INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

NOTE TO USERS

This reproduction is the best copy available.

UMI®

SPECIFICATION AND VALIDATION OF THE COMMON SIGNALING TRANSPORT PROTOCOL IN SDL

XINGGUO SONG

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

SEPTEMBER 2002
© XINGGUO SONG, 2002

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

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-72944-3

Canada

Abstract

Specification and Validation of the Common Signaling Transport Protocol in SDL

Xingguo Song

The Resource reSerVation Protocol (RSVP) version 1 is the dominant Internet protocol for signaling Quality of Service (QoS) requirements. It has been extended for use in a wide variety of Internet signaling applications, such as traffic engineering and label distribution. However, the multiple features of the extended RSVP increase its complexity, and interactions among the features could cause confusion. In addition, due to the constraints of the original architecture, it is difficult to specify requirements for new Internet applications, such as mobile IP.

The Internet Engineering Task Force has mandated discussion of a new Internet Signaling Protocol Suite (ISPS). One proposal coming from this discussion is the Internet Draft "A Two-level Architecture for Internet Signaling". It proposes a Common Signaling Transport Protocol (CSTP), concentrating on state management and reliable data delivery, coupled with separate Application Layer Signaling Protocols, which implement the features of individual signaling applications.

Starting from the English specification of CSTP in the Internet Draft, a specification of CSTP has been written in the formal specification language SDL, and validated for several scenarios, which were based on the typical operation of RSVP version 1. Several errors have been found in the original specification, and solutions to these problems have been proposed.

Acknowledgements

First of all, I would like to take this chance to thank my supervisor Dr. J.W.Atwood, who guided me to research the secrets of the Internet. His serious academic characteristics impressed me very much. His valuable advice and practical guidance greatly helped me to finish my thesis successfully. I have learned some valuable virtues as well as research methods from him. From now on, the Internet will not be strange for me. I like to live and work in the fantastic Internet world!

I also would like to thank Dr. Ferhat Khendek, who provided me with a powerful tool ObjectGEODE, so that I could finish my thesis efficiently.

I will always remember the beneficial discussions with Mr. Xin Shen, who gave me many useful suggestions. I would like to express my great thanks to him.

A special thanks is devoted to Ms. Halina Monkiewicz who provided me with extensive miscellaneous help, and Mr. Stan Swierz who maintained ObjectGEODE.

Moreover, I would like to extend my thanks to all those people, who ever helped me.

Finally, I would like to give great thanks to my whole family, especially to my parents. May they always have health and happiness.

Contents

List of Tables x

List of Figures xi

List of Acronyms xiii

1 Introduction 1

1.1 Signaling, QoS, QoS Signaling and RSVP 1

1.2 A Two-level Architecture for Internet Signaling 3

1.3 Motivation and Scope of the Thesis 3

1.4 Organization of Contents 4

2 A Two-level Architecture for Internet Signaling 6

2.1 Overview 6

2.2 Rationale of a Two-level Architecture for Internet Signaling 7

2.2.1 Introduction to RSVP Version 1 and Its Extensions 7

2.2.2 Limitation of RSVP Version 1 10

2.2.3 QoS Signaling Requirements 11

2.2.4 Possibility of an ISPS Based on RSVP Version1 and Its Extensions 12

2.3 Common Signaling Transport Protocol (CSTP) 12

2.3.1 Major Requirements 12

2.3.2 CSTP Messages 14

2.3.3 CSTP Mechanism 17

2.4 Application-Layer Signaling Protocols (ALSPs) 23

2.5 The Interface between CSTP and ALSPs 24

2.6 An Example of an ISPS Offering the Simplest Unicast Resource Reservation Services of RSVP Version 1 25

3 Introduction to SDL and MSC **32**

3.1 SDL 32

3.1.1 History 32

3.1.2 Characteristics 33

3.1.3 SDL Model Components 34

3.1.4 Object-Oriented Characteristics 43

3.2 MSC 44

4 A CSTP Model **46**

4.1 Model Requirements 46

4.1.1 Assumptions of Model Environment 46

4.1.2 Model Requirements 47

4.2 System Model Architecture 47

4.2.1 Model Description 47

4.2.2 Satisfaction of the Model Requirements 48

4.3 CSTP Module 49

4.3.1 Soft State Management 49

4.3.2 Soft State Blocks 54

4.3.3 Transmission Processes 59

4.4 ALSP Module 61

4.5 Lower_layer Module 66

4.6 Modeling Constraints 66

4.6.1 Probabilistic Decisions 66

4.6.2 Timing 68

4.6.3 Resource Specified 68

4.6.4 Signal Priority 68

5 Validation of CSTP **69**

5.1 Tasks and Techniques 69

5.2 Validation of Set One Scenarios of CSTP 71

5.2.1 Scenario: *Send New SAPU* 71

5.2.2 Scenario: *Send Mod SAPU* 75

5.2.3 Scenario: *Send Tear SAPU* 78

5.2.4 Scenario: *Send Event SAPU* 80

5.3 Validation of Set Two Scenarios of CSTP 84

5.4 Validation Results 88

5.4.1 Validation of Set One Scenarios of CSTP 88

5.4.2 Validation of Set Two Scenarios of CSTP 89

6 Discussion **90**

6.1 CSTP Design Faults 90

6.1.1 CSTP Bundling Message Definition 90

6.1.2 Session Distinction 91

6.1.3 The Unexpected Signal xSig(NACK) 92

6.1.4 Interface Calls 92

6.1.5 Hop-by-Hop Refreshment Mechanism 94

6.2 Suggestions for CSTP Design 95

6.2.1 Adding a SAPUid in the Signature of RecvNewSAPU and RecvModSAPU 95

6.2.2 Sending an xSig(EVENT) at an Intermediate Node 95

6.2.3 Adding a Context for the Hsrc State Transit Diagram 96

6.2.4 Denoting the Generation of a Modified SAPUid in SendMod-SAPU Clearly 96

7 Conclusion 98

7.1 Conclusion of Work 98

7.2 Contributions 99

7.3 Future Work 99

List of Tables

1 Predefined Sorts in SDL 41

2 The Table of CSTP Messages and ALSPs/CSTP Interface Calls . . . 93

List of Figures

1 A Two-level Architecture for Internet Signaling 7

2 B-header Structure 15

3 Challenge Object 16

4 CSTP Pair Behaviors 17

5 H-Src CSTP Trigger Messages State Transit Diagram 18

6 H-Sink CSTP Trigger Messages State Transit Diagram 19

7 Resource Reservation in RSVP Version 1 25

8 Resource Reservation in The Two-level Architecture 28

9 An SDL System Structure 34

10 Basic SDL Legend 35

11 A Remote Procedure Call 36

12 Process Identifier PId 39

13 A bMSC example 45

14 A CSTP Model 48

15 The CSTP Module 50

16 H-Src CSTP Event Messages State Transit Diagram 60

17 H-Sink CSTP Event Messages State Transit Diagram 60

18 The ALSP Module 62

19 The Lower_layer Module 67

20 A Successful Scenario for Scenario *Send New SAPU* 73

21 A Failure Scenario for Scenario *Send New SAPU* 74

22 A Success Scenario for Scenario *Send Mod SAPU* 76

23 A Failure Scenario for Scenario *Send Mod SAPU* 77

24 A Successful Scenario for Scenario *Send Tear SAPU* 79

25 A Failure Scenario for Scenario *Send Tear SAPU*: SendFail 80

26 A Failure Scenario for Scenario *Send Tear SAPU*: State Life Timer Time-Out 81

27 A Successful Scenario for Scenario *Send Event SAPU* 82

28 A Failure Scenario for Scenario *Send Event SAPU* 83

29 A Successful Scenario for Scenario *Offering Unicast Reservation Features of RSVP Version 1 by CSTP* 86

30 A Failure Scenario for Scenario *Offering Unicast Reservation Features of RSVP Version 1 by CSTP* 87

List of Acronyms

2LAIS ....Two-Level Architecture for Internet Signaling

ADT ....Abstract Data Type

ALSP ....Application-Layer Signaling Protocol

bMSCs ....basic Message Sequence Charts

BNF ....Backus-Naur Form

CSTP ....Common Signaling Transport Protocol

EFSMs ....Extended Finite State Machines

ERSSB ....Event Receiving Soft State Block

ESSSB ....Event Sending Soft State Block

hMSCs ....high level Message Sequence Charts

IETF ....the Internet Engineering Task Force

ISPS ....Internet Signaling Protocol Suite

MSC ....Message Sequence Charts

MTU ....Maximum Transmission Unit

NSIS ....Next Steps In Signaling

PHOP ....Previous HOP

PId ....Process Identifier

QoS ....Quality of Service

QSCs ....Qos Service Classes

RSVP ....Resource reSerVation Protocol

SAPU ....Signaling Application Protocol Unit

SDL ....Specification and Description Language

TRSSB ......Trigger Receiving Soft State Block

TSSSB ....Trigger Sending Soft State Block

WG ....Working Group

Chapter 1

Introduction

1.1 Signaling, QoS, QoS Signaling and RSVP

The word 'Signaling' comes from telephony. In telephony, signaling is the exchange of information between involved points in the network that sets up, controls, and terminates each telephone call. Now this word is broadly used in Internet protocols to denote the setup, control and termination of data transmission sessions among hosts and routers in the Internet domain.

Quality of Service (QoS) is defined in ITU-T E800 [14] as the collection of service performances which ensures the degree of satisfaction of a user of the service. In the Internet, there are two basic architectures to realize IP QoS: Integrated Service (IntServ) and Differentiated Service (DiffServ). Also, there are some other IP QoS solutions extended from these two basic ones.

According to Quittek [25], QoS Signaling is a way to communicate QoS Service Classes (QSCs) and QoS management information between hosts, end systems and network devices, etc. It may include request and response messages to facilitate negotiation/re-negotiation, asynchronous feedback messages (not delivered upon request) to inform End Hosts, QoS initiators and QoS controllers about current QoS levels and QoS querying facilities.

The Resource Reservation Protocol (RSVP) is one of the prominent signaling protocols. It has been adopted within the IntServ architecture for resource reservation. In 1991, Dr. Lixia Zhang first proposed the concept, RSVP [9]. So far, centered on RSVP, there are many interactions between the RSVP Working Group (WG) and eight other WGs within IETF. Moreover, there are 136 Internet Drafts (IDs) with the titles that include the word RSVP published by the IETF. Some of them have become RFCs, such as RFC2205 [8], RFC2961 [5]. Currently people treat RFC2205 and RFC2961 as the core part of RSVP version 1. RSVP version 1 and its extensions introduce many features, ranging over transport, routing, soft-state mechanisms and strongly-typed encoding [6] [29] [2] [26] [10] [22]. Obviously, all those RSVP extensions can form a multi-function Internet Signaling Protocol Suite (ISPS). In industry, many enterprises have supported RSVP in their products [17].

Following extensive experience with RSVP, many limitations or defaults of RSVP version 1, summarized in two aspects of signaling and QoS, have been disclosed. In the aspect of signaling, for example, it has a lack of scalability. Due to the fact that QoS signaling is on a per-flow basis, a large number of states is needed at the core of the network. Also, as RSVP is a generic signaling protocol, multi-featured RSVP results in implementation complexity, such as handling multicast reservations, as well as conversational applications. Those multiple features also cause confusion because of feature interactions. Moreover, it can not interwork well with new Internet applications, such as Mobile IP in the aspect of address switching [11]. In the aspect of QoS, for example, RSVP carrying IntServ QoS parameters can not adapt to all QoS architectures' needs in the Internet domain. Therefore, RSVP version 1 needs to be improved or a new QoS signaling protocol needs to be invented.

A generic QoS Signaling protocol is needed in the real world. Thus, a new IETF WG named "Next Steps in Signaling" was set up in 2001. Its task is to develop the requirements, architecture and protocols for the next IETF steps on QoS Signaling. Its starting point is to evaluate RSVP. NSIS's framework covers the use of a simplified version of RSVP, and the potential for an Internet signaling toolbox or building block contribution [30].

1.2 A Two-level Architecture for Internet Signaling

Based on RFC2205, RFC2961, RFC1191 [13] as well as practice of RSVP, Mr. Braden, the first author of RSVP version 1, proposed a two-level architecture for Internet signaling, in his Internet Draft (ID) [7]. The two levels are a Common Signaling Transport Protocol (CSTP) at the lower level, and various Application-Layer Signaling Protocols (ALSPs) at the upper level. CSTP features mainly include reliable delivery and soft state management, while each ALSP is to implement a specific signaling application. The main purpose of this architecture is to propose an Internet Signaling Protocol Suite (ISPS) with improved modularity for both wired and wireless applications.

According to his ID, one could treat the design of CSTP and ALSPs as an RSVP version 2 in some sense, though CSTP is designed more generally than a strictly RSVP-like protocol. We will abbreviate "Two-level Architecture for Internet Signaling" as 2LAIS in this document. Obviously, 2LAIS fits in with the task of the NSIS WG. Hopefully this idea could fully take advantage of RSVP version 1 and greatly improve it for a wider application range. Without doubt, 2LAIS is valuable, from the academy to the industry.

1.3 Motivation and Scope of the Thesis

To setup an Internet Signaling Suite is a long term project. Currently, work on the design of CSTP is just at the very beginning. In the ID of 2LAIS, Mr. Braden only gave a general idea or framework. According to Mr. Braden, his ID had not been simulated or prototyped. Thus, our main purpose on the thesis is to find by simulation CSTP design errors and discrepancies; to explore CSTP scenarios; to propose suggestions on CSTP design; to give an SDL model for later research use; and to gain confidence that ISPS can offer the features of RSVP version 1.

The work consists of four parts:

- Explaining the original ID, and noting necessary corrections
- Writing a CSTP specification in SDL
- Validating the CSTP scenarios with the ObjectGEODE simulator
- Simulating the simplest resource reservation feature of RSVP version 1 by simulating the combination of CSTP and an ALSP driver

1.4 Organization of Contents

The thesis includes seven chapters.

Chapter 1 gives a conceptual introduction related to the thesis context.

Chapter 2 gives an overview of this two-level signaling architecture. First, the rationale for this research is explained. Then CSTP, ALSP and the interface between CSTP and ALSPs are introduced. Finally, that the cooperation of ALSPs and CSTP can offer features of RSVP version 1 is explained through an example.

