type XTPSPPS
is   XTPSPtype, ···, Boolean
opns
    s_address : xtpsp -> address_segment
    s_flags : xtpsp -> flags
    ...
eqns
    forall a : address_segment, o : options,
        s : service, f : flags, bf : user_buffer, ···
    ofsort address_segment
      s_address (open_request (a, o, s, f, bf)) = a;
```

```
        s_flags (open_request (a, o, s, f, bf)) = f;
      ...
    endtype (* XTPSPPS *)
```

Sample 5.5: Definition of the parameter selections

## 5.4 Process structures

The top-level decomposition of the service, shown in Figure 5.1, describes the services that the service provider can offer, that is, the initiator and responder server. The Network-medium represents the correct bi-directional transfer of service primitives. In this section, we will discuss the structures of these processes in detail.

### 5.4.1 The initiator server

The initiator server is an instance of process *Server* with a parameter *initiator*. It represents the XTP unicast service constraints on the users who desire to initiate an association. Figure 5.2 shows the structure of this process.

According to the constraint-oriented approach, the initiator server is decomposed into four processes, *Addressing, Keymatching, SPordering* and *Datatransfer*. These processes also interact at a hidden gate K. The following is the corresponding LOTOS specification, in which some parameters are omitted for the purpose of brevity.

```
    process Server [XTPS, N] (NA : Network-address, R : serverrole) : exit :=
```

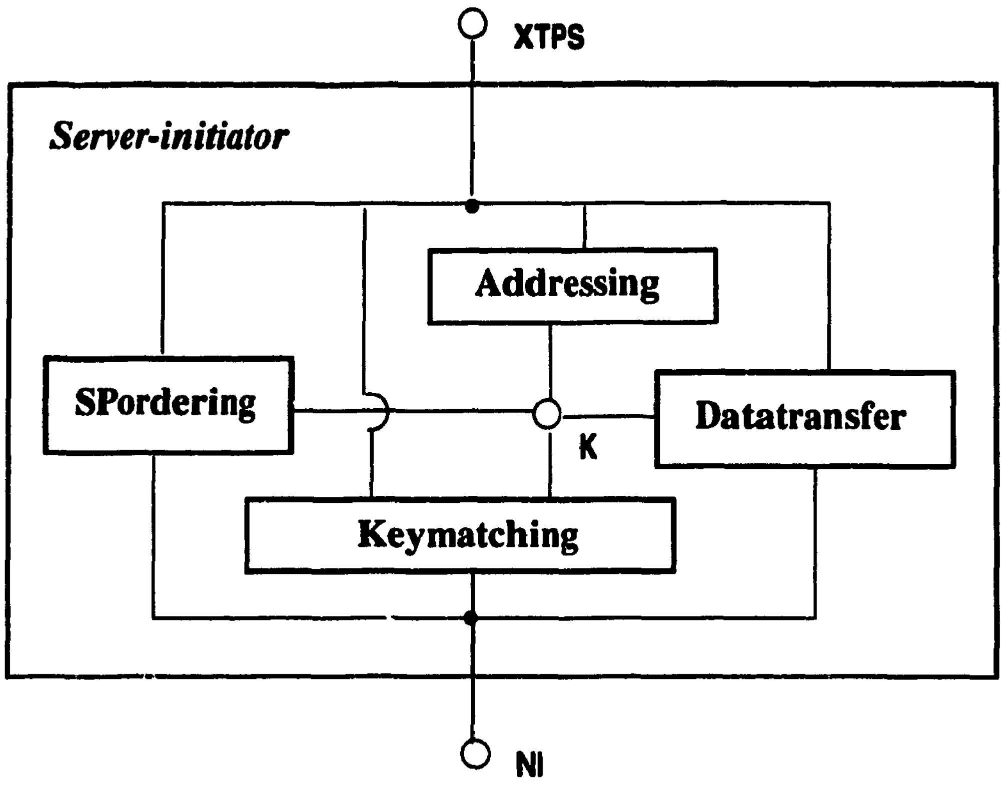

Figure 5.2: Constraint-oriented decomposition of process Server-initiator

```
hide K in
( ( SPordering [XTPS, N, K] (R)
      ||
      Datatransfer [XTPS, N, K] (···)
      ||
      Keymatching [XTPS, N, K] (···)
  )
  |[XTPS, K]|
  Addressing [XTPS, K] (NA)
)
>>
Server [XTPS, N] (NA, R)
```

**endproc** (* Server *)

Sample 5.6: Specification of the initiator server

## The Addressing

The process *Addressing* focuses on the local constraint on the service access point (also known as the address) at which the service primitives occur, i.e., all service primitives must occur at the address given in the parameter. The value is determined on the first event, in cooperation with XTP-service user, and thereafter is constant.

## The Keymatching

The process *Keymatching* ensures that a unique connection endpoint identifier, i.e., a key, is used for every XTP association at the local side. It means that all service primitives must occur with the key given in the parameter, and the same key is used in the whole lifetime of the association.

## The SPordering

The process *SPordering* for the initiator describes the constraints on the sequences of actions at an initiator's connection endpoint. In the normal situation, the ordering of service primitives is essentially constrained as follows:

1. the first event may only be an OPEN.request;

2. the event following the initial OPEN.request may be an OPEN.confirm; if OPEN.confirm is an unsuccessful one, the behaviour may be started again. Otherwise, any sequence of SEND.request may occur; and

3. the expected event is ASSOCIATION.confirm; if the ASSOCIATION.confirm is an unsuccessful one, the behaviour may be started again. Otherwise,

4. following the occurrence of an ASSOCIATION.confirm, any sequence of RECEIVE.request may occur;

5. SEND.request and RECEIVE.request with flags related to closing association will trigger actions as a close-association primitive;

6. at any point after the successful ASSOCIATION.confirm, a CLOSE-SEND.request (CLOSE-RECEIVE.request or CLOSE.request) or a CLOSE-SEND.indication (CLOSE-RECEIVE.indication or CLOSE.indication) may occur; and

7. after the association is closed, the whole behaviour may be repeated.

As discussed in chapters 3 and 4, the intentions to gain efficiency and to provide flexible operations in XTP mean that the lifetime of an association cannot be simply divided into a connection phase, data transfer phase and termination phase. To characterize these features, a state-oriented approach is used. In the state-oriented approach a process is decomposed into a series of sequential processes that may be regarded as the states in a final state machine.

In the specification, the process *SPordering* for the initiator is decomposed into a series of subprocesses grouped as follows:

1. *Halfopen*, *HopenWC* (Half open and closing write), *HopenRC* (Half open and closing read), *HopenWRC* (Half open and closing write/read), *HopenWCRC*

(Half open, closing write and closing read);

2. *Fullopen*;

3. *Lclosing* (Local closing), *LclosingW* (Local closing Write), *LclosingR* (Local closing Read), *Rclosing* (Remote closing), *RclosingW* (Remote closing Write), *RclosingR* (Remote closing Read);

4. *LclosingWR* (Local closing Write and Read), *Readonly*, *CrossedCW* (Crossed Close Write), *CrossedCR* (Crossed Close Read), *Writeonly*, *RclosingWR* (Remote closing Write and Read);

5. *LRclosed* (Local closed), *ROLcloseR* (Read Only and Local close Read), *WOLcloseW* (Write Only and Local close Write), *RORcloseW* (Read Only and Remote close Write), *WORcloseR* (Write Only and Remote close Read), *RLclosed* (Remote closed).

Each of them represents a state of the association. The transitions from a state are described by a choice of events followed by process instantiations. As an example, the following is a partial specification of process *Halfopen*.