Chapter 3 introduces the specification languages SDL and MSC. It is expected that the introduction could help readers to understand our work.

Chapter 4 gives a CSTP model. First, the CSTP requirements in this simulation are addressed. Then, the model architecture is introduced. Followingly, the reasons why the model can satisfy the CSTP requirements are given. After that, the model is introduced in detail. Finally, the model constraints are discussed.

Chapter 5 discusses the validation of the scenarios. First, the tasks and techniques in the validation are discussed. Then, various scenarios in the validation are discussed. Finally, the simulation results are given.

Chapter 6 discusses the problems in CSTP design, based on the thesis work. Also, some suggestions for the CSTP design are given.

Chapter 7 makes conclusions on the thesis work and our contributions, and the future work is indicated.

Chapter 2

A Two-level Architecture for Internet Signaling

2.1 Overview

The main purpose of the 2LAIS design is to guide the implementation of ISPS. Figure 1 depicts the two-level architecture. CSTP is at the lower level, and a group of ALSPs lies at the upper level. A set of standard interface calls connects these two levels together. The CSTP and the ALSPs together form the ISPS.

CSTP mainly takes charge of reliable delivery of Signaling Application Protocol Unit (SAPU) and maintains the soft states by sending refreshing messages. As a control center, an ALSP is an implementation of a specific signaling application, which is in charge of signaling control and other auxiliary services. The virtue of this architecture is to lessen the complexity, reorganize the feature interactions, as well as to be open to new signaling application requirements. To reuse the current RSVP version 1 implementations, a gateway is suggested to bridge them with ISPS.

In the following sections of this chapter, we will explain the reasons to choose this topic, and the details of CSTP and ALSPs. The specification in the ID is just a framework; it is incomplete and includes errors. Thus, we develop CSTP senarios in

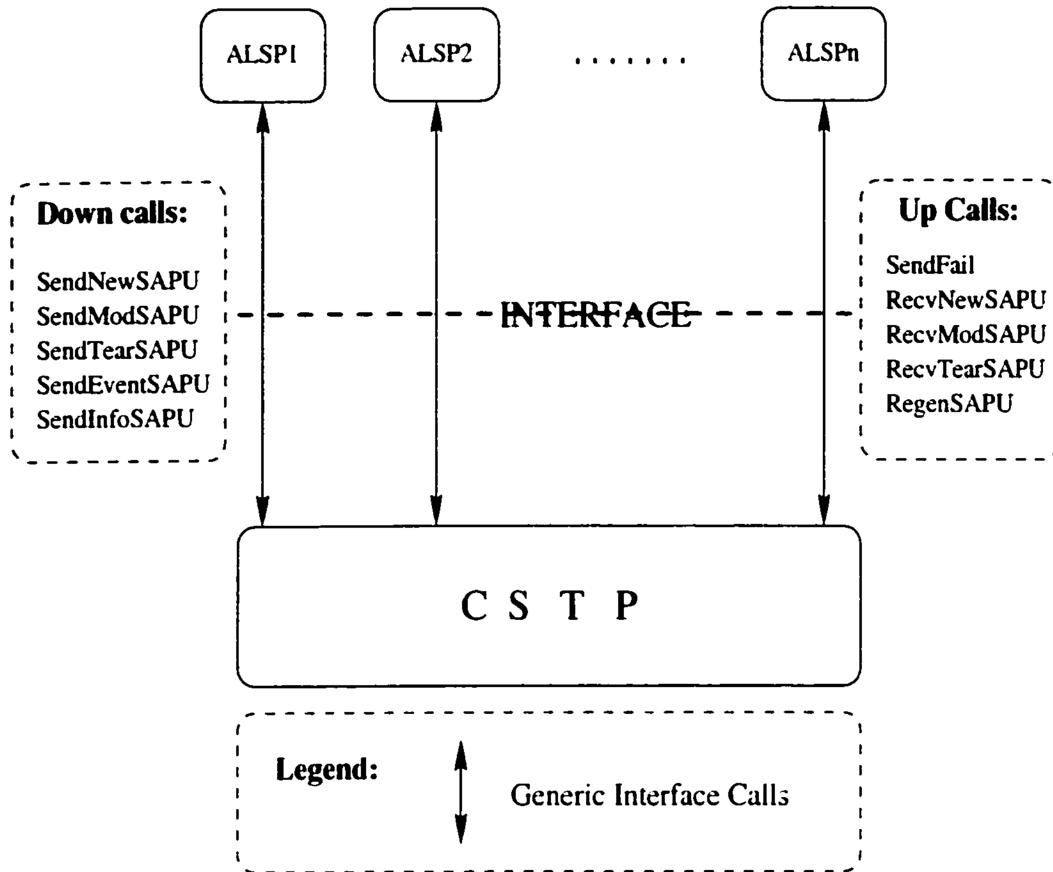


Figure 1: A Two-level Architecture for Internet Signaling

detail, and also we give a revised CSTP definition in Backus-Naur Form (BNF).

2.2 Rationale of a Two-level Architecture for Internet Signaling

2.2.1 Introduction to RSVP Version 1 and Its Extensions

RSVP version 1 is a setup protocol for Internet resource reservation. It is used to create flow-specific resource reservation states in routers and hosts along an end-to-end path. Some applications request resources through RSVP with IntServ parameters.

Though RSVP could carry IntServ Information, RSVP is separable from IntServ. The model of IntServ/RSVP is a currently prevailing model to realize QoS services in edge networks.

Generally, RSVP version 1 has the following functions:

- support End-to-End resource reservation, for unicast and multicast.
- May carry IntServ QoS parameters.
- Maintain path and reservation soft states.
- Support uni-directional reservation
- Merge and share multiple flows
- Be separate from routing functions, depending on local routing queries to get route information.
- Carry policy data to realize policy control.
- Integrity.

So far, RSVP WG has set up a direct relationship with the following IETF WGs:

- IntServ: RSVP carries IntServ QoS parameters to all nodes along a path.
- DiffServ: RSVP brings Admission Control and Resource Allocation in DiffServ Architecture.
- ISSLL: ISSLL maps RSVP and IntServ into a specific layer.
- MPLS: RSVP-TE sets up LSPs for MPLS and GMPLS. Also, RSVP-TE extensions try to manage Resilience in MPLS.
- RAP: RAP will continue to document changes to COPS objects needed to support any extensions to RSVP.
- MobileIP: QoS in the mobile IP environment uses DiffServ and/or IntServ/RSVP.

- TEWG: RSVP-TE in MPLS implements traffic engineering.
- NSIS: The working group particularly put the evaluation of RSVP as its starting point. Its framework covers using a simplified version of RSVP, building a signaling tool box or blocks, and maybe designing a new version RSVP.

From history, there are about 136 IDs with titles that contain the word 'RSVP' published by the IETF. Some of them have become RFCs. Moreover, we cannot quantify those IDs with titles that do not include the word 'RSVP'. Though many IDs had not been converted into RFCs, most of them still can give us some helpful suggestions today.

Since RSVP version 1 was published, many RSVP version 1 extensions have been created. Basically they are extended in two ways: adding one or more new objects/messages to deal with new functions; and changing message processing rules [16]. All the extensions use RSVP core functions which are reservation and soft state maintenance.

In practice, RSVP version 1 and its extensions cover the following major application fields:

- QoS setup across DiffServ clouds [6]
- Setting up MPLS paths with QoS [29]
- Provisioning VPNs [2]
- Configuring optical networks [26]
- QoS setup for PacketCable [10]
- Setup Multicast LSP Tunnels [23]
- Extensions to Policy Control [18]
- Local Protection/Fast Reroute [24]
- Support Mobile IP version 6 in wireless environment [12]

2.2.2 Limitation of RSVP Version 1

In practice, more and more RSVP limitations are disclosed:

- Scalability. Per-flow based RSVP suffers severely scalability problems in a large scale network.
- Complexity. RSVP version 1 offers more generic functions. In the real world, most QoS sensitive applications do not use the full multicast capabilities offered in RSVP. For example, the multicast research community only focuses on finding solutions to one-to-many rather than many-to-many in RSVP. Without the support for many-to-many reservations, the implementation complexity of RSVP will be reduced.
- Uni-direction. RSVP has been designed to reserve resources only in one direction for unicast and multicast. Though this is sufficient for streaming applications, it does not satisfy the requirements of conversational applications, which need bi-directional reservations with minimum delay.
- End-to-End vs. Edge-to-Edge. RSVP is an end-to-end protocol, but it is not clearly defined to request QoS from edge-to-edge without involving end nodes. In the core network, the RSVP-TE has similar requirements.
- Mobility. RSVP works over IP, therefore the RSVP message is opaque to IP. Theoretically, RSVP could be used in IP environments including Mobile IP. However, supporting reservation well while handing-off between two cells is a big problem with RSVP due to the characteristics of mobility.
- QoS. The prevailing model of RSVP/IntServ is widely used in the Internet society. With the model, RSVP carries Intserv QoS parameters traversing Internet work domains. However, the IntServ QoS specification is not sufficient for a variety of applications. For instance, wireless QoS requirements may need acceptable error ratios that cannot be specified in the IntServ QoS specification. When processing RSVP packets at an edge point between two domains, it is sometime a problem to map QoS parameters between the RSVP specifications and the paradigm used in each domain.

Obviously, RSVP is a prominent signaling protocol; however, its limitations restrict its applications. Therefore, the issue to improve RSVP for the more generic usage is an important research topic in the Internet domain.

2.2.3 QoS Signaling Requirements

The IETF has noticed the above problems. The newly set up NSIS WG is working on this issue. Its working strategy is to start by evaluating the current existing signaling protocols, i.e., RSVP, to set up and define requirements, and to develop new QoS signaling protocols. The NSIS working document *Requirements for QoS Signaling Protocols* [28] gives the general requirements on QoS signaling protocol on the aspects of architecture, design goals, signaling flows, additional information beyond signaling of QoS information, layering, QoS control information, performance, flexibility, security, mobility and interworking with other protocols and techniques. It indicates, "Two general (and potentially contradictory) goals for the solution are that it should be applicable in a very wide range of scenarios, and at the same time lightweight in implementation complexity and resource requirements in nodes."

Basically, the Requirements [28] document specifies the need for:

- Generality. QoS signaling protocol should be applied to more applications.
- QoS on request. When requested on QoS, the signaling protocol should offer QoS.
- Modularity. QoS signaling protocol should offer different functions based on modularity, selected by applications.
- Decoupling Signaling and QoS in QoS signaling protocol to enhance interoperability between QoS and signaling.
- Reuse of existing protocols.
- Work both in end-to-end and edge-to-edge scenarios.
- Scalability.

- Uni/bi-directional reservation. Both uni-directional and bi-directional reservations must be possible.
- Security.

2.2.4 Possibility of an ISPS Based on RSVP Version1 and Its Extensions

2LAIS plans to construct two levels: the lower level is going to take charge of reliable delivery of messages as well as soft state maintenance, while the upper level is going to be made up of N ($N \geq 1$) modules and each module has an independent service for different tasks.

From the idea of 2LAIS, it is possible for 2LAIS to make use of existing RSVP version 1 and its extensions. At the CSTP level, CSTP borrows and develops the concepts from RFC2961. At the ALSP level, since RSVP version 1 and its extensions have multi-features traversing the Internet domain, those features could be reorganized into each ALSP. Therefore, 2LAIS can possibly be designed based on existing RSVP version 1 and its extensions.

2.3 Common Signaling Transport Protocol (CSTP)

2.3.1 Major Requirements

According to the ID, CSTP should satisfy the following requirements:

- Support for Path-Directed Signaling. This requires that CSTP must install, modify, and remove states in routers and other entities along the path of some particular data flow. CSTP does not have end-to-end semantics. CSTP handles only the dynamics of reliably transmitting signaling state between neighbours, and of refreshing this as soft states. This requirement assures that CSTP

supports simplex or full-duplex signaling, and CSTP will support receiver- or sender-initiated signaling.

- RSVP Version 1 Support. The combination of CSTP with an appropriate ALSP must support the functionality of any flavour of RSVP Version 1. The CSTP design would not directly interoperate with RSVP Version 1, due to differing packet formats. However, a signaling gateway could be developed to translate RSVP Version 1 signaling messages to and from (CSTP, ALSP) messages.

- Reliable Delivery of Signaling Messages. CSTP must provide reliable delivery of trigger messages so that states can be reliably and promptly added, changed, and explicitly removed. CSTP must provide a mechanism to avoid packet loss or the threat of reordering.

- Ordered Delivery of SAPUs. A Signaling Application Protocol Unit (SAPU) is the basic transmission unit for signaling. A SAPU is derived from the signaled state in the h-src node and it is used to set, modify, or delete the state in the h-sink node. CSTP must ignore out-of-order trigger messages.

- Soft State Support. Soft state support is a fundamental robustness mechanism of CSTP. This mechanism removes the states that are not periodically refreshed or explicitly torn down.

- Fragmentation, Reassembly, and Bundling of SAPUs. CSTP must be able to fragment and reassemble SAPUs that exceed one MTU size. Bundling permits a single IP packet to carry multiple small SAPUs.

- Full Internet-Layer Support. CSTP should support the full range of Internet-layer protocols, including IPv4 and IPv6, unicast and multicast delivery, and IPSEC for data security.

- Partial Deployment. It must be possible for signaling protocols supported by CSTP to operate correctly through CSTP-incapable nodes.

- Congestion Control. In some degree, the techniques of TCP-friendly congestion control may be applicable to CSTP. CSTP might be a good candidate for the 'Congestion Manager' [4].

- Optional hop-by-hop integrity.

Comparing the CSTP requirements with the NSIS QoS signaling requirements, we know that CSTP basically complies with NSIS QoS signaling requirements.

2.3.2 CSTP Messages

The following is a revised CSTP message definition based on the ID. We have corrected the errors in the CSTP message definition from the original ID. See the reasons in Chapter 6.

```
<CSTP Message>      ::=     <B-header> < Bundling datagram>

<B-header>          ::=     <Total-length of datagram>
                            <fragment offset> <MF bit>
                            <Integrity Check>
                            <ALSP-id> <R>

< Bundling datagram>::=     <M-header-MF-pair> | <M-header-pair-list>

<Integrity Check>   ::=     <Checksum> | <Keyed hash integrity object>

< Keyed hash integrity object >::= <challenge-object>

<M-header-MF-pair>  ::=     <M-header-MF> <SAPU>

<M-header-MF>       ::=     <M-header>

<M-header-pair-list>::=     <M-header> [<SAPU>] | <M-header-pair-list>

<M-header>          ::=     <h-src> [<h-sink> | <h-dest>]
                            <length of the pair of <M-header>,[SPAU]>
```

```
                          <CSTP message type>
                          <SAPUid-list>

<CSTP message type> ::=   <NEW> | <MOD> | <TEAR> | <REFRESH>
                          | <ACK> | <NACK> | <EVENT> | <CHALLENGE>
                          | <RESPONSE>

<SAPUid-list>       ::=   <empty> | <SAPUid> <SAPUid-list> | <SAPUid>

<SAPU>              ::=   <length of SAPU> <SAPU datagram>
```

Here,

- $< Total - length\ of\ datagram >$ counts in bytes.
- MF in 'MF bit' means 'More Fragments'.
- R means refresh time for all messages bundled in the datagram.
- SPAU, a Signaling Application Protocol Unit (SAPU), is the basic transmission unit for signaling. A SAPU is derived from the signaled state in an h-src node and it is used to set, modify, or delete the state in an h-sink mode.

To help readers understand the B-header definition, we present the B-header diagram in Figure 2. 'Other details' includes information such as version numbers, IP address format specifications, and flags.

<table>
<tr><td>Datagram length</td><td colspan="2">Fragment offset</td><td colspan="2">MF bit</td></tr>
<tr><td colspan="2">Checksum or keyed hash integrity object</td><td colspan="2">ALSP id</td><td>R</td></tr>
<tr><td colspan="5">Other details</td></tr>
</table>

Figure 2: B-header Structure

When there is a burst of signaling requests, CSTP permits the bundling of many individual CSTP messages into one CSTP bundled message. Moreover, if an individual CSTP message is larger than the MTU size, CSTP can fragment the SAPU into pieces and reconstruct fragmented CSTP messages for the fragmented SAPU. Therefore, a bundling header 'B-header' is mandatory.

The CHALLENGE and RESPONSE messages are used to initialize the keyed hash integrity check. The $< challenge - object >$ is carried as a CSTP-level SAPU, which is a special case; all other SAPUs are opaque to CSTP and carried on behalf of an ALSP. $< challenge - object >$ has the format as seen in Figure 3. For all details around CHALLENGE message, RESPONSE message and $< challenge - object >$ see [3].

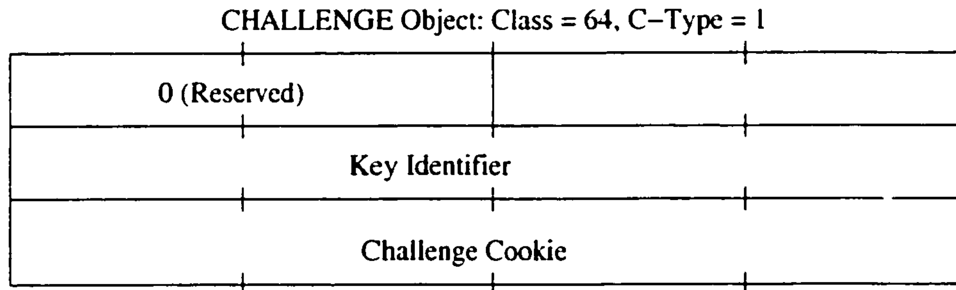


Figure 3: Challenge Object

In total, CSTP has nine types of messages. We categorize them into:

- Trigger message: NEW, MOD, TEAR
- Event message: EVENT
- Refresh message: REFRESH
- Acknowledge message: ACK, NACK
- Integrity message: CHALLENGE, RESPONSE

This division will help us to handle CSTP packet delivery and soft state management in our model.

2.3.3 CSTP Mechanism

The original ID only gives an H-Src State Transit Diagram and some framework-style mechanism for CSTP. Thus, in this section, we expand on the details of the CSTP mechanism.

Note that CSTP messages delivery behaviors are hop-by-hop behaviors, which means that the behaviors always take place between two CSTP neighbours h-src and h-sink. Figure 4 shows the hop-by-hop behaviors. Compared to Hop1, Hop2 is an h-sink node; compared to Hop3, Hop2 is an h-src node. Delivery behaviors between CSTP neighbours could be one-to-one for unicast or one-to-many for multicast.

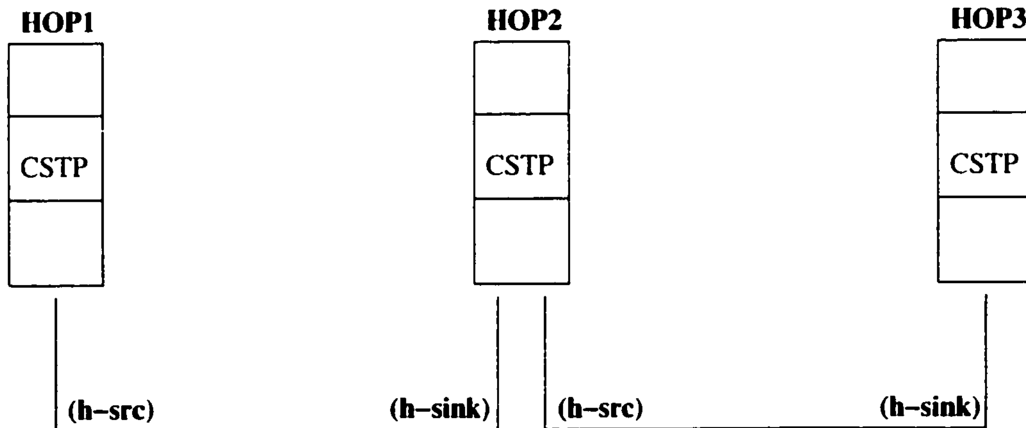


Figure 4: CSTP Pair Behaviors

The basic services of CSTP are reliable delivery and soft state management. Around these two points, CSTP mechanism is designed. In CSTP, there are two kinds of messages: trigger messages and event messages. CSTP's task is to deliver those two kinds of messages according to their characteristics.

Trigger messages need not only to be delivered reliably, but also to be refreshed as soft states. Figure 5 is a revised H-Src CSTP State Transit Diagram, which

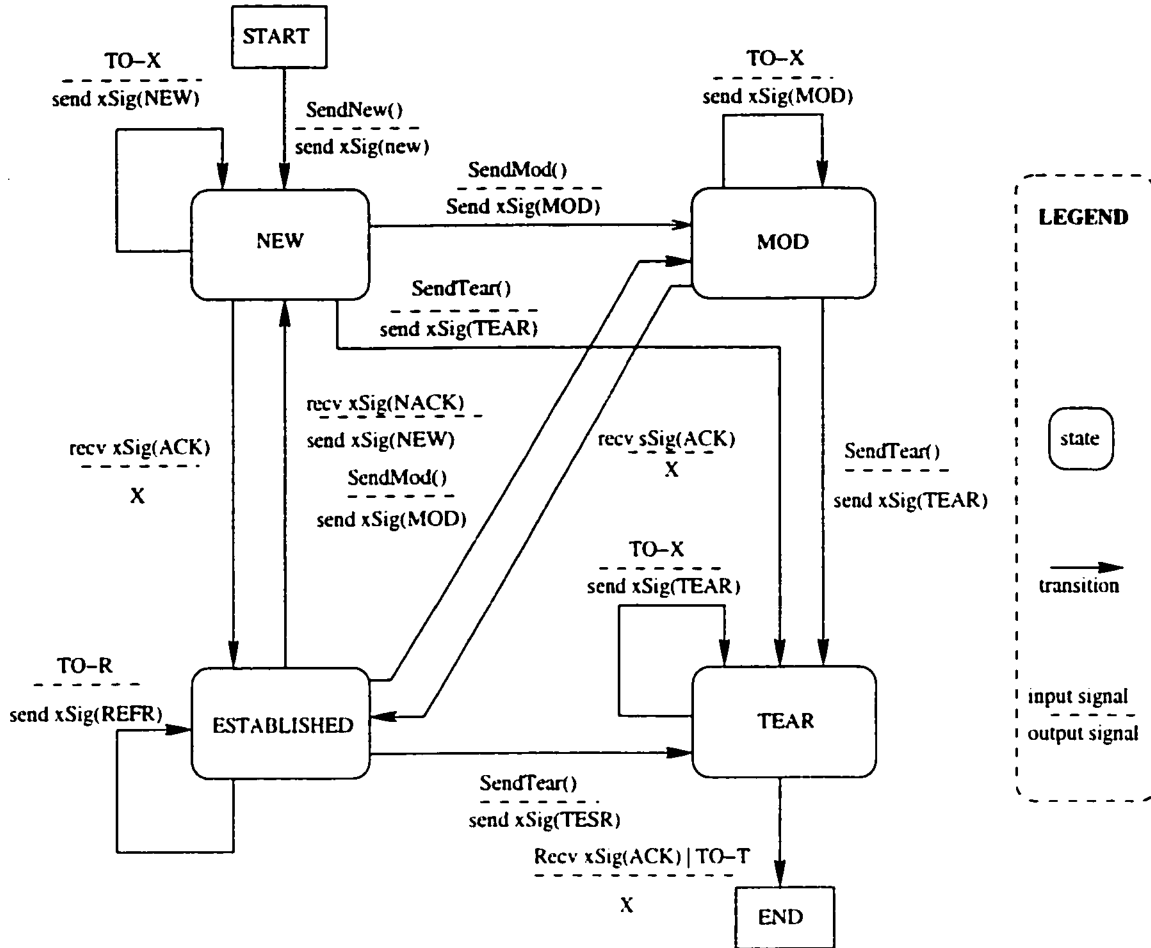


Figure 5: H-Src CSTP Trigger Messages State Transit Diagram

describes the trigger message transmission mechanism of CSTP at an h-src node. Figure 6 is an H-Sink CSTP State Transit Diagram, which describes the trigger message transmission mechanism of CSTP at an h-sink node and also is one of our contributions. From the cooperation of the two diagrams, we can learn how the CSTP mechanism for delivery of trigger messages works.

At an h-src node, the life cycle of the H-Src CSTP Trigger Messages State Transit Diagram can be divided into three phases: Creating, Living and Ending.

1. Creating. As soon as a CSTP receives an interface downcall SendNewSAPU from an local ALSP, a Trigger Sending Soft State Block (TSSSB) is created.

2. Living. As soon as the TSSSB is created, the first task of the CSTP is to

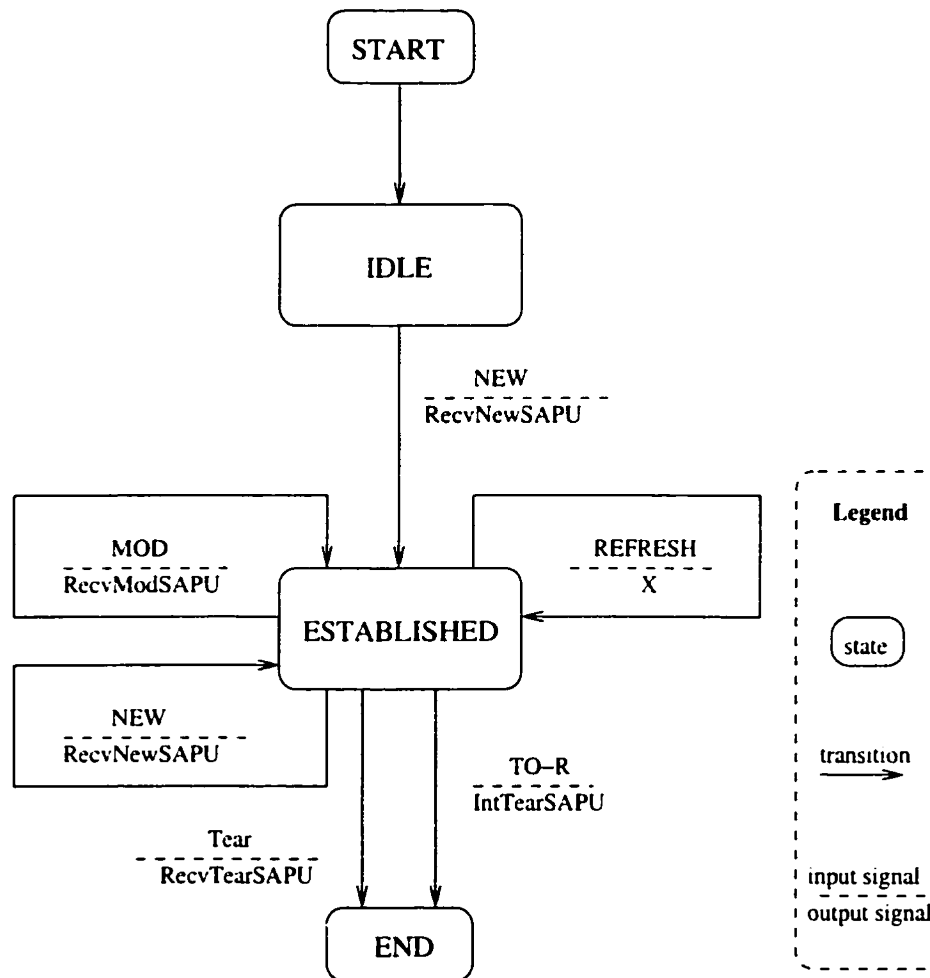


Figure 6: H-Sink CSTP Trigger Messages State Transit Diagram

compose a CSTP message xSig(NEW) and send it to the h-sink node. Then, the TSSSB will set its resending timer and resending counter. After that, the TSSSB will transit its state from START into NEW.