```
process Halfopen [XTPS, N, K] (sp : xtpsp) : exit :=
    let f : flags = s_flags (sp) in
      (
       [Is_close_receive_request (sp) or Isrc (f_rc (f)))]
                  -> Hopenrc [XTPS, N, K]
       []
       [Is_close_send_request (sp) or Iswc (f_wc (f)))]
                  -> Hopenwc [XTPS, N, K]
    []
    [Is_close_request (sp) or (Iswc (f_wc (f))) and Isrc (f_rc (f)))]
                  -> Hopenwrc [XTPS, N, K]
    []
    ...        )
endproc (* Halfopen *)
```

Sample 5.7: Specification of process Halfopen

The constraints depend on which state the association is in. For example, after the occurrence of a successful ASSOCIATION.confirm, the association is in state "*Fullopen*", both SEND.request and RECEIVE.request are available. Once the CLOSE-SEND.request is issued by the local user, the association will be in a state "*LclosingW (Local closing Write)* at the local end. In this state, any SEND.request is no longer available to the local user. To explain the time sequence of these processes, the state

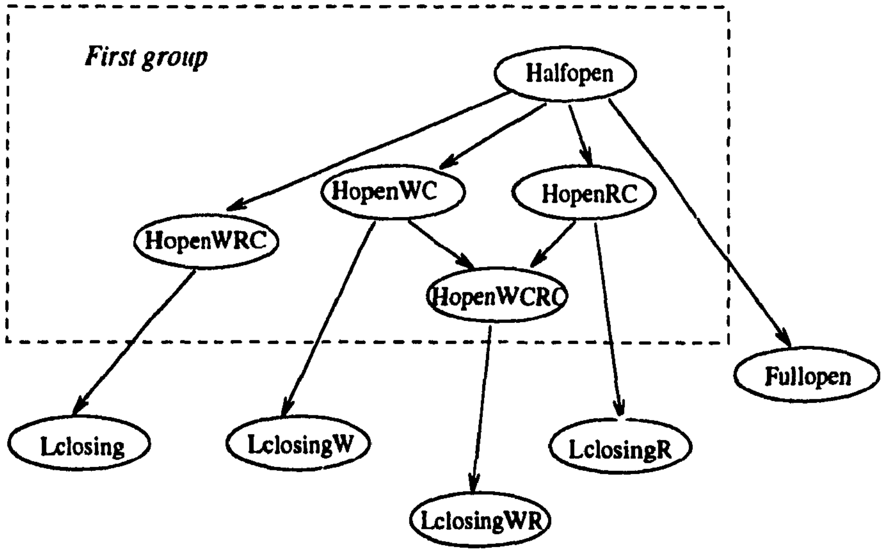

Figure 5.3: The first group subprocesses of initiator's SPordering

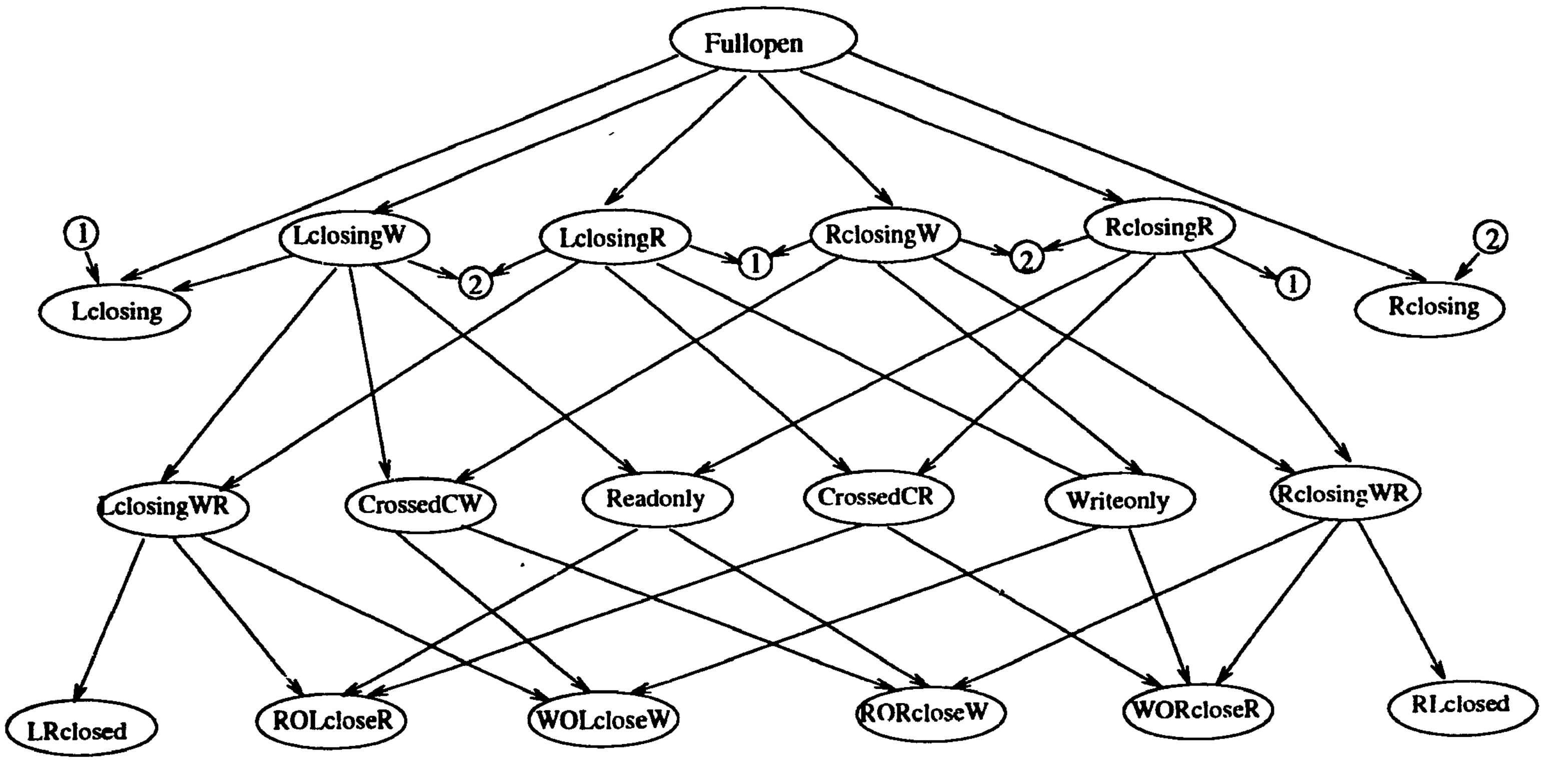

Figure 5.4: State diagram of the subprocess Group 2-5

diagrams as shown in Figure 5.3 and 5.4 are used. The diagram in Figure 5.4 is also applicable to the responder server, which will be discussed later.

### The Datatransfer

The process *Datatransfer* focuses on the constraints on the data transfer between two service users of an association. These constraints include:

- the occurrence of a SEND.confirm depends on either receiving an object of a RECEIVE.confirm from the other end and the size of user buffers in the object and in the SEND.request which is to be confirmed, or occurrence of events related to closing an association;
- the order and number of occurrences of SEND.confirm must correspond to occurrences of SEND.request (including OPEN.request if it has user data);
- the occurrence of a RECEIVE.confirm depends on either receiving an object of a SEND.request (including OPEN.request if it has user data) from the other end and the size of user buffers in the object and in the RECEIVE.request which is to be confirmed, or the occurrence of events related to closing an association;
- the order and number of occurrences of RECEIVE.confirm must correspond with occurrences of RECEIVE.request.

Two local data structures are needed for the description of the constraints mentioned above. One is *"sbq"* (sending-buffer-queue), which is used to record the sequence of user buffer heads (i.e., buffer address and length) in SEND.requests. Another one is *"rbq"* (receiving-buffer-queue), which records the sequence of user buffer heads in RECEIVE.requests. The basic operations on these data structures follow a FIFO discipline defined in the type *Buffer_queue* in the LOTOS specification.

The process *Datatransfer* is decomposed into four processes as follows:

The process *Sending* takes care of primitives SEND.request issued by the local user and objects of RECEIVE.confirm from the far end. For the former, the corresponding user buffer is added to the *sbq* to record the order of user buffers that will be confirmed. For the latter, it may trigger issuing one or more SEND.confirms to the user, and then popping the queue *sbq*.

The process *DiscardS* describes the constraints related to ending an output data stream. It will forcibly confirm all previous SEND.request primitives with a failure flag.

The process *Receiving* concerns primitives RECEIVE.request issued by the local user and objects of SEND.request from the far end. For the RECEIVE.request, the corresponding user buffer is added to the end of *rbq* to record the order of buffers that will be confirmed. For the object of SEND.request, the length of user buffer inside the object is added to a variable recording the length of data to be delivered. Both of them may trigger issuing one or more RECEIVE.confirms to the user, and then popping the queue *rbq*.

The process *DiscardR* describes the constraints related to ending an input data stream. It will forcibly confirm all previous RECEIVE.request primitives with a failure flag.

### 5.4.2 The responder server

The responder server is another instance of process *Server* with a parameter *responder*. It represents the XTP unicast service constraints on the users who desire to respond to an association. Figure 5.5 shows the structure of this process.

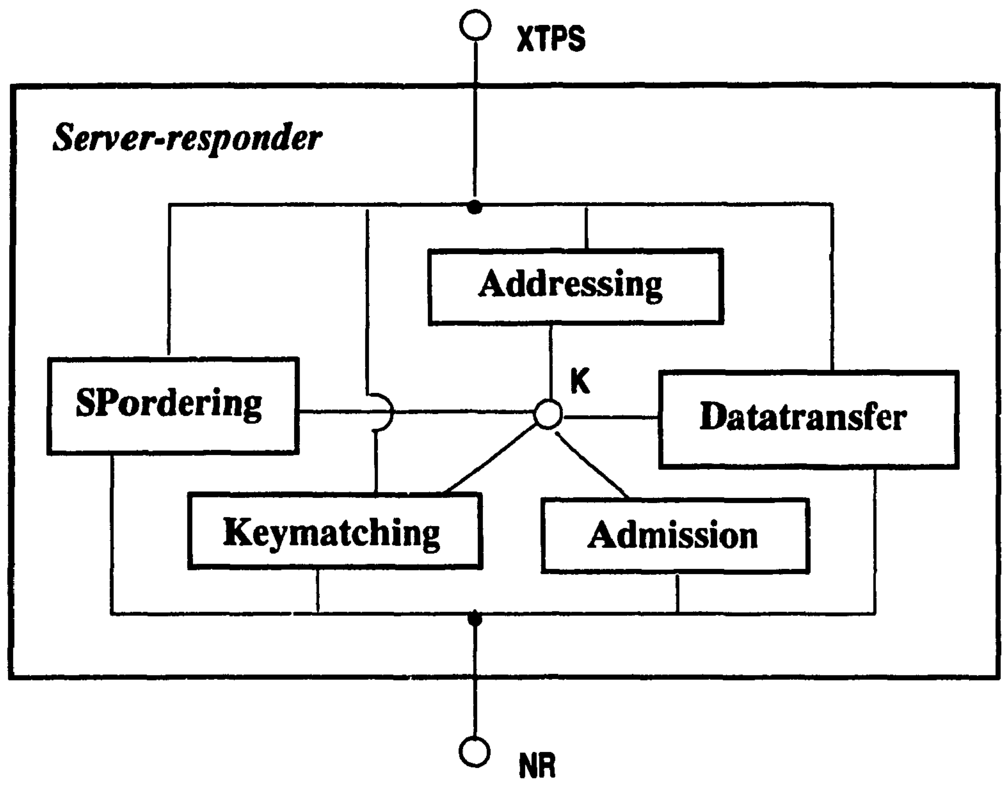

Figure 5.5: Constraint-oriented decomposition of process Server-responder

The structure of *Server-responder* is similar to the process Server-initiator except that there is one more process, *Admission.* The process *Admission* represents the constraint that an object of OPEN.request is accepted if and only if its address-segment matches the address-segment indicated in the listen-request.

Processes *Addressing, Keymatching* and *Datatransfer* are identical to those of the *Server-initiator.* However, the *SPordering* is different from that of the *Server-initiator* because constraints on establishing an association are different as follows.

1. the first event may only be a LISTEN.request;
2. the event following the initial LISTEN.request may be a LISTEN.confirm; if LISTEN.confirm is an unsuccessful one, the behaviour may be started again. Otherwise,
3. the expected event is an ASSOCIATION.indication and ASSOCIATION.response; if ASSOCIATION.response is an unsuccessful one, the next expected event is still an ASSOCIATION.indication.
4. following occurrence of a successful ASSOCIATION.response, any sequence of SEND.request and RECEIVE.request may occur; and
5. before receiving an ASSOCIATION.indication, issuing CLOSE-RECEIVE.request is illegal.

Therefore, the first group of processes in the decomposition of the responder's *SPordering* includes *Listening, ListeningWC (Listening and closing write), Listened, ListenedWC (Listened and closing write)*, as shown in Figure 5.6. The other groups are the same as those in the initiator's *SPordering.*

### 5.4.3 The Network-medium

The network service upon which XTP operates is not observable by the XTP service users. Nevertheless, it should be noted that we are modelling a totally reliable medium for both this and the multicast case. It is described by the process *Network-medium*, which provides two FIFO object queue mechanisms between the initiator and the

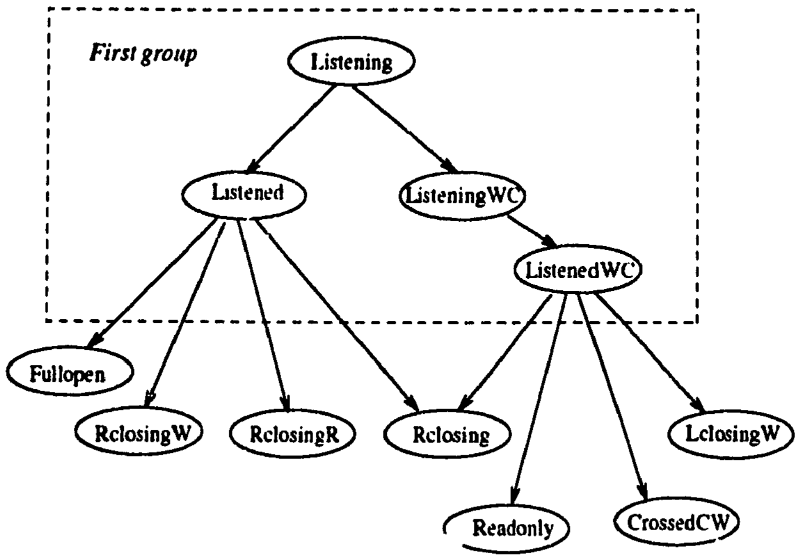

Figure 5.6: The first group subprocesses of responder's SPordering

responder. The sorts of the queues are defined in the definition of type Object-queue, which is a local data type of the process *Network-medium.*

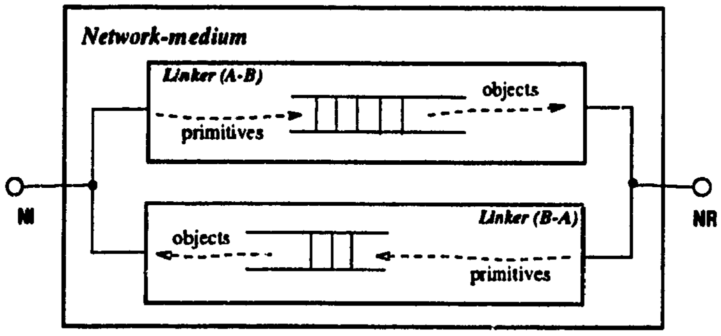

Figure 5.7: The structure of the process Network-medium

As shown in figure 5.7, the process *Network-medium* consists of a pair of interleaving processes, *Linker.* Service primitives, as the input of *Network-medium* at one end, are converted to the objects of these primitives and put in the object queue. Then these objects will be transferred to the other end as the output of *Network-medium*. The following is the specification of process *Network-medium*, in which we omitted the definition of local data structure *Object-queue*.