At the state NEW, the TSSSB could have four different actions corresponding to the four kinds of triggers:

- Once the resending timer expires, it will trigger the TSSSB to resend the same message. At the same time, the resending counter will increase by one.
- If the resending counter reaches its limit, the TSSSB will send an up interface call SendFail to the ALSP.
- If the TSSSB receives a local down call SendModSAPU, it will compose a CSTP message xSig(MOD) and send it to the h-sink node. Then it will set its resending timer and resending counter. After that, it will transit itself to the state MOD.
- If the TSSSB receives a CSTP message xSig(ACK), it will transit itself to the state ESTABLISHED.
- If the TSSSB receives a local interface down call SendTearSAPU, it will compose a CSTP message xSig(Tear) and send it to the h-sink node. Then it will set its tear timer, resending counter, and state life timer. After that, it will transit itself into the state TEAR and enter into the Ending phase.

At the state MOD, the TSSSB could have four different actions corresponding to the four kinds of triggers:

- Once the resending timer expires, it will trigger the TSSSB to resend the same message. At the same time, the resending counter will increase by one.
- If the resending counter reaches its limit, the TSSSB will send an up interface call SendFail to the ALSP.
- If the TSSSB receives a local down call SendModSAPU, it will compose a CSTP message xSig(MOD) and send it to the h-sink node. Then it will set its resending timer and resending counter. After that, it will stay at the state MOD.

- If the TSSSB receives a CSTP message xSig(ACK), it will transit itself to the state ESTABLISHED.
- If the TSSSB receives a local interface down call SendTearSAPU, it will compose a CSTP message xSig(Tear) and send it to the h-sink node. Then it will set its tear timer, resending counter, and state life timer. After that, it will transit itself into the state TEAR and enter into the Ending phase.

From the above description, we know that the actions from the state MOD are extremely like the ones from the state NEW. At the state ESTABLISHED, the TSSSB could have four different actions corresponding to the four kinds of triggers:

- If the refreshing timer goes off, the TSSSB will send an xSig(REFRESH) to the h-sink node. Then it will restart its refreshing timer again.
- If the TSSSB receives a local down call SendModSAPU, it will compose a CSTP message xSig(MOD) and send it to the h-sink node. Then it will set its resending timer and resending counter. After that, it will transit its state into MOD.
- If the TSSSB receives a CSTP message xSig(NACK), it must regenerate a CSTP message xSig(NEW) and send it to the h-sink node. Then it will set its resending timer and resending counter. After that, it will transit itself to the state NEW.
- If the TSSSB receives a local interface down call SendTearSAPU, it will compose a CSTP message xSig(Tear) and send it to the h-sink node. Then it will set its tear timer, resending counter, and state life timer. After that, it will transit itself into the state TEAR and enter into the Ending phase.

3. Ending. While the TSSSB receives a local interface downcall SendTearSAPU at any of the state NEW, MOD or ESTABLISHED, it will transit its state into TEAR and enter into the Ending phase. At the state TEAR, the TSSSB could have three different actions corresponding to the three kinds of triggers:

   - If the resending timer expires, the TSSSB will resend the CSTP message xSig(TEAR) to the h-sink node. Then it will set its tear timer and make

its resending counter increase by one. After that, it will stay at the state TEAR.

- If the resending counter reaches its limit, the TSSSB will send an up interface call SendFail to the ALSP.
- If the state life timer expires, the TSSSB will kill itself immediately.
- If the TSSSB receives a CSTP message xSig(ACK), it will kill iteself immediately.

As shown in Figure 6, the life cycle of the H-Sink CSTP Trigger Messages State Transit Diagram is much simpler than the one at an h-src node, though it can also be divided into three phases: Creating, Living and Ending:

1. Creating. As soon as a CSTP at an h-sink node receives a CSTP message xSig(NEW) incoming from an h-src node, a Trigger Receiving Soft State Block (TRSSB) is created. Then the TRSSB will pass the incoming SAPU to the local ALSP. After that, the TRSSB will set its refereshing timer and transit itself into the state ESTABLISHED. By now, the TRSSB has entered into the second phase: Living.
2. Living. At the state ESTABLISHED, once the TRSSB receives a CSTP message either xSig(New) or xSig(MOD), it will restart its refreshing timer and pass the incoming SAPU to the local ALSP. However, if it receives nothing until its refreshing timer goes off, or if it receives a CSTP message xSig(TEAR), it will enter into the phase Ending.
3. Ending. At this phase, the TRSSB can have two different actions corresponding to the two kinds of triggers:
    - Once the refreshing timer expires, the TRSSB will send a local interface up call IntTearSAPU to the ALSP. Then it will kill itself.
    - Once the TRSSB receives a CSTP message xSig(TEAR), it will send a local interface up call RecvTearSAPU to the ALSP. Then it will kill itself.

According to the ID, Event messages need to be sent reliably but not to be refreshed as soft states. Therefore, the mechanism of Event message transmission is simlar to the mechanism we described above, except that it does not need refreshment. To avoid repetition, we omit the introduction to the CSTP Event message transmission mechanism. For more information, please also see the Event Sending/Receiving Soft State Transit Diagrams in Chapter 4.

2.4 Application-Layer Signaling Protocols (ALSPs)

As we mentioned, an ALSP is an implementation of algorithms and data structures for a particular application. Under the architecture, ALSPs will reuse current RSVP version 1 and its extensions features as blueprints to implement various signaling services. The future ISPS will have a gateway to bridge the current RSVP version 1.

According to the ID, almost all the RFC2205 (RSVP version 1) can be taken as an ALSP. RFC2205 specifies the basic RSVP functionality. RSVP can be used with multicast and unicast traffic to reserve bandwidth on each node for a particular flow along a given data path. RSVP is not a routing protocol, but it does use routing protocols and consults the local router tables for routes. A typical reservation flow is initiated by sending a PATH message downstream to the receiver. Each node in the data path establishes a PATH soft state, to maintain the appropriate QoS. A PATH message states the flow ID, reservation information, and the source and destination address. Once the PATH message reaches its destination, the receiver will compose an RESV message, and send it to the sender along the path that the PATH message was coming. Along the path, each router will examine the request carried in the incoming RESV message. If a router has enough resource for the request, it will establish an RESV soft state for the RESV message, and then forward the RESV message to the sender. If a router does not have enough resource for the request, it will send an ResvErr message to the receiver. Once the sender receives the RESV message, the reservation is then set up. Once the receiver receives the ResvErr message, the reservation is then failed. To maintain PATH soft states, the sender has to send PATH messages to the receivers periodically. Similarly, to maintain RESV soft states, the

receiver has to send RESV messages to the sender periodically. Once the sender and receiver are done with the reserved flow, a PathTear message is sent to tear down the flow. Resources are then released to be used in a later reservation.

Standard RSVP (RFC2205) maintains states by sending PATH and RESV messages periodically. These messages are used to both synchronize state between RSVP neighbors and to recover from lost RSVP messages. The use of these messages to cover many possible failures has resulted in a number of operational problems. One problem relates to scaling, another relates to the reliability and latency of RSVP Signaling. RFC2961 is the addition to RSVP, to address to resolve these two problems. It forms the basis for CSTP.

The combination of an ALSP based on RFC2205 and the CSTP based on RFC2961 will give us a complete signaling stack, with equivalent functionality to RSVP version 1, although the packets formats will be different.

2.5 The Interface between CSTP and ALSPs

Under the architecture, CSTP is on the bottom level, while ALSPs are on the top level. Both levels are independent, so a generic interface is mandatory between these two levels.

There are two kinds interface calls: down calls and up calls. Down calls include:

- SendNewSAPU: send a new SAPU from an ALSP to CSTP
- SendModSAPU: send a modified SAPU from an ALSP to CSTP
- SendTearSAPU: send a tear down SAPU from an ALSP to CSTP
- SendEventSAPU: send an Event SAPU from an ALSP to CSTP
- SendInfoSAPU: send an informational SAPU from an ALSP to CSTP

Up calls include:

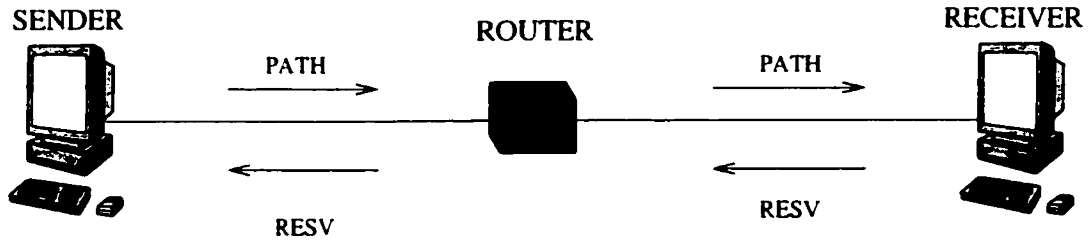


Figure 7: Resource Reservation in RSVP Version 1

- SendFail: send a 'fail in sending' message from CSTP to a specific ALSP
- RecvNewSAPU: pass a received new SAPU from CSTP to a specific ALSP
- RecvModSAPU: pass a received modified SAPU from CSTP to a specific ALSP
- RecvTearSAPU: pass a received tear SAPU from CSTP to a specific ALSP
- RegenSAPU: send a 'request to regenerate a SAPU' from CSTP to a specific ALSP

Figure 1 also depicts the interface calls between CSTP and ALSPs. The detailed scenarios of major interface calls will be given in Chapter 5. Some suggested changes of the interface calls will be given in Chapter 6.

2.6 An Example of an ISPS Offering the Simplest Unicast Resource Reservation Services of RSVP Version 1

RFC2205 regulates the RSVP version 1 resource reservation scenarios. Figure 7 depicts the simplest case of a unicast end-to-end signaling application with only one router. First, let us survey how RSVP version 1 offers resource reservation services in this model.

To set up a path and to reserve resources along the path in RSVP version 1, first, SENDER should create a soft state for the PATH message to be sent and then

send out the PATH message, which includes information such as destination address, session object. Before SENDER sends the packet, RSVP has to query the outgoing interface for this packet so that the lower layers do not need to decide the route of this packet.

When ROUTER receives this PATH message, it judges if it is the receiver of this message. If not, it sets up a path soft state for the PATH message, updates necessary information in the PATH message, such as PHOP (an object used in RSVP messages), and then forwards the message to the outgoing interface based on its internal route query.

Once RECEIVER receives this PATH message, it creates a path soft state for this PATH message. Then it creates a soft state block for a reservation message RESV to be sent. In this state block, it composes a resource reservation message RESV. As the PHOP object carried in the incoming PATH message includes the address of the next router or host, RECEIVER can send the RESV message back along the path the PATH message came.

When ROUTER gets the RESV message, the local RSVP daemon decides if ROUTER has enough resource for this request from the RESV message. There are two situations:

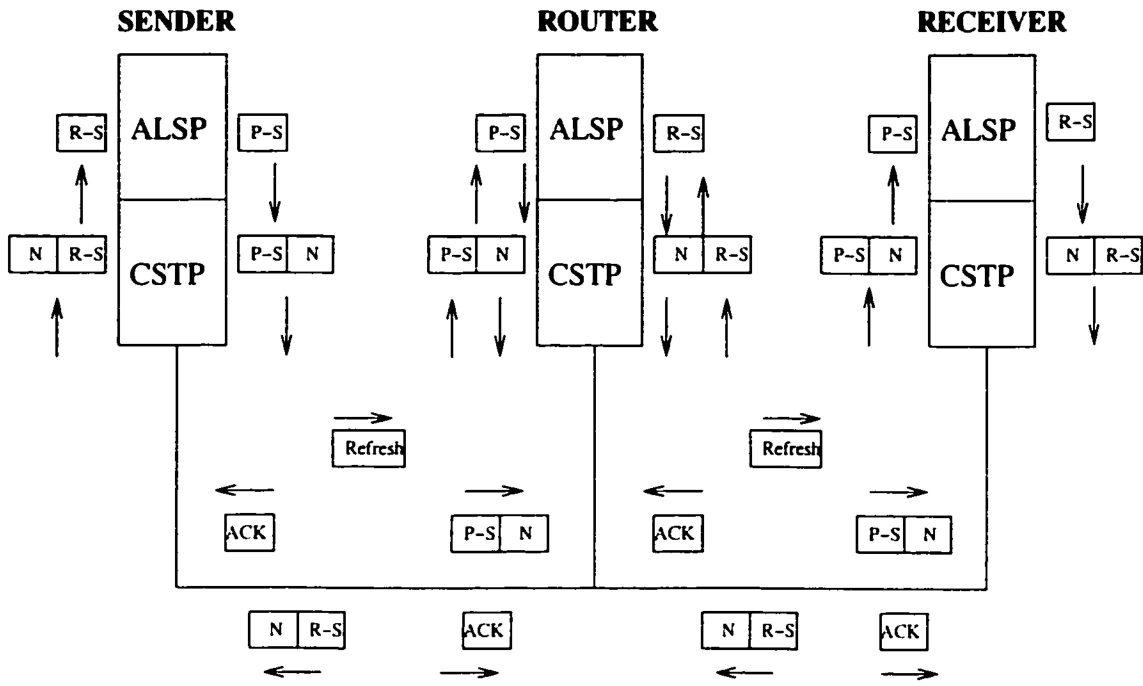
- If it has enough resource, it creates a reservation soft state block for this incoming RESV message. After retrieving the address of the next host from the corresponding path soft state block, ROUTER gets the outgoing interface for the RESV message by a query. Then, ROUTER can forward the RESV message to SENDER along the path that the PATH message came. Once SENDER receives the RESV message, it sets up a reservation soft state block for this incoming RESV message. Until now, RECEIVER has succeeded in its resource reservation.
- If it has not enough resource, it creates a ResvErr message. After retrieving the address of RECEIVER from the object PHOP carried in the incoming RESV message, ROUTER can get the outgoing interface of this ResvErr message by internal query. Then, ROUTER sends the ResvErr message to the RECEIVER.

By getting this message, RECEIVER knows the failure of its resource reservation.

Once SENDER sends out the first PATH message to RECEIVER and it does not receive any error message, it has to periodically issue a PATH message (taken in RFC2205) or a SREFRESH message (taken in RFC2961) to RECEIVER along the signaled path to maintain this path. At each node, the PATH or SREFRESH message refreshes the path soft state block and keeps it alive. Likewise, once RECEIVER sends out the first RESV message and it does not receive any error message, it has to periodically issue a RESV message (taken in RFC2205) or a SREFRESH message (taken in RFC2961) to SENDER along the signaled path to refresh all reservation soft state blocks to keep them alive. Once a reservation soft state block along the path receives a ResvTear message sent by RECEIVER, or it does not receive a PATH or SREFRESH message before its refreshing timer expires, it will kill itself immediately. Resulting from this, RECEIVER will not be able to receive packets from SENDER. Once a path soft state block along the path receives a PathTear message sent by SENDER, or it does not receive a PATH or SREFRESH message before its refreshing timer expires, it will kill itself immediately. Resulting from this, this sending path ends, and RECEIVER cannot receive information from SENDER anymore.

Now, let us examine how this two-level architecture can offer the above reservation services. Corresponding to Figure 7, we set up a three-node model, as shown in Figure 8. Each node has two levels: ALSP and CSTP. ALSP and CSTP are independent from each other. They communicate by interface calls. In this model, we only take one ALSP module to fulfill the resource reservation task at the ALSP level, though a group of ALSP modules will be coexisting to fulfill kinds of tasks at the ALSP level in the real world. From another point of view, CSTP is in charge of reliable delivery of SAPUs and soft state management. The SAPU at ALSP could be of different types, such as path SAPU or reservation SAPU. Here basically, a path SAPU and a RESV SAPU are respectively equal to a PATH message and a RESV message in RSVP version 1, except some small changes. All SAPUs are opaque to the CSTP.

Once SENDER wants to set up a signaled path to RECEIVER, first of all, the ALSP of SENDER has to create a path soft state block for the PATH SAPU to



Legend:

PATH SAPU: P-S

RESV SAPU: R-S

CSTP ACK message: ACK

CSTP REFRESH message: Refresh

CSTP NEW messages: P-S N   N R-S

Figure 8: Resource Reservation in The Two-level Architecture

be sent. In this state block, it composes a PATH SAPU, and then queries the outgoing interface for the PATH SAPU to RECEIVER. Finally, it sends a down call SendNewSAPU to pass the PATH SAPU and that outgoing interface to the local CSTP. At CSTP level, CSTP creates a sending soft state block for this incoming SAPU, then encapsulates the SAPU into a CSTP message and sends this newly created CSTP message to ROUTER. At ROUTER, when CSTP receives this incoming CSTP message, it creates a receiving soft state block for this incoming CSTP message and sends back a CSTP message ACK to the SENDER; then, it retrieves the PATH SAPU from the incoming CSTP message and issuses an upcall RecvNewSAPU to pass the incoming PATH SAPU to the local ALSP module. Upon the receipt of the PATH SAPU, the ALSP module of ROUTER creates a path soft state block for this incoming PATH SAPU, modifies necessary information of the incoming PATH SAPU, queries the outgoing interface for the modified PATH SAPU, and then issues a down interface call SendNewSAPU to pass the modified PATH SAPU and outgoing interface to the local CSTP. Once the CSTP receives this call, it creates a sending soft state block for this incoming SAPU, then it encapsulates the SAPU into a CSTP message and sends this new created CSTP message to RECEIVER. At RECEIVER, when CSTP receives this incoming CSTP message, it creates a receiving soft state block for this incoming CSTP message and sends back a CSTP message ACK to the ROUTER; then, it retrieves the PATH SAPU from the incoming CSTP message and issues an upcall RecvNewSAPU to pass the incoming PATH SAPU to the local ALSP module. Upon the receipt of the PATH SAPU, the ALSP module of RECEIVER creates a path soft state block for this incoming PATH SAPU. By now, a signaled path has been set up.

The next is how RECEIVER tries to make resource reservation along the signaled path. First of all, the ALSP module of RECEIVER has to create a reservation soft state block for the RESV SAPU to be sent. In this state block, it composes a RESV SAPU. From the corresponding local path soft state block, it gets the next hop ROUTER's IP address. Then, it queries the outgoing interface for this IP address. Finally, it sends a down call SendNewSAPU to pass the RESV SAPU as well as that outgoing interface to the local CSTP. At local CSTP, CSTP creates a sending soft state block for this incoming SAPU, then encapsulates the SAPU into a CSTP message and sends this newly created CSTP message to ROUTER. At ROUTER,

when CSTP receives this incoming CSTP message, it creates a receiving soft state block for this incoming CSTP message and sends back a CSTP message ACK to SENDER; then, it retrieves the RESV SAPU from the incoming CSTP message and issues an upcall RecvNewSAPU to pass the incoming RESV SAPU to the local ALSP module. Upon the receipt of the RESV SAPU, the ALSP module of ROUTER creates a reservation soft state block for this incoming RESV SAPU, modifies necessary information of the incoming RESV SAPU, gets the next hop SENDER's IP address from the corresponding local path soft state block, queries the outgoing interface for the RESV SAPU, and then issues a down interface call SendNewSAPU to pass the modified RESV SAPU and that outgoing interface to the local CSTP. Once the local CSTP receives this call, it creates a sending soft state block for this incoming SAPU, then encapsulates the SAPU into a CSTP message and sends this newly created CSTP message to RECEIVER. At RECEIVER, when CSTP receives this incoming CSTP message, it creates a receiving soft state block for this incoming CSTP message and sends back a CSTP message ACK to the ROUTER; then, it retrieves the RESV SAPU from the incoming CSTP message and issuses an upcall RecvNewSAPU to pass the incoming RESV SAPU to the local ALSP module. Upon the receipt of the RESV SAPU, the ALSP module of RECEIVER creates a reservation soft state block for this incoming RESV SAPU. By now, RECEIVER has succeeded in reserving resources. From the view of CSTP level in the whole model, RECEIVER has also set up a signaled reservation path to SENDER. In all, there are two virtual paths at CSTP level: SENDER-ROUTER-RECEIVER for delivery of ALSP PATH SAPU, and RECEIVER-ROUTER-SENDER for delivery of ALSP RECV SAPU.

To maintain these virtual paths, CSTP refreshment messages are necessary after a sending soft state block succeeds in delivery SAPU. Since CSTP behaviours happen between CSTP neighbours, to maintain the path of SENDER-ROUTER-RECEIVER, CSTP at SENDER has to periodically send CSTP refreshment messages to refresh receiving soft state block at ROUTER, while CSTP at ROUTER has to periodically send CSTP refreshment messages to refresh receiving soft state block at RECEIVER. Once a receiving soft state block receives a CSTP TEAR message, or its own state timer goes off, it will kill itself immediately. Resulting from this, this signaled path will be broken and RECEIVER cannot receive information from SENDER anymore.

Likewise, to maintain the path of RECEIVER-ROUTER-SENDER, CSTP at RECEIVER has to periodically send CSTP refreshment messages to refresh receiving soft state block at ROUTER, while CSTP at ROUTER has to periodically send CSTP refreshment messages to refresh receiving soft state block at SENDER. Once a receiving soft state block receives a CSTP TEAR message, or its own state timer goes off, it will kill itself immediately. Resulting from this, this signaled path will be broken and RECEIVER cannot receive information from SENDER anymore.

Chapter 3

Introduction to SDL and MSC

3.1 SDL

Specification and Description Language (SDL), which is the ITU-T Recommendation Z.100 [21], was developed by CCITT (now ITU-T). It is an object-oriented, formal language to specify and describe systems, and is mainly used in complex, real-time applications. In this chapter, we will give a basic introduction to SDL.

3.1.1 History

The basic SDL was available in 1976. From then on, it was improved every four years. By 1988, SDL-88 reached a stable form described in the single Recommendation Z.100 in the CCITT Blue Book. SDL-92 was extended with the mechanism supporting object-oriented, parameterized types and packages. In 1996, a few updates were made to the language in an addendum to SDL-92. The addendum relaxed a number of rules for the language to make it easier to use in an even more flexible way. The latest version, SDL-2000, gives better support for object modeling and for code generation. Today SDL is a complete language in all senses. It is widely used for development work in the telecommunication industry and in addition, it is used in standards on

signaling and network functions [15].

One of ITU Study Groups, Languages for Telecommunication Applications, covers the maintenance of SDL, Message Sequence Charts(MSC), and joint work with ISO on utilizing specifications in conformance testing.

3.1.2 Characteristics

From the view of system engineering, the dominant characteristics of SDL are:

- Formalism. It is a formal language on the basis of the finite state machine, thus it ensures precision, consistency, and clarity in design. It can detect errors at the early phase of software development by simulation and verification. It also can assure that the specification stays in accordance with the user requirements by validation.
- Support of most phases of software engineering. It can be applied on the phases from system architecture design to implementation under tool support. SDL specification can be automatically translate into C code at the phase of implementation. Moreover, it can create test cases for implementation conformance test.
- Complex application environment. It is suitable for real-time systems, especially for reactive, discrete systems. It is a model-oriented specification language, particularly suitable for specifying systems in which it is important to understand behavior aspects before they are implemented. With advanced tools, SDL descriptions even become the basis for rapid prototyping.
- Object-oriented support. It supports object-oriented characteristics.
- Scalability and portability. It supports large-scale complex systems, and it can run on various platforms, such as Unix and Windows.
- Readability. It supports both graphic and text presentations. With a graphic presentation, it is easy to read.

- High efficiency. As a result of its formalism, it has a high degree of testability. The quality and speed improvements are dramatic compared to traditional informal design techniques.
- Security of investment. It is well supported by both commercial tools and a standardization body helping to secure engineering investment in SDL.

3.1.3 SDL Model Components

An SDL system consists of structure, communication, behavior and data. In an executable environment, an SDL system model runs on a parallel set of extended finite state machines (EFSMs). These machines are independent of each other and communicate with discrete signals.

3.1.3.1 Structure

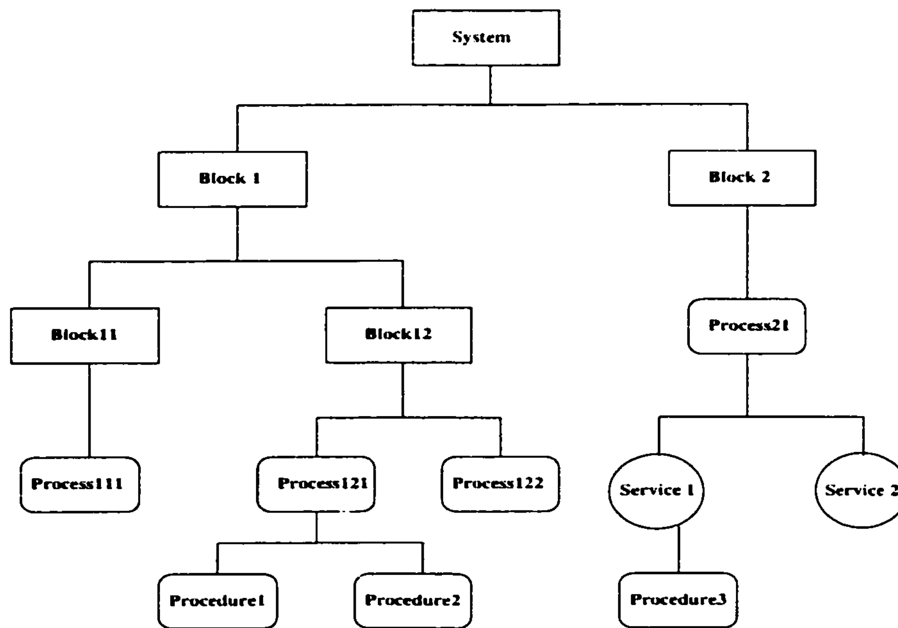


Figure 9: An SDL System Structure

A system structure consists of a set of blocks, and a block can embed in another

block. Each block consists of one or more processes, while a process consists of services or procedures. Note that services can not stay at the same level with procedures in a process; however, a service can include procedures inside, while a procedure can not have a service inside. Figure 9 shows a SDL system structure. Figure 10 shows the basic high level SDL legend.

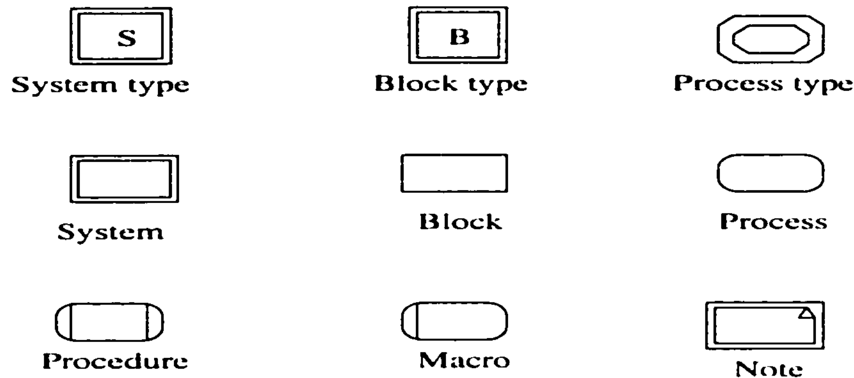


Figure 10: Basic SDL Legend

A process describes a unit of dynamic system behavior in the form of an extended state machine. A service defines a partial behavior of a process, representing a different way to specify a process while expressing the same behavior. Procedures are only defined once but can be called more than once in the same process.

A procedure is a parameterized part of a behavior with its own local scope. It is always local to the process in which it is specified. In case the procedure specification is outside a process, i.e., in a package, a copy of the procedure specification will be placed in the processes that call the procedure. A procedure can have two kinds of parameters, in and in/out.

A remote procedure is a special procedure that can be called in another process where it is defined. Sometimes, a remote procedure call is a more elegant model than the use of two explicit signals. A remote procedure call implies that the call of a procedure is defined within another context.