```
process Network-medium [NI, NR] (NA, NB : Network-address) : noexit :=
    Linker [NI, NR] (NA, NB, newobjq)
    |||
    Linker [NR, NI] (NA, NB, newobjq)
where
    process Linker [In, Out] (a, b : Network-address, q : object-queue) :
                    noexit :=
       (
         In ? Key : key-index ? sp : xtpsp [hostaddr (key) eq a];
         Linker [In, Out] (a, b, addobjq (object (sp), q))
       )
       []
       [not (Isemptyq (q))] ->
          (
            Out ? key : key-index ! first (q) [hostaddr (key) eq b];
            Linker [In, Out] (a, b, remove1st (q))
          )
    endproc (* Linker *)
endproc (* Network-medium *)
```

Sample 5.8: Specification of process Network-medium

# Chapter 6

# Multicast service specification in LOTOS

In the last chapter, we presented the formal specification of the XTP end-to-end service, that is, the unicast service. Based on this specification, we will now discuss the specification of the multicast service. Both the design principles and global data type definitions in the multicast service specification are the same as those discussed in the last chapter. Therefore, the emphasis in this chapter is on presenting the structure of the processes.

## 6.1 Architecture of the specification

The one obvious difference between unicasting and multicasting is the number of users involved in an association. In the unicast case, an association has exactly two users: one initiator and one responder. However, there may be more than two users connected to an association in the multicast, i.e., one initiator and multiple responders. This is why the multicast service is very difficult to describe by time sequence diagrams.

The global structure of the multicast service specification is shown as Figure 6.1, in which there are $m$ responders. In the specification text, we only represent two responders. In fact, all responders are identical and two responders are enough to specify the possible sequences of primitives for multicasting.

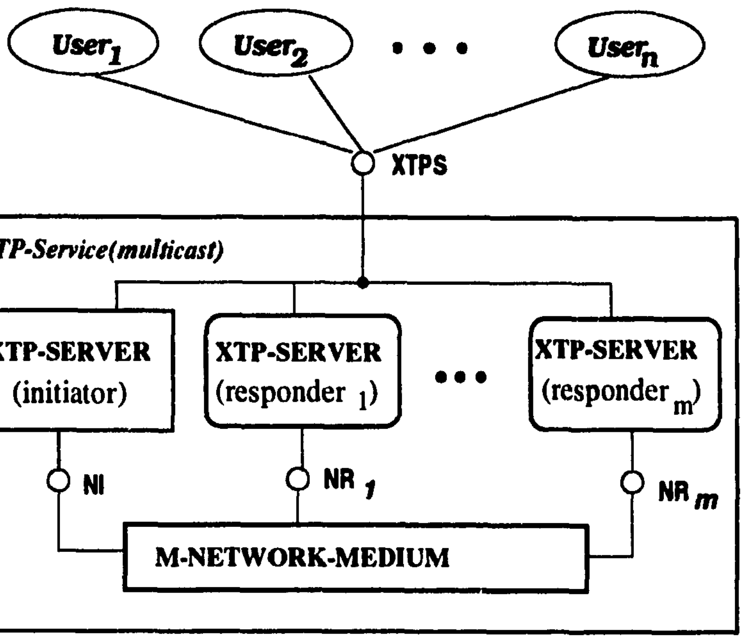

Figure 6.1: Architecture of XTP service (multicast)

One should note that components of the multicast service are different from those in the unicast service even although their names are very similar. For example, the underlying service of XTP multicast, called M-Network-medium, provides the same object-queue mechanisms as Network-medium in the unicast service. However, it also provides mechanisms for broadcasting objects amongst several communication partners. This will be covered in detail later.

The behaviour part of the specification text is as follows:

```
behaviour
   hide NI, NR1, NR2 in
   ( (  Server-m [XTPS, NI] (NA0, initiator)
        |||
        Server-m [XTPS, NR1] (NA1, responder)
        |||
        Server-m [XTPS, NR2] (NA2, responder)
     )
     | [NI, NR1, NR2] |
     M-Network-medium [NI, NR1, NR2] (NA0, NA1, NA2)
   )
```

Sample 6.1: Specification of the multicast service

## 6.2 The initiator server

The initiator server is an instance of process *Server-m* with parameter *initiator.* It represents the XTP multicast service constraints on the users who desire to initiate a multicasting association.

As Figure 6.2 shows, the initiator server is decomposed into five processes, *Addressing, Keymatching, Mi-SPordering, Broadcast* and *Timer.* The first two processes are the same as those in the unicast case. The processes *Mi-SPordering* and *Broadcast* are variants of *SPordering* and *Datatransfer* from the unicast case.

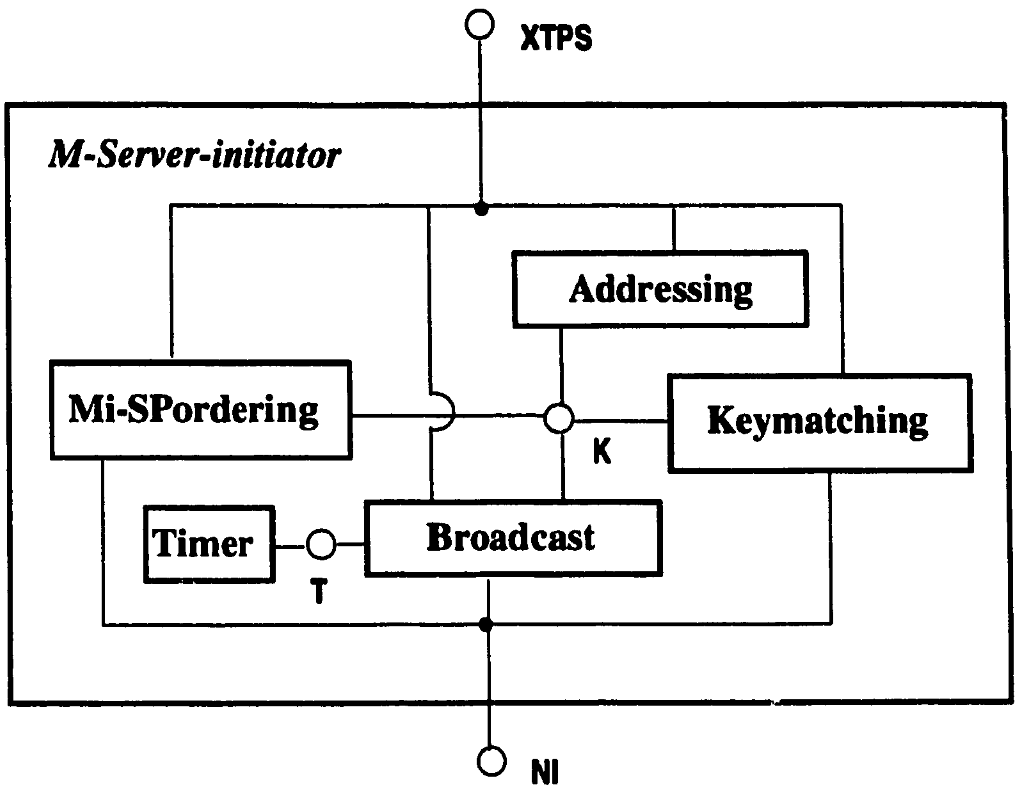

Figure 6.2: Decomposition of process Server-initiator

### 6.2.1 The Mi-SPordering

The process *Mi-SPordering* describes the constraints on the sequence of actions for a multicasting initiator. To determine what kinds of constraints it focuses on, we need to compare it with *Spordering* in the unicast initiator.

First, in the phase of opening an association, the process *M-Opening* is the same as *Opening* in unicast except that the multicast initiator may get back more than one acknowledgment of OPEN.request because there may be multiple responders. Even though it is possible to have multiple ASSOCIATION.responses back, only one ASSOCIATION.confirm occurs at the gate XTPS, and results from the first response to arrive. On the other hand, the rest of the responses are still acceptable.

Secondly, the multicast initiator will use the association in a different way from unicast initiators. The multicasting association only carries a one-way data stream, that is, from the initiator to the responders. Therefore, all events concerned with receiving data are disabled at the initiator side. For the same reason, all primitives for closing the receiving direction of the association cannot be used by the multicast initiator. The process *M-Writeonly*, as a direct successor to *M-Opening*, describes all constraints in the data transfer phase.

Thirdly, during the data broadcast phase, the object of a DROP.response may be received from some responders. Additionally, it is possible to receive an object of JOIN.response from a responder who missed the *FIRST* packet and wants to join the in-progress association. In fact, it may take place at any point after sending the OPEN.request, even if the association is being closed.

Finally, constraints on the action of closing a multicasting association are very different from the unicast case. To clarify these differences, let us imagine a multicasting association as a "tree". The initiator's end of the association is just like the root of the tree, and responders' ends are the "branches". Any responder's close will cut away only a branch of the tree. The tree is still alive even though all of the branches have been cut. Only a close from the initiator will result in the root of the tree being cut. This implies that only a close request from the initiator causes the transfer from *M-Writeonly* to the process *M-Lclosing.* Additionally, in the disconnection phase, the association is completely shut down after all existing responders have responded to the close request.

With the above analysis, the process *Mi-SPordering* in the specification consists of processes *M-Opening, M-Writeonly* and *M-Lclosing.*

### 6.2.2 The Broadcast and Timer processes

The process *Broadcast* focuses on the constraints on the data transfer from the initiator to the multiple responders. Unlike *Datatransfer* in the unicast case discussed in the last chapter, the constraints on receiving data are no longer of concern. The emphasis is on how to deal with reliable multicasting, that is, acknowledged multicasting, which requires that all responders receive data correctly. Although this issue of error control is outside the scope of specifying the service, we have to consider the constraints on when and how a SEND.confirm is returned to the initiator.

To specify the acknowledged multicasting, we employ a "timestamp" mechanism. The idea behind it is the following:

1. keep a list of existing responders, $L = \{r_i\}(i = 1, 2, \cdots, N)$ and
2. for each responder $r_i$ in the list, record the length of data that have been delivered, say $d_i$;
3. for each SEND.request, record its buffer address, buffer length and a "timestamp" in the timed-sending-buffer-queue (tsbq);
4. a SEND.request within tsbq is confirmed, when
    - all of responders have delivered the data in the SEND.request, that is,
      $$d_i \geq s, (i = 1, 2, \cdots, N)$$
      where $s$ is the sum of the length of confirmed data and length of buffer to be confirmed. At this point, the SEND.request is confirmed with a success flag.
    - timeout. The occurrence of timeout means that the data in the SEND.request have been delivered only by some responders within a certain period. The

reliable multicasting is downgraded to unreliable. A special case is when no responder delivered any data of the SEND.request. In this case, the SEND.request is confirmed with a failure flag.

To describe the constraints mentioned above, the process Broadcast uses local data structures, *"tsbq"* (timed-sending-buffer-queue) and *listeners*. The *tsbq* data structure is similar to the *sbq* discussed in the last chapter except timestamps are attached to each buffer head. The *listeners* data structure is used to record the existing responders and their delivered sequence numbers. In addition, it needs a variable to keep the length of confirmed data and a constant to specify the amount of time the sender waits for a reliable SEND.confirm.

In order to generate timestamps, the process *Timer*, a timing mechanism, is introduced. *Timer* interacts with the process *Broadcast* at the gate *T* to offer a logical timer.

The process *Broadcast* is decomposed into the following processes:

- *Connected* is used to deal with the object of ASSOCIATION.response, update the list *listeners*, and maybe issue an ASSOCIATION.confirm to the initiator.
- *Join* is related to the object of JOIN.response, and similar to the process Connected.
- *Rejoin* focuses on the object of DROP.response, and updates the responder's length of delivered data.
- *Disconnected* deals with the objects of CLOSE.request or CLOSE.response, and updates the *listeners*.

- *Sending* takes care of primitive SEND.request and adds it to *tsbq*
- *Sent* looks at the object of RECEIVE.confirm, decides whether to issue a SEND.confirm.
- *Unreliable-send* is triggered by the interaction with the process *timer*. It may result in issuing an unreliable SEND.confirm.

## 6.3 The responder servers

Each responder server is an instance of process *Server-m* with a parameter *responder*. It describes the XTP multicast service constraints on the users who want to respond to an association, and consists of four processes. Its structure is shown in Figure 6.3. We only discuss two processes here, *Mr-SPordering* and *Datareceiver*.

### 6.3.1 The Mr-SPordering

Unlike the responder's SPordering in unicast, the process *Mr-SPordering* has nothing in common with *Mi-SPordering* because their behaviors for transferring data and closing an association are totally different. Besides this difference, *Mr-SPordering* needs to focus on new constraints like joining and rejoining an existing association.

In the period of waiting for an object of OPEN.request from a multicast initiator, the process *M-Listening* is also monitoring the occurrence of objects of data sent by the initiator to determine whether an in-progress multicast association already exists. If it happens, the passive responder may take the initiative in joining the existing association. The process *Joining* will substitute for *M-Listening* to wait for

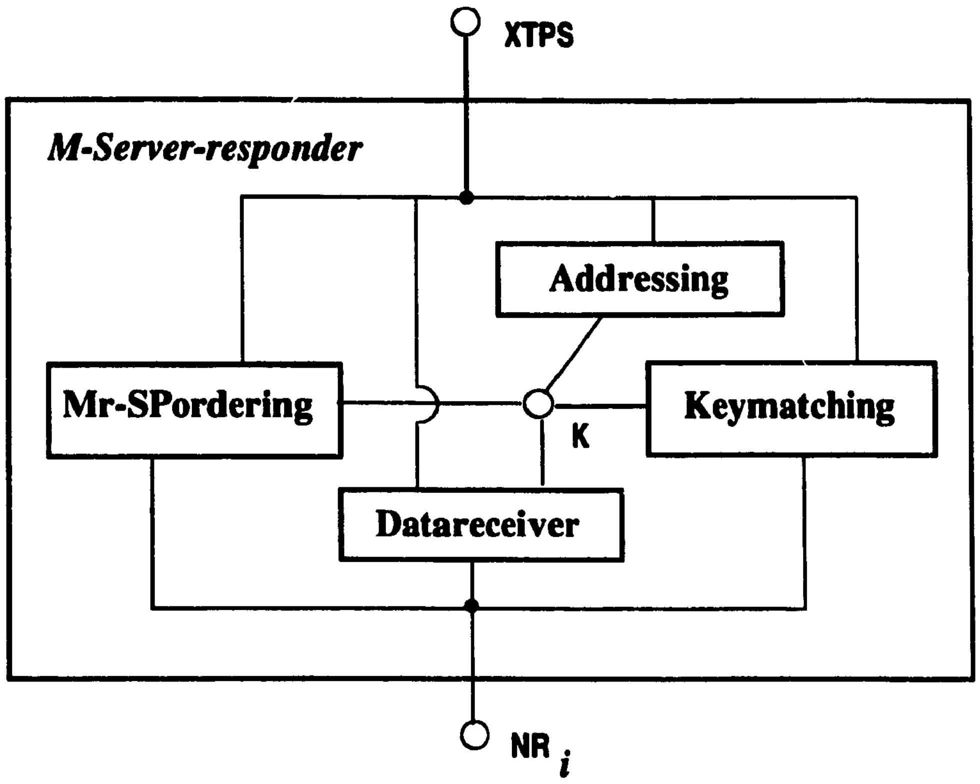

Figure 6.3: Decomposition of process Server-responder

the confirmation of the initiator.

Either *M-Listening* or *Joining* will lead to the process *M-Listened*, then the process *M-Readonly* may be activated. *M-Readonly*, like *M-Writeonly*, describes all possible primitives in the data transfer phase at a responder side.

During the data transfer phase, the primitive DROP.indication may be used to inform the local user of a failure for some reason. Then the normal process of receiving data will be interrupted to wait for the user's decision to rejoin or drop out. These constraints are specified by the processes *Dropout* and *Rejoining.*

The specification about closing an association is much simpler than that for unicast. The reasons are that the number of available close primitives to the user is reduced, and that it is no longer necessary to wait for the initiator's response to the responder's close request. However, the user is still required to respond to the close request from the initiator, which is described by the process *M-Rclosing*.

As a summary, Figure 6-4 shows all subprocesses of *Mr-Spordering* and their relationships.

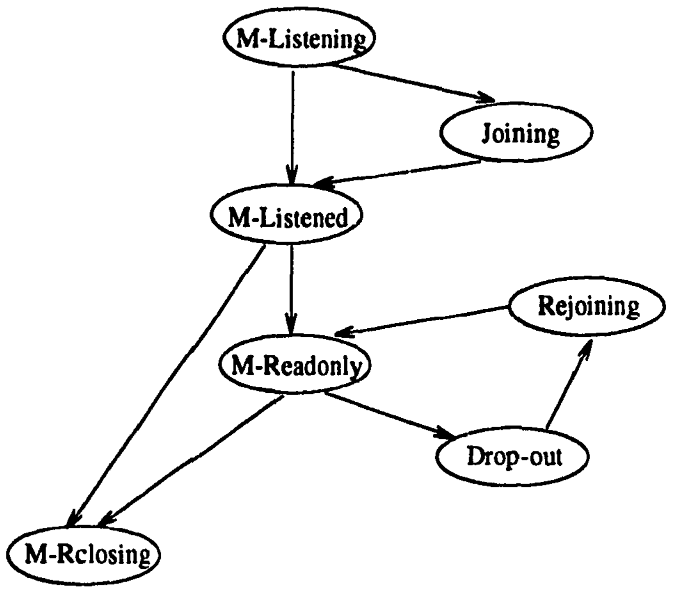

Figure 6.4: Decomposition of process Mr-SPordering

### 6.3.2 The Datareceiver

The process *Datareceiver* is a simplified *Datatransfer*. The local data structure *sbq* and processes related to data sending in *Datatransfer* are not necessary because responders are not allowed to send data.

## 6.4 The M-Network-medium

The process *M-Network-medium* consists of a set of processes, *M-Linker*, each bound to a communication partner in the multicasting. Similarly to the *Linker* in the unicast service, the process *M-Linker* provides an object-queue mechanism to keep the objects received from the other partners.

Notice that there may be more that two communication partners in the multicasting. *M-Network-medium* needs to be able to broadcast objects among partners. To model this mechanism, an internal gate *M* is used. All of *M-Linkers* will synchronize at the gate *M* for exchanging objects.

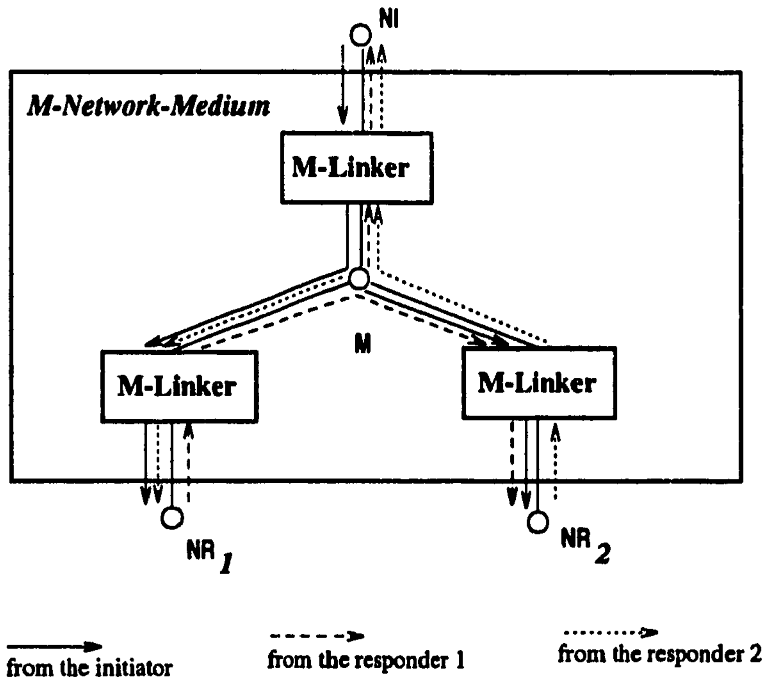

Figure 6.5: The structure of the process M-Network-medium

Figure 6.5 shows the structure of *M-Network-medium* and streams of objects. The simplified specification is shown as follows.