Figure 11 depicts a remote procedure call. The declaration of the remote procedure

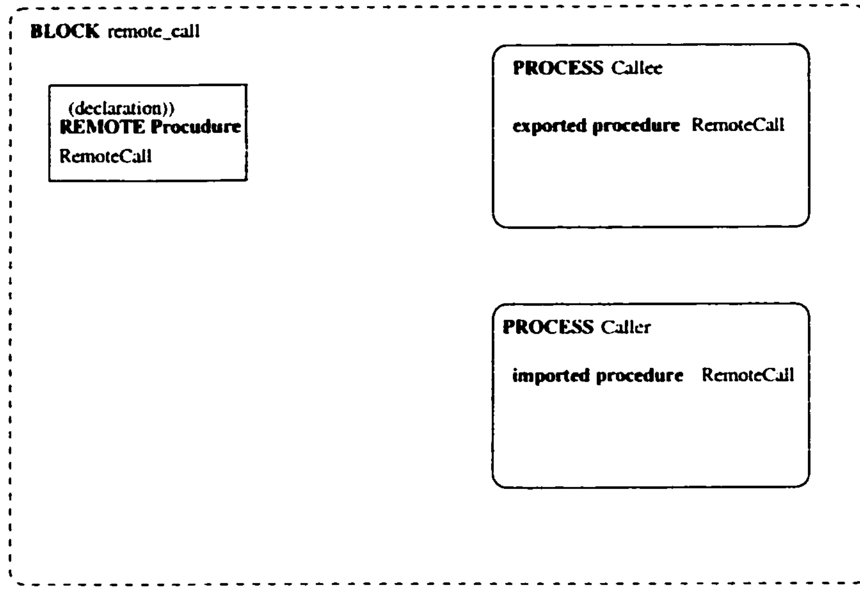


Figure 11: A Remote Procedure Call

RemoteCall should be visible for both the two processes. Caller and Callee.

To implement a remote procedure call. we have to declare:

- Exported procedure
- Imported procedure
- Remote procedure

The declaration of an exported procedure is included in an independent process, while the declaration of an imported procedure is included in the process that issues a remote procedure call. The declaration of a remote procedure should be placed at a certain level that is visible for both the declarations of the exported procedure and the imported procedure. An exported procedure and several imported procedures may refer to the same remote procedure. The calling process waits in an implicit state until the called procedure returns a signal. At this time, the remote calling process has been completed.

3.1.3.2 Communication

Signals communicate among SDL structure components, or between SDL structure components and the environment, by channels or routes. A channel connects two blocks, as well as a block and the environment. A route connects:

- Two processes
- Two services
- A process and the environment
- A service and the environment

In SDL, there are two kinds of signals: discrete signals and continuous signals. Both signals have priority; however, a priority discrete signal can only appear in a service. A signal out of an expired timer in a process instance is treated in the same way as a discrete signal. To put it simply, we refer to discrete signals as signals.

A signal may or may not carry parameters. There may be one or more parameters. A signal instance is created when a process executes an output, and it ceases to exist when the receiving process consumes the signal in an input. Communication paths (channels and routes) convey the signal instances from the sender to the receiver. SDL assumes that signal instances can also be created and consumed in the environment.

A signal transmits asynchronously between sending object and receiving object. However, remote procedure call is a special case. It can pass parameters synchronously in an implicit way. In order to reduce states, we usually take remote procedure calls instead of signals to pass parameters.

When a signal instance is outputted it is directed to a process instance set. It will be put into an unbounded FIFO-queue in each destination process instance when it arrives. Signals to be sent simultaneously along the same channel or route are conveyed in a random order.

A signal will be consumed if it is declared in a current active state in a process. It can also be saved in the queue in the order in which it arrives for other transaction uses later. If the signal is not declared in the current active state, it will be lost. A Boolean condition can be added to a signal to decide which signal should be consumed if the condition holds, or be implicitly saved if the condition does not hold.

Continuous signals are local variables of a process instance. Each continuous signal is declared in a transition at the process instance level. It is kept alive during a process's life, but it can only be triggered when the queue of the process instance becomes empty. If there is more than one continuous signal with the TRUE condition under a state, the transaction corresponding to the continuous signal with the highest priority is fired.

When a signal is consumed, a transition is triggered and a sequence of actions in the transaction is executed. At the end of this transaction, another or the same state as the starting state is reached.

3.1.3.3 Behavior

As a FDT, SDL can describe and specify all behaviors of a real-time reactive system. The process diagrams in SDL describe patterns of behavior, whereas the parts in the actual behavior are process instances. The basis for behavior description is communicating extended finite state machines (EFSMs). EFSMs are represented by processes. Processes can be dynamically created and terminated during system run time. Multiple instances of a process can coexist. Each instance is identified by a unique Process Identifier (PId), which makes it possible to send signals to each instance of a process.

To manage instances, four kinds of special, local PId expressions can be accessed in each process instance:

- SELF: denoting the current process instance itself
- SENDER: denoting the instance from which the most recent signal was sent to the current process instance

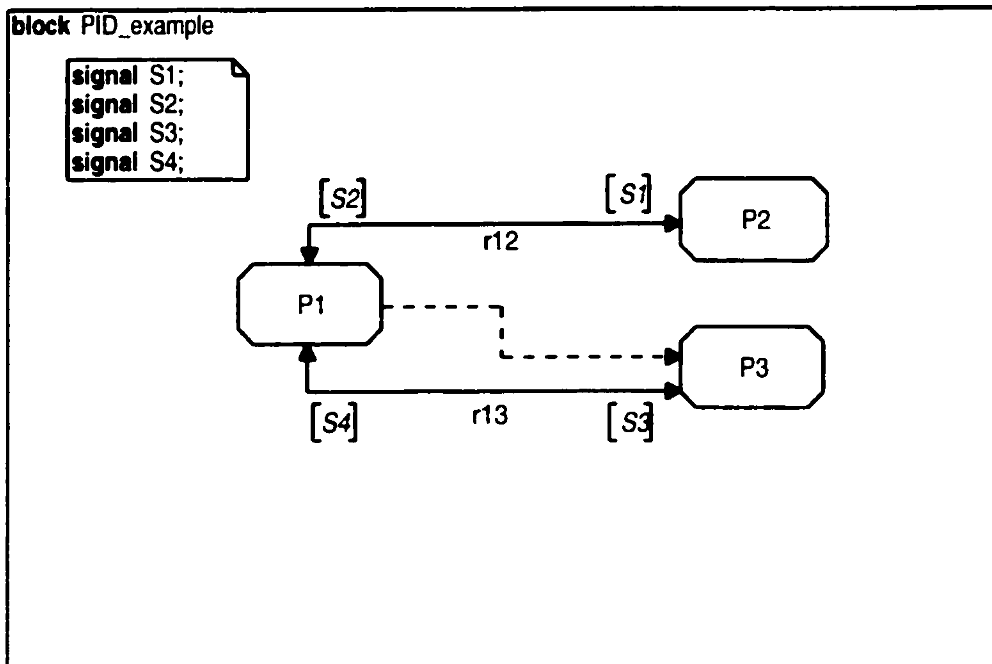


Figure 12: Process Identifier PId

- PARENT: denoting the instance which created the current process instance
- OFFSPRING: denoting the instance most recently created by the current process instance.

For example, in Figure 12, process instance P1 sends signal S1 to process instance P2. Thus, in P2, signal S1's SENDER process is P1. Process instance P1 creates process instance P3. Thus, P3's PARENT is P1, while P1's OFFSPRING is P3. In P1, the PId SELF denotes itself.

A process instance can be passed through parameters at the time of creation. Inside, each state connects to a transaction. At a certain state, an incoming stimulus triggers an execution of the transaction, starting at the current state and ending at the next state. After that, the process instance is waiting for the next stimulus to repeat the above operation. The process ceases when it executes a stop symbol.

To extend the state-space, variables are taken in a process instance, combining with states. All variables are local to process instances. Tasks are used to manipulate

local information, such as for the assignment of values to a process instance variable. Although the set and reset operations on timers and the export operations also use the task symbol, essentially they are not tasks.

In a process, a timer is a very important component, which is much like a signal: when set with an expiration time, a timer instance is created, and when expiring or reset, it ceases. When the timer goes off, a signal with the name of the timer is inserted in the input port of the process. Eventually, this signal can be handled like any other signal in an input with the name of the timer. When no longer needed, a timer can be reset before it expires or while it is in the input port, to avoid spurious expirations.

3.1.3.4 Data

Data in SDL is based on the concept of Abstract Data Types (ADTs). ASN.1 that has been standardized can also be used as a data type notation in combination with SDL so that SDL can share data with other languages, as well as reuse existing data structures. ADTs in SDL are called sorts.

Abstract Data Types

The ADT concept used within SDL is very well suited to the specification language. An abstract data type is a data type with no specified data structure. Instead, it specifies a set of values, a set of operations allowed, and a set of equations that the operations must fulfill. This approach makes it simple to map an SDL data type to data types used in other high-level languages.

SDL ADT can be divided into three main categories:

- Simple sorts,
- Structured sorts,

Predefined Sort Category	**Sorts Name**
Predefined simple sorts	Boolean
	Character
	Integer
	Natural
	Real
	Duration
	Time
	PId
Predefined structured sorts	Charstring
Predefined generator sorts	String
	Powerset
	Array

Table 1: Predefined Sorts in SDL

- Generator sorts.

The above each category can also be distinguished by Predefined Sorts and User Defined Sorts. Predefined Sorts are provided automatically with their set of operators. All Predefined Sorts are available in Table 1.

Each sort has a scope. The scope of a predefined sort is global to the system, while the scope of a user-defined sort is local. A data sort can be defined at any level in an SDL specification. Each sort has two parts: an interface part and a behavior part. The interface part defines how and which literals and operators can be used to

obey the language rules, and the behavior part defines the semantics of the literals and operators. The concept of ADT reflects the fact that most operations can be applied without knowing any details about how things are really done. This helps to gain better information hiding and appropriate abstraction level in the specification of target systems.

Variables in SDL are typed; in other words they are associated with a particular predefined or user-defined data type. An abstract data type defines in particular a set of values. A variable associated with a sort can only receive values defined for that sort.

The following is examples of user defined sorts.

```
SYNTYPE DATA_TTL_t = NATURAL
  CONSTANTS 1:100
ENDSYNTYPE;

NEWTYPE ALSP_Table_t
  ARRAY(SESSION_ID_t, PID)
ENDNEWTYPE;
```

ASN.1 data

ASN.1 data type definition mechanism is very similar to the ones usually known from programming languages, as well as the SDL data part. Here are some examples:

```
 M_header_t ::= SEQUENCE {
       hsrc         IPADDR_t,
       hdest        IPADDR_t  OPTIONAL,
       length_MS    NATURAL,
       CSTP_MT      CSTP_MT_t,
       SAPUid_list SAPUid_list_t
```

```
};

CSTP_MT_t ::= ENUMERATED {
      NEW, MODI, TEAR,  REFRESH,
      ACK, NACK, EVET, CHALLENGE,
      RESPONSE
};

SAPUid_list_t ::= SEQUENCE of SAPUid_t;
```

The first one above defines a structure named M_header_t, which has five fields, and the field 'hdest' is optional. The second one above defines a type named CSTP_MT_t, which includes all CSTP message type names. The third one above defines a list type named SAPUid_list_t for the type SAPUid_t.

3.1.4 Object-Oriented Characteristics

The OO concepts of SDL give users powerful tools for structuring and reusing. SDL covers the four basic aspects of object orientation (identity, classification, polymorphism and inheritance). A class in SDL is called a type, and an object in SDL is called an instance.

Type declarations can be placed anywhere, either at system level, or at each level inside the system. Type declarations can also be placed in packages outside the system, shared with other different specifications. To allow for generic type specifications, a type specification can be parameterized with so-called formal context parameters.

Corresponding to seven main kinds of instances: system, block, process, service, signal, and data, SDL classifies them into types: system type, block type, process type, service type, signal type and data type. A type can be used to define instances and to be specialized as a new type.

In SDL, specialization of types is easily accomplished in two ways:

- A subtype might have added properties not defined in the supertype. One can, for example, add new transitions to a process type, add new processes to a block type, etc.
- A subtype can redefine virtual types and virtual transitions that are defined in the supertype. It is possible to redefine the contents of a transition in a process type, to redefine the contents structure of a block type, etc.

3.2 MSC

MSC is a standard formal language, widely used for depicting selected system runs or traces or scenarios. It has been standardized by ITU in its Recommendation Z.120 [20]. Like SDL, it also has graphical and textual representations.

The main advantages of MSC are:

- MSCs provides an intuitive understanding of the system behaviors with their graphical layout.
- MSCs provides level abstraction for system requirements by merely describing the message flow at each system level.

MSCs are mainly used in

- requirement specifications
- capturing behaviors of systems
- validating behaviors of systems.

Recommendation Z.120 of MSC'96 suggests the use of high level Message Sequence Charts (hMSCs) that compose basic Message Sequence Charts (bMSCs) to

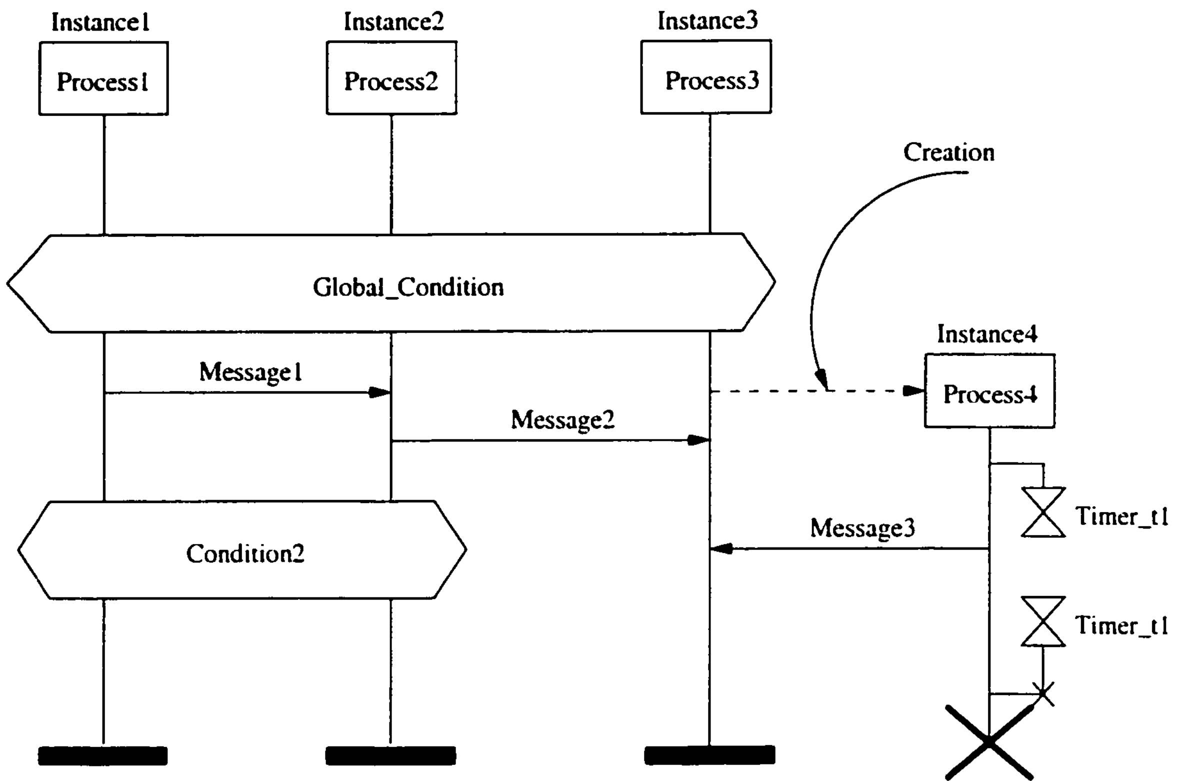


Figure 13: A bMSC example

specify systems using various compositional, recursive and nondeterministic operators. The basic components for a bMSC are instance, message, action, condition, instance creation, timer and instance stop. Figure 13 is an example of a bMSC. For more information, see the Recommendation Z.120.