```
process M-Network-medium [NI, NR1, NR2]
                        (NA0, NA1, NA2 : Network-address) : noexit :=
  hide M in
  (
    M-Linker [NI, M] (NA0, newmobjq)
    |[M]|
    M-Linker [NR1, M] (NA1, newmobjq)
    |[M]|
    M-Linker [NR2, M] (NA2, newmobjq)
  )
where
  (* definition of process M-linker *)
endproc (* M-Network-medium *)
```

Sample 6.2: Specification of process M-Network-medium

# Chapter 7

# Conclusions

The layered architecture model of ISO simplifies the complexity of data communication systems. Based on this model, the concept of a service provider is introduced. To abstract the service provider, the definition of the service is as important as the definition of the protocol. The definition of the service, as an abstract interface between the service provider and service users, expresses the functions of a complex protocol in a simple way. It also defines the behaviors of the provider rendering the service to the next higher layer. In other words, it not only describes what services are provided, but also how the services are correctly used.

The first objective of this thesis was to contribute by proposing a definition of service for the XTP protocol. In chapter 4, a complete service definition is presented. It covers both the unicast and multicast service, and reflects the features of XTP as much as possible.

It may seem odd that there are 27 service primitives in the definition. In fact, 12 primitives related to the closing of an association are designed as ones without parameters. To reduce the number of primitives, we can alter the form of these primitives to be primitives with parameters. For example, CLOSE.request(path) could be used to substitute for CLOSE-SEND.request, CLOSE-RECEIVE.request, and CLOSE.request by indicating the parameter "path" as send, receive, and null, respectively. Similarly, other close primitives can be grouped together. Thus, the total of primitives could be reduced to 19. However, this alteration does not improve the definition of the functionality of XTP, or simplify the interactions of closing an association.

By defining the service of XTP, we came to the conclusion that the procedure for closing an association in XTP should be improved. XTP provides implicit association setup to get the benefit of efficiency. It also provides flexible ways to shut down an association, that is, both graceful and forced close are allowed as discussed in chapter 3. The cost of this flexibility is an increase in the complexity, which seems contrary to the intention to gain efficiency. Additionally, to adapt the service interface for this flexibility the close primitives must indicate which path is to be closed, i.e., send or receive. This results in either the number of primitives being increased as presented in chapter 4, or the syntax of the primitives becoming complex as discussed above. From the user's point of view, it is not convenient in either case. For instance, every user needs four interactions with the service provider for either graceful or forced close of an association, as shown in Figure 4.14.

To eliminate the forced close from XTP may be a possible improvement. If so, 12 close primitives proposed in chapter 4 can be simplified as the following primitives with new semantics:

1. CLOSE.request

   This is used to close a SEND path in an XTP association. Note that only the

SEND path, rather than both the SEND path and the RECEIVE path, is to be closed.

2. CLOSE.indication

   This indicates that the remote peer has terminated sending data.

3. CLOSE.response

   This is used to respond to receipt of the CLOSE.indication, and to close its own SEND path. After issuing this primitive, the user will no longer interact with the service provider.

4. CLOSE.confirm

   This indicates that the association has been closed completely, and is issued after all data received from the remote peer have been delivered.

In the case of a crossed close resulting from both users issue a CLOSE.request simultaneously, the service provider will return a CLOSE.confirm to the local user after all data received from the remote peer have been delivered. This implies that CLOSE.indication and CLOSE.response are omitted in this case.

With this improvement, only two interactions between the user and the service provider will be necessary at both endpoints of an association.

Based on the improvement discussed above, the restriction on using forced close may be lifted for some special cases in the following sense. When a receiver indicates forced close, it probably means that the receiving user has expired, the correct procedure is to send the END bit in the protocol. So the issue of forced close can be dealt with by adding an "abort context" primitive.

The second objective of this thesis was to contribute by presenting a formal specification of the XTP service in LOTOS. Chapters 5 and 6 discussed the LOTOS specification for unicast and multicast, respectively.

In the specification, a mixture of constraint-oriented and state-oriented approaches has been used. Based on a constraint-oriented approach, the service decomposed into a set of concurrent processes that focus on a collection of constraints. These constraints, as requirements, filter out only those interactions that may take place at a given gate at each moment. As discussed before, the lifetime of an association in XTP cannot simply be described by three phases as in usual protocols. To deal with this issue the specification has also used a state-oriented approach, in which a process is decomposed into a series of sequential processes. Each of them may be regarded as a state in a final state machine.

The completed specification has successfully passed syntax and static checks, and been verified with the LOTOS utility, called ISLA (Lotos Integrated Static Analyser), developed by the University of Ottawa [Logrip88, Logrip91]. The ISLA utility provides a tool to verify a specification written in LOTOS in a UNIX environment. ISLA consists of a LOTOS translator and an ISLA interpreter. The LOTOS translator is used to analyze the LOTOS specification, syntax and static semantics, and to produce a suitable PROLOG format of the specification for testing. The ISLA interpreter is used to simulate execution of the translated specification, which allows the user to verify the semantic correctness. To verify the specifications, we have simulated more than twenty different situations, in which all types of primitives have been used. In the appendix, a simulation result produced by ISLA is given.

Because LOTOS is a constructive formalism that expresses behaviour explicitly, but cannot explicitly express properties, we cannot directly prove the safety and the liveness properties to verify our specifications. If an utility of non-constructive formalism such as Temporal Logic and a relevant translator to convert LOTOS specifications are available, our specifications can be verified by automatically proving the safety and the liveness properties.

The specification of the XTP service shows that LOTOS is a powerful formal description technique, which has successfully been applied to specification of a complete service interface for a real communication protocol, XTP. The existing XTP document is defective in the service definition. The proposed specification provides a precise and unambiguous service definition for XTP. It is hoped that this formally defined XTP service will help the further development of XTP.

## Future work

At the present stage, the specification of the service is divided into two parts, unicast and multicast. A future improvement on this work would be to combine them. A future extension to this work would be an attempt to specify XTP itself.

A lesson drawn from our practice is that the poor data typing notation and insufficient library in LOTOS result in specifications being diffuse. In addition, it is difficult to express absolute timeout in LOTOS because of being lack of timing mechanisms. Therefore, to enhance LOTOS in the above two aspects will be attractive topics for future research.

# References

[Atwood92] J.W. Atwood. *Validation of the XTP Context Machine,* Technical Report, Department of Computer Science, Concordia University, July, 1992.

[Blyth] D. Blyth, et al.. *Architectural and Behavioural Modelling in Computer Communication,* Proceedings of the IFIP WG 10.3 Working Conference on Distibuted Processing, pp.53-70, 1988.

[Bochm88] G. v. Bochmann. *Specifications of a Simplified Transport Protocol Using Diffident Formal Description Techniques,* Computer Networks and ISDN Systems, Vol. 18, pp.335-377, 1989/1990.

[Bochm90] G. v. Bochmann. *Protocol Specification for OSI,* Computer Networks and ISDN Systems, Vol. 18, pp.167-184, 1989/1990.

[Bolognesi] T. Bolognesi and E. Brinksma. *Introduction to the ISO Specification Language LOTOS,* Computer Networks and ISDN Systems, Vol.14, pp. 25-59, North-Holland, 1987.

[Brinksma] E. Brinksma. *Specification Modules in LOTOS,* in Proceedings of Formal Description Techniques, II, pp.101-115, North-Holland, 1990.

[Ehrig] H. Ehrig. *Fundamentals of Algebraic Specification 1,* Springer-Verlag, Berlin, 1984.

[ISO7498] ISO. *Information Processing Systems - Open Systems Interconnection - Basic Reference Model*, ISO 7498, 1988.

[ISO82] ISO. *Data Processing - Open Systems Interconnection - Service Conventions,* ISO TC97/SC16 N897, Jan. 1982.

[ISO8807] ISO. *Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, ISO 8807, 1989.

[ISO90] ISO. DTR 10167, *Guidelines for the Application of Estelle, LOTOS, and SDL,* ISO TC97/SC21 N4259, Jan. 1990.

[Karjoth] G. Karjoth. *Implementing Process Algebra Specifications by State Machines,* in Proceeding of Protocol Specification, Testing, and Verification, VIII, North-Holland, 1988.

[Leduc] G.J. Leduc. *The Intertwining of Data Types and Processes in LOTOS,* in Proceedings of Protocol Specification, Testing, and Verification, VII, pp.123-136, North-Holland, 1987.

[Logrip88] L. Logrippo, et al. *An Interpreter for LOTOS, a Specification Language for Distributed Systems,* Software-Practice and Experience, Vol. 18, pp. 365-385, 1988.

[Logrip91] L. Logrippo. *The University of Ottawa LOTOS Toolkit,* Formal Description Techniques, III, pp. 563-566, 1991.

[Madel] E. Madelaine and D. Vergamini. *Specification and Verification of a Sliding Window Protocol in LOTOS,* Formal Description Techniques, IV, pp. 495-510, 1992.

[Milner] R.Milner. *A Calculus of Communicating Systems,* Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, 1980.

[Murphy] S.L. Murphy and A.U. Shankar. *Connection Management for the Transport Layer: Service Specification and Protocol Verification,* IEEE Transactions on Communications, Vol. 39, No. 12, pp. 1762-1775, Dec. 1991.

[Navarro] J. Navarro and P.S. Martin. *Experience in the Development of an ISDN Layer 3 Service in LOTOS,* Formal Description Techniques, III, pp. 327-336, 1991.

[Santoso] H. Santoso and S. Fdida. *Transport Layer Multicast: An Enhancement for XTP Bucket Error Control,* in Proceedings of High Performance Networking 92, pp. G2-17, 1992.

[Turner] K.J. Turner. *An Architectual Semantics for LOTOS,* in Proceedings of Protocol Specification, Testing, and Verification, VII, pp.15-28, North-Holland, 1987.

[Vissers] C.A. Vissers and L. Logrippo. *The importance of the Service Concept in the Design of Data Communication Protocols,* in Proceeding of Protocol Specification, Testing, and Verification, V, pp. 3-17, North-Holland, 1986.

[Vissers] C.A. Vissers, et al. *Architecture and Specification Style in Formal Descriptions of Distributed Systems,* in Proceeding of Protocol Specification, Testing, and Verification, VIII, pp. 189-204, North-Holland, 1988.

[XTP92] *XTP protocol Definition, Revision 3.6,* XTP Forum. 1900 State Street, Santa Barara CA 93101, Jan. 1992.

# Appendix A

# A simulation result

The following is a result to verify the specification of the XTP unicast service with ISLA. For convenience, the gates XTPS, NI and NR are relablled as H, A and B, respectively. In this simulation, there are 35 events occurred at these gates. These events simulate a typical situation, in which the behaviors between the XTP service provider and the users for setting up an association, sending and delivering data, and terminating the association are included.

The result shown is the script of ISLA's output on the screen. For the purpose of brevity, only the outputs related to the first two events and the last event are presented. However, the omitted events can be found in the history of events listed in the output related to the final event.

In the simulation, some predefined constants used are as follows.

| | |
|---|---|
| \$u00, \$u11 | user indexes (SAPs); |
| \$k00, \$k11 | key indexes (endpoints); |
| \$a0 | an address-segment used in the primitve open_request; |
| \$op1 | options used in the primitive open_request; |
| \$ba01 | a buffer address; |
| \$bs1 | buffer size; |
| \$b01, \$b02, | |
| \$b10, \$b15 | user buffers; |
| \$sp0 | a primitive open_request with pararmeters \$a0, \$op1, and \$b01; |
| \$sp1 | a primitive listen_request; |
| \$swc | a primitive send_request with the flag Wclose; |
| \$rrc | a primitive receive_request with the flag Rclose; |
| \$r10 | a primitive receive_request. |

ISLA
Interactive System for LOTOS Applications

```
=================== xtpservice ========================
CP/MANUAL  IW/100  ED/50  IA/skip  FG/skip  Level/0
Path/[]
Events:


================================================
No Internal actions

<1>- H ?ui,ui,uiuser_index ?key,key,keykey_index ?sp2,sp1,spxtpsp [and(Is_open_request(sp1),
Is_conn(s_service(sp1)))] [eq(hostaddr(ui),NA(0))] —> bh1 [1121,1098,1054]
<2>- H ?ui,ui,uiuser_index ?key,key,keykey_index ?sp2,sp1,spxtpsp [and(Is_listen_request(sp1),
Is_conn(s_service(sp1)))] [eq(hostaddr(ui),NA(1))] —> bh2 [1137,1098,1054]

================================================

ACT: la[ctions][<N>] | ? | h[elp] | <action number> | <command>
==> 1
The existing predicate(s) for this action : and(Is_open_request(sp1), Is_conn(s_service(sp1)))]
[eq(hostaddr(ui),NA(0))]

Enter a set or a value for ui,ui,uiuser_index
=> $u00
Enter a set or a value for key,key,keykey_index
=> <>
Enter a set or a value for sp2,sp1,spxtpsp
=> $sp0
Evaluating Predicate(s) ...
Predicate evaluated to true
Applying inferences rules !
```

ISLA
Interactive System for LOTOS Applications

```
==================== xtpservice ========================
CP/MANUAL  IW/100  ED/50  IA/skip  FG/skip  Level/1
Path/[1]
Events:
[1] H ?$u00:user_index ?<>:key_index ?$sp0:xtpsp

=====================================================
No Internal actions

<1>- value for k:key_index is needed for 'choice' —> bh1 [1124]
<2>- H ?ui,ui,uiuser_index ?key,key,keykey_index ?sp2,sp1,spxtpsp [and(Is_listen_request(sp1),
Is_conn(s_service(sp1)))] [eq(hostaddr(ui), NA(1))] —> bh2 [1137,1098,1054]

=====================================================

ACT: la[ctions][<N>] | ? | h[elp] | <action number> | <command>
==> 1
```

```
                         ISLA
          Interactive System for LOTOS Applications