Chapter 4

A CSTP Model

4.1 Model Requirements

The CSTP specification must be written based on the English text of the original 2LAIS Internet Draft. The overall requirements for CSTP illustrated in Chapter 2 are the ultimate target for CSTP. At the current research stage, we have to select the requirements to adapt to our target, though we try to model a completed CSTP. Before continuing, we have to give the assumptions of the model environment.

4.1.1 Assumptions of Model Environment

In short, the model would work in the environment where:

- all nodes are CSTP capable;
- the network could produce loss and delay;
- the routing table is stable and has no change.

4.1.2 Model Requirements

This model must satisfy two basic requirements: reliable delivery of signaling messages and soft state management. To implement reliable delivery, CSTP has to deal with fragmentation and reassembly of SAPU; in addition, CSTP has to deal with congestion control and ordered delivery of SAPUs. Generally, the ID gives two ways for CSTP to implement congestion control:

- CSTP dynamically computes the appropriate value for retransmission timers;
- CSTP performs complex scheduling of signaling message transmissions, taking into account the congestion at each target node and the signaling load.

At the current stage, we take the first way for the congestion control requirement. The second way remains to be developed.

Because CSTP is only a common signaling transport protocol, the combination of CSTP and ALSP drivers must support minimum core RSVP version 1 features at least.

The ID states that the bundling requirement may need further consideration. Thus, we will not consider the bundling except that we add a B-header on each CSTP message in this model.

4.2 System Model Architecture

4.2.1 Model Description

Figure 14 depicts the architecture of our model. In this model, there are three nodes: HSrc, HSink and HDest. HSrc is the sender, HSink is the router, and HDest is the receiver. CSTP messages are transmitted between each pair of CSTP neighbour nodes.

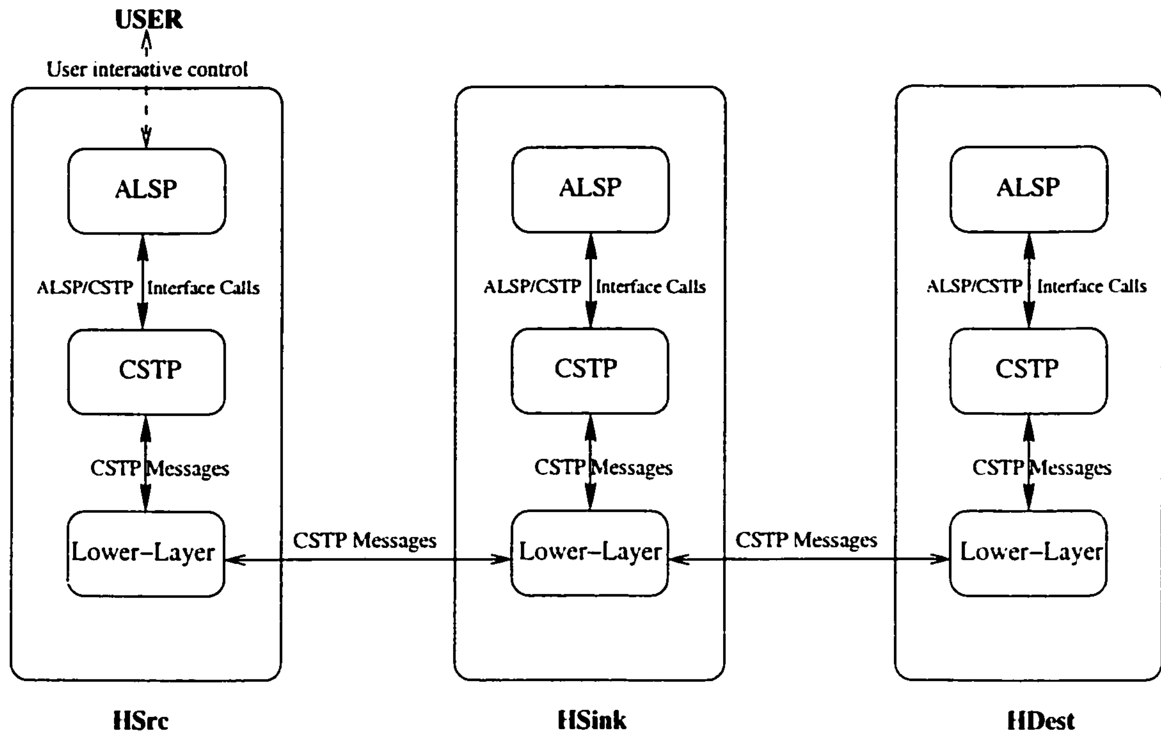


Figure 14: A CSTP Model

Each node has three layers: ALSP, CSTP and Lower_Layer. Layer ALSP and Lower_Layer are the test drivers, which also offer the working environment for CSTP. The layer CSTP is made up of the CSTP module, which implements the CSTP protocol services. The channels between each pair of CSTP neighbour nodes represent the network medium. In our simulation tool ObjectGEODE, they can also produce packet delay.

4.2.2 Satisfaction of the Model Requirements

This three-node model can satisfy our model requirements, and can be used to examine the CSTP behaviour. Obviously, this model satisfies the requirement of hop-by-hop behaviour, because the node pair HSrc and HSink, as well as HSink and HDest, are the two pairs of CSTP neighbours. By the appropriate design following the CSTP specification, this model can implement all the model requirements, such as reliable delivery, soft state management, fragmentation, reassembly, etc. By designing the test drivers in an ALSP module, this model can offer RSVP version 1 unicast features.

This model also can be extended. If we want to extend this model with multiple routers instead of one router later, we do not need to change our specification, since the CSTP specifications in each router are the same, and all the ALSPs in various routers are the same either.

In sum, this model can satisfy our current research aim. In the following section, we are going to discuss how this model is designed and how it fulfills the CSTP requirements in detail.

4.3 CSTP Module

From the functionality point of view, this module has three functional parts: Soft State Management, Soft State Blocks, and Transmission Processes, as shown in Figure 15. The Soft State Management includes the process StateManager; the Soft State Blocks include the processes TrgSendFSM, TrgRecvFSM, EventSendFSM and EventRecvFSM, while the Transmission Processes include Multiplexing and Demultiplexing.

Note that we will frequently use the words h-src, H-Src, HSrc, h-sink, H-Sink and HSink in the rest of the thesis. The words h-src and h-sink are adjectives to describe a node's status: when a node sends out a message, it is an h-src node; when a node receives a message, it is an h-sink node. Once these two words appear in a title of a procedure, they respectively become H-Src and H-Sink. HSrc and HSink are two specific names of the nodes in the model; see Figure 14.

4.3.1 Soft State Management

Soft state management is very important, as it is the one of the two basic services in CSTP. Basically, it has three major tasks: to create sending soft state blocks triggered by local ALSP downcalls SendNewSAPU or SendEventSAPU, to manage state blocks by maintaining data structures, to respond to queries. All the CSTP soft

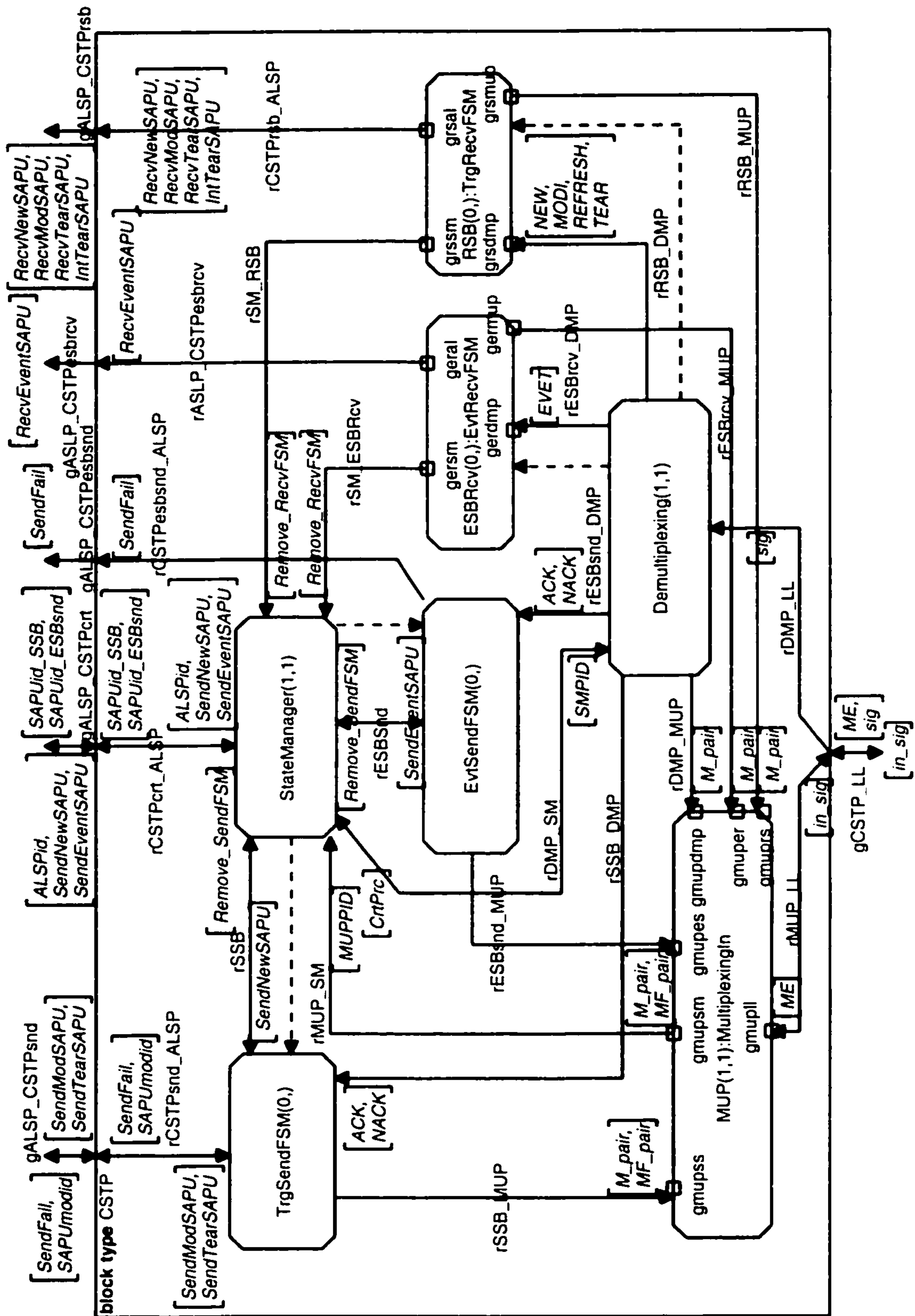


Figure 15: The CSTP Module

state management services are implemented in the process StateManager. To give a thorough understanding of it, we are going to introduce them from the following aspects.

4.3.1.1 Soft State Block Category

CSTP messages can be divided into two major categories: trigger messages and event messages. Message NEW, MOD and TEAR are trigger messages. To transmit a trigger message, we need a Trigger Sending Soft State Block at an h-src node and a Trigger Receiving Soft State Block at an h-sink node. Event messages have different characteristics from trigger messages, because they only need reliable delivery. Once an event message is delivered successfully, the Event Sending Soft State Block at an h-src node and the Event Receiving Soft State Block at an h-sink node must be killed. Thus, we can classify all CSTP soft state blocks into four categories:

- Trigger Sending Soft State Block (TSSSB)
- Trigger Receiving Soft State Block (TRSSB)
- Event Sending Soft State Block (ESSSB)
- Event Receiving Soft State Block (ERSSB)

4.3.1.2 Soft State Management Functions

The process StateManager will take charge of the following services:

- generating unique SAPUids for the interface down calls SendNewSAPU, SendModSAPU and SendEventSAPU;
- creating TSSSB or ESSSB triggered by a local ALSP down call SendNewSAPU or SendEventSAPU;
- managing state blocks by maintaining data structures;

- responding to kinds of queries;
- processing the ordered SAPU delivery;
- managing ALSP modules' processid at the CSTP level, so that a TRSSB or ERSSB can pass the incoming message to the right ALSP module.