==================== xtpservice =========================
CP/MANUAL  IW/100  ED/50  IA/skip  FG/skip  Level/35
Path/[1,1,1,2,1,2,2,2,2,5,2,2,8,1,17,7,13,8,8,15,10,8,9,2,2,2,7,2,9,1,2,2,2,4,2]
Events:
[1] H ?$u00:user_index ?<>:key_index ?$sp0:xtpsp
[1, 1] choice(k:key_index = $k00)
[1, 1, 1] H !$u00:user_index !$k00:key_index !open_confirm($k00, succ_code):xtpsp
[1, 1, 1, 2] A !$k00:key_index !$sp0:xtpsp
[1, 1, 1, 2, 1] H !$u00:user_index !$k00:key_index ?$swc:xtpsp
[1, 1, 1, 2, 1, 2] A !$k00:key_index !$swc:xtpsp
[1, 1, 1, 2, 1, 2, 2] H ?$u11:user_index ?<>:key_index ?$sp1:xtpsp
[1, 1, 1, 2, 1, 2, 2, 2] choice(k:key_index = $k11)
[1, 1, 1, 2, 1, 2, 2, 2, 2] H !$u11:user_index !$k11:key_index !listen_confirm($k11, succ_code):xtpsp
[1, 1, 1, 2, 1, 2, 2, 2, 2, 5] B !$k11:key_index !object($sp0):Object
[1, 1, 1, 2, 1, 2, 2, 2, 2, 5, 2] H !$u11:user_index !$k11:key_index !association_indication($a0,$op1):
xtpsp
[1, 1, 1, 2, 1, 2, 2, 2, 2, 5, 2, 2] B !$k11:key_index !association_response($a1, $op1, succ_code):xtpsp
[1, 1, 1, 2, 1, 2, 2, 2, 2, 5, 2, 2, 8] A !$k00:key_index !object(association_response($a1, $op1,
succ_code)):Object
[1, 1, 1, 2, 1, 2, 2, 2, 2, 5, 2, 2, 8, 1] H !$u00:user_index !$k00:key_index !association_confirm($a1,
$op1, succ_code):xtpsp
[1, 1, 1, 2, 1, 2, 2, 2, 2, 5, 2, 2, 8, 1, 17] B !$k11:key_index !object($swc):Object
[1, 1, 1, 2, 1, 2, 2, 2, 2, 5, 2, 2, 8, 1, 17, 7] H !$u11:user_index !$k11:key_index !close_send_indication:
xtpsp
[1, 1, 1, 2, 1, 2, 2, 2, 2, 5, 2, 2, 8, 1, 17, 7, 13] H !$u11:user_index !$k11:key_index ?$rrc:xtpsp
[1, 1, 1, 2, 1, 2, 2, 2, 2, 5, 2, 2, 8, 1, 17, 7, 13, 8] H !$u11:user_index !$k11:key_index
!receive_confirm($b15, succ_code):xtpsp
[1, 1, 1, 2, 1, 2, 2, 2, 2, 5, 2, 2, 8, 1, 17, 7, 13, 8, 8] B !$k11:key_index !receive_confirm($b15,
succ_code):xtpsp
[1, 1, 1, 2, 1, 2, 2, 2, 2, 5, 2, 2, 8, 1, 17, 7, 13, 8, 8, 15] B !$k11:key_index !close_send_response:xtpsp
[1, 1, 1, 2, 1, 2, 2, 2, 2, 5, 2, 2, 8, 1, 17, 7, 13, 8, 8, 15, 10] H !$u11:user_index !$k11:key_index
?close_send_request:xtpsp
[1, 1, 1, 2, 1, 2, 2, 2, 2, 5, 2, 2, 8, 1, 17, 7, 13, 8, 8, 15, 10, 8] A !$k00:key_index !object(receive_confirm
($b15, succ_code)):Object
[1, 1, 1, 2, 1, 2, 2, 2, 2, 5, 2, 2, 8, 1, 17, 7, 13, 8, 8, 15, 10, 8, 9] A !$k00:key_index !object(close_send_
response):Object
```

```
[1, 1, 1, 2, 1, 2, 2, 2, 2, 5, 2, 2, 8, 1, 17, 7, 13, 8, 8, 15, 10, 8, 9, 2] H !$u00:user_index !$k00:key_index
!close_send_confirm:xtpsp
[1, 1, 1, 2, 1, 2, 2, 2, 2, 5, 2, 2, 8, 1, 17, 7, 13, 8, 8, 15, 10, 8, 9, 2, 2] H !$u00:user_index !$k00:
key_index !send_confirm(u_buffer($ba01, $bs1), succ_code):xtpsp
[1, 1, 1, 2, 1, 2, 2, 2, 2, 5, 2, 2, 8, 1, 17, 7, 13, 8, 8, 15, 10, 8, 9, 2, 2, 2] H !$u00:user_index !$k00:
key_index !send_confirm($b02, fail_code):xtpsp
[1, 1, 1, 2, 1, 2, 2, 2, 2, 5, 2, 2, 8, 1, 17, 7, 13, 8, 8, 15, 10, 8, 9, 2, 2, 2, 7] B !$k11:key_index
!close_send_request:xtpsp
[1, 1, 1, 2, 1, 2, 2, 2, 2, 5, 2, 2, 8, 1, 17, 7, 13, 8, 8, 15, 10, 8, 9, 2, 2, 2, 7, 2] H !$u00:user_index
!$k00: key_index ?$r10:xtpsp
[1, 1, 1, 2, 1, 2, 2, 2, 2, 5, 2, 2, 8, 1, 17, 7, 13, 8, 8, 15, 10, 8, 9, 2, 2, 2, 7, 2, 9] A !$k00:key_index
!object (close_send_request):Object
[1, 1, 1, 2, 1, 2, 2, 2, 2, 5, 2, 2, 8, 1, 17, 7, 13, 8, 8, 15, 10, 8, 9, 2, 2, 2, 7, 2, 9, 1] H !$u00:user_index
!$k00:key_index !close_send_indication :xtpsp
[1, 1, 1, 2, 1, 2, 2, 2, 2, 5, 2, 2, 8, 1, 17, 7, 13, 8, 8, 15, 10, 8, 9, 2, 2, 2, 7, 2, 9, 1, 2] H !$u00:user_index
!$k00:key_index ?close_send_response:xtpsp
[1, 1, 1, 2, 1, 2, 2, 2, 2, 5, 2, 2, 8, 1, 17, 7, 13, 8, 8, 15, 10, 8, 9, 2, 2, 2, 7, 2, 9, 1, 2, 2] H !$u00:user_index
!$k00:key_index receive_confirm($b10, fail_code):xtpsp
[1, 1, 1, 2, 1, 2, 2, 2, 2, 5, 2, 2, 8, 1, 17, 7, 13, 8, 8, 15, 10, 8, 9, 2, 2, 2, 7, 2, 9, 1, 2, 2, 2] A !$k00:key_index
!close_send_response:xtpsp
[1, 1, 1, 2, 1, 2, 2, 2, 2, 5, 2, 2, 8, 1, 17, 7, 13, 8, 8, 15, 10, 8, 9, 2, 2, 2, 7, 2, 9, 1, 2, 2, 2, 4]
B !$k11:key_index !object(close_send_response):Object
[1, 1, 1, 2, 1, 2, 2, 2, 2, 5, 2, 2, 8, 1, 17, 7, 13, 8, 8, 15, 10, 8, 9, 2, 2, 2, 7, 2, 9, 1, 2, 2, 2, 4, 2]
H !$u11:user_index !$k11:key_index !close_send_confirm :xtpsp
=================================================
No Internal actions

<1>- H ?ui,ui,uiuser_index ?key,key,keykey_index ?sp2,sp1,spxtpsp [and(Is_open_request(sp1),
Is_conn(s_service(sp1)))] [eq(hostaddr(ui), NA(0))] —> bh1 [1121,1098,1054]
<2>- H ?ui,ui,uiuser_index ?key,key,keykey_index ?sp2,sp1,spxtpsp [and(Is_listen_request(sp1),
Is_conn(s_service(sp1)))] [eq(hostaddr(ui), NA(1))] —> bh2 [1137,1098,1054]

=================================================

ACT: la[ctions][<N>] | ? | h[elp] | <action number> | <command>
==> exit
EXIT ISLA ? (n/y) => y

Bye Bye!!
```