4.3.1.3 Data Structure and Algorithms

To send an incoming message to the right active state block, and also to prevent an incoming message from being sent to a killed state block as an unexpected signal, we design the following data structures to manage the soft state blocks. With the data structure management, the StateManager can respond to kinds of queries, such as inquiries about soft state address. The following is a part of the data structures.

```
/*  For Trigger/Event/Refresh Message */
Source_t ::= SEQUENCE {
    HSRC                IPADDR_t,
    ALSPid              ALSPid_t,
    CSTPmsgT            CSTP_MT_t
};

NEWTYPE SourceSet_t
      POWERSET(Source_t)
ENDNEWTYPE;

SAPUid_PID_t ::= SEQUENCE {
     SAPUid             SAPUid_t,
     SBPid              PID
};

NEWTYPE StateTbl_t
    ARRAY(Source_t, SAPUid_PID_t)
ENDNEWTYPE;
```

```
/*  For ACK/NACK message */
NEWTYPE SAPUid_PIDtbl_t
    ARRAY(SAPUid_t, PID)
ENDNEWTYPE;

NEWTYPE SAPUidSet_t
    POWERSET(SAPUid_t)
ENDNEWTYPE;
```

The Source_t data is used to identify an incoming message's session. All active session data are collected in the powerset SourSet_t. If an incoming message's session is not in the powerset, it implies that the corresponding soft state block at this node has been killed. Otherwise, the soft state block corresponding to the incoming message's session is still active. The following algorithm is to respond to the soft state block address query. In addition, it manages ordered message delivery by both maintaining the newest SAPUid information, and discarding the older messages.

```
When the StateManager receives a query for a trigger message's
Destination Soft State Block PID (DSSBP),

If the source (HSRC, ALSPid, trigger/event/refresh) not in SourceSet
then return false;        /* the destination state block deactive */
else retrieve the DSSBP from the table StateTbl
     if the incoming SAPUid > the saved SAPUid
     then if the incoming message is NEW
          then updates the StateTbl table,
               return the DSSBP;
          else return false;      /* Discard the unordered message */
     else if the incoming SAPUid = the saved SAPUid
          then retrieves the DSSBP,
               if the incoming message is MOD
```

```
                then updates the StateTbl table,
                     return the DSSBP;
                else return false; /* the incoming SAPUid is older */
```

When a StateManager generates a new SAPUid for a soft state block, it must update its powerset SAPUidSet. It must ensure that all the SAPUids in that powerset are the newest ones and also that they are unique for corresponding soft state block. Thus, based on the following algorithm, an incoming ACK/NACK can find its DSSBP if the state block is still alive.

```
When the StateManager receives a query for an ACK/NACK message's
Destination Soft State Block PID (DSSBP),

If the incoming SAPUid in SAPUidSet
then retrieves the corresponding DSSBP,
     return the DSSBP and true;
else return false;  /* The ACK/NACK hasn't found DSSBP */
```

4.3.2 Soft State Blocks

Soft state blocks in CSTP modules are TrgSendFSM, TrgRecvFSM, EventSendFSM and EventRecvFSM. TrgSendFSM and TrgRecvFSM are a pair; TrgSendFSM is at an h-src node to send CSTP messages, while TrgRecvFSM is at an h-sink node to receive messages sent from the h-src node. Likewise, EventSendFSM and EventRecvFSM are a pair; EventSendFSM is at an h-src node to send Event messages, while EventRecvFSM is at an h-sink node to receive those Event messages sent from the h-src node. In Chapter 2, we have illustrated the trigger sending/receiving finite state machines. Here, we will illustrate them from the modeling point of view.

4.3.2.1 Trigger Sending Soft State Block: TrgSendFSM

The Trigger Sending Soft State Block, TrgSendFSM, is an implementation of Figure 5. It is used to reliably deliver the SAPU and maintain the sending soft state. Basically, it has three major services:

- composing SAPUs into CSTP messages; if a SAPU is greater than an MTU size, fragment it into pieces and compose the fragmented CSTP messages;
- delivering the CSTP messages reliably;
- refreshing trigger receiving soft state at an h-sink node by sending xSig(REFRESH)s periodically.

In some sense, the Trigger Sending Soft State Block can be treated as a group of predefined routines connected or triggered by both interface down calls and incoming CSTP messages. Triggered by an interface down call SendNewSAPU, the process StateManager creates an instance of TrgSendFSM, then it forwards the down call to the instance. At this time, the routine **Reliable Delivery Procedure at H-Src Node** is called. From then on, the following routines will be ready to be called until the instance is killed.

Reliable Delivery Procedure at H-Src Node When receiving a trigger down call, this procedure will be called. First, the state block sets the resending timer and the resending counter. Then the state block calls the routine **Sending A CSTP Message Procedure**. After that, the state block waits for an ACK. During the wait,

- if it receives an ACK at a non-Tear state, it will reset the resending timer, set its refreshing timer, and then call the routine **Refreshment Procedure at H-Src Node**; if it receives an ACK at the state TEAR, it will return back to the calling routine **State Block Killed Procedure at H-Src Node**;

- if it receives a down call SendModSAPU, it will start the routine **Reliable Delivery Procedure at H-Src Node**;
- if it receives a down call SendTearSAPU, it will start the routine **State Block Killed Procedure at H-Src Node**;
- if the resending timer expires, the state block has to recall the routine **Sending A CSTP Message Procedure**;
- if the resending counter reaches its limit, the state will issue an interface up call SendFail to the ALSP, and then it will kill itself.

Refreshment Procedure at H-Src Node The sending soft state block at an h-src node will periodically send a CSTP message xSig(REFRESH) to the IP-destination. If it receives a down call SendModSAPU, it will call the routine **Reliable Delivery Procedure at H-Src Node**; if it receives a down call SendTearSAPU, it will call the routine **State Block Killed Procedure at H-Src Node**.

State Block Killed Procedure at H-Src Node This procedure starts by receiving a SendTearSAPU from an ALSP. If a sending soft state block receives this call, it sets the state life timer and also calls the routine **Reliable Delivery Procedure at H-src Node**. If the called routine returns a result of true, or if the state life timer expires, the state block kills itself immediately.

Sending A CSTP Message Procedure When this routine is called, a SAPU is passed in. If the SAPU is less than or equal to an MTU size, it will be joined by a CSTP message header and will be sent to the h-sink node via the Multiplexing process and the Lower_Layer module. Otherwise, it has to be fragmented into pieces that are less than an MTU size. Then the routine adds a CSTP message header on each fragmented SAPU and sends them to the h-sink node via the Multiplexing process and the Lower_layer module.

4.3.2.2 Trigger Receiving Soft State Block: TrgRecvFSM

The Trigger Receiving Soft State Block, TrgRecvFSM, is an implementation of Figure 6. It is used to reliably deliver an SAPU and maintain the receiving soft state. Basically, it has three major services:

- reassembling the fragmented CSTP message if the incoming messages are fragmented messages;
- passing the incoming SAPU or SAPUid to the ALSP;
- keeping itself alive by periodically receiving a refreshing message. However, if the refreshing timer expires, it will send an interface call IntTearSAPU to the ALSP. Then it will send a removing registration request to the process StateManager. After that, it will kill itself.

In some sense, the Trigger Receiving Soft State Block can be treated as a group of predefined routines connected or triggered by incoming CSTP messages. Triggered by an CSTP message xSig(NEW), the process Demultiplexing at an h-sink node creates an instance of TrgRecvFSM; then it forwards the incoming xSig(NEW) to the instance. At this time, the routine **Reliable Delivery Procedure at H-Sink Node** is called. From now on, the following routines will be ready to be called until the instance is killed.

Reliable Delivery Procedure at H-Sink Node This procedure starts from the receipt of a CSTP trigger message. If the incoming message is not fragmented, the routine will remove the CSTP message header and send the incoming SAPU to the local ALSP by an up call. Then it will send an ACK back. If the incoming message is xSig(TEAR), it will call the routine **State Block Killed Procedure at H-Sink Node**. If the incoming message is fragmented, it will create a process instance of Reassembly Process, register the instance, and forward all the fragmented messages with the same SAPUid into the instance. If a Reassembly Process instance returns a completed CSTP message, it will retrieve the SAPU from the message, pass it to the

local ALSP by an up call. and then send an ACK back; if the instance returns the result of false, it will remove the registration of the instance.

Refreshment Procedure at H-Sink Node Once a receiving soft state block at an h-sink node gets an xSig(REFRESH) message, this routine will be started. First of all, the state block need to restart the state life timer right away to keep itself alive. Before its state life timer expires, if it receives an xSig(REFRESH), it will restart the timer; if it receives an xSig(NEW) or an xSig(MOD), it will restart the timer and call the routine **Reliable Delivery Procedure at H-Sink Node**; if it receives an xSig(TEAR), it will restart the timer and call the routine **State Block Killed Procedure at H-Sink Node**. If the timer expires, it will also call the routine **State Block Killed Procedure at H-Sink Node**.

State Block Killed Procedure at H-Sink Node If the state life timer expires, the soft state block will send an up call IntTearSAPU to the local ALSP and then kill itself; if the soft state block receives a CSTP message xSig(TEAR), it will send an ACK back and an up call RecvTearSAPU to the ALSP. Then, it will kill itself.

Reassembly Process When receiving the first incoming fragmented CSTP message, the process will set the reassembly timer and wait for all the rest of the fragmented messages with the same SAPUid. If it gets all the fragmented messages before the timer expires, it will reset the timer and reassemble them into one CSTP message, then it will return the message to the calling routine. If the timer goes off, it will discard all the received fragmented messages, and return the result of false to the calling routine. After it returns any result, it will kill itself immediately.

Here, we conclude a simplified algorithm for reassembly.

```
if the timer goes off
   then sends back a NACK;
if a message offset = 0
   then this message is the first fragment;
```

```
if a message MF = '1'
   then this message is the last fragment;
if (a message N's offset + its length
      == another message M's offset)
   then message M is the next to message N;
if (the total received message length
       == last Message's offset + its length)
   then all the fragmented CSTP messages have arrived,
        sends back an ACK,
        reorders and composes the fragmented CSTP message together.
```

Fragmentation and reassembly only suit for NEW, MOD and EVENT CSTP messages, because the size of the rest of the CSTP messages are usually less than an MTU size.

4.3.2.3 Event Sending/Receiving Soft State Block: EvtSendFSM and EvtRecvFSM

Following the CSTP specification, we have developed CSTP Event Messages State Transit Diagram at an h-src node (Figure 16) and at an h-sink node (Figure 17). Figure 16 is identical to part of the reliable delivery of Figure 5, while Figure 17 is identical to part of the reliable delivery of Figure 6. Meanwhile, Event Sending Soft State Block EvtSendFSM is the implementation of the Figure 16, while Event Receiving Soft State Block EvtRecvFSM is the implementation of Figure 17. To avoid repetition we omit the details of the two soft state blocks here.

4.3.3 Transmission Processes

The Multiplexing process multiplexes all CSTP messages incoming from each local soft state. All CSTP messages have to be less than or equal to an MTU size. In this process, multiplexing would bundle multiple individual CSTP messages into one

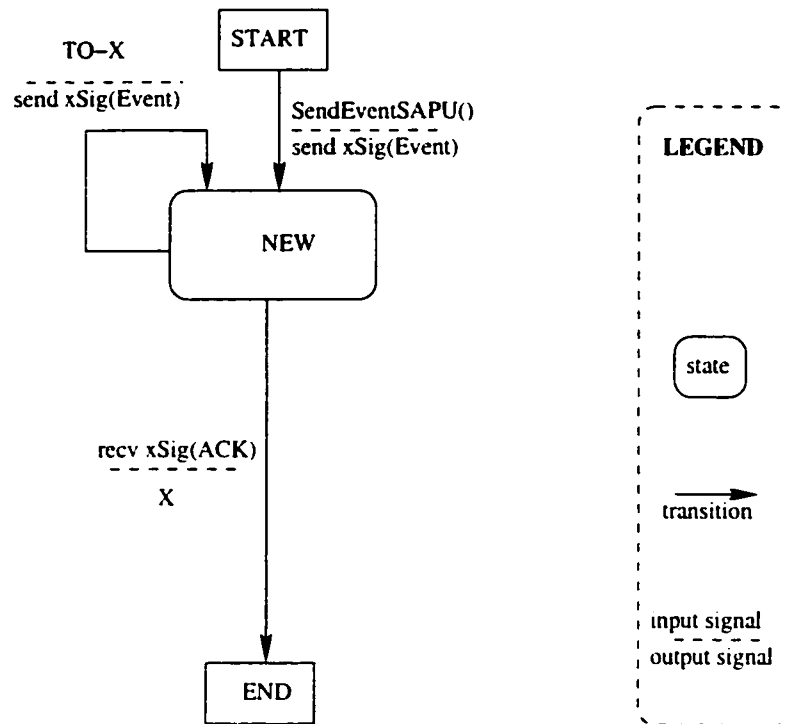


Figure 16: H-Src CSTP Event Messages State Transit Diagram

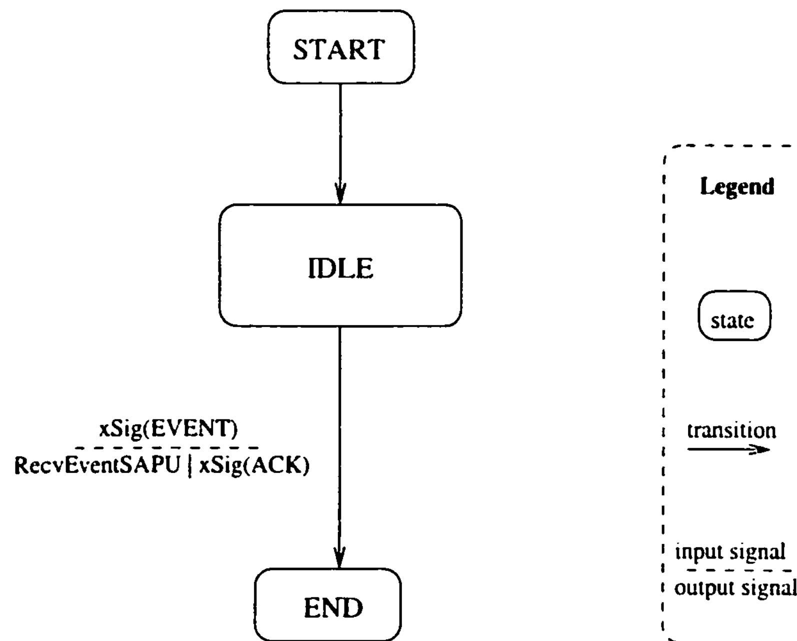


Figure 17: H-Sink CSTP Event Messages State Transit Diagram

CSTP bundled message, based on the indication of the Burst_Flag in the incoming CSTP messages. However, we only add a bundling header on each CSTP message rather than bundling more individual CSTP messages together in this model. After bundling, the Multiplexing process passes the CSTP messages through the local Lower_layer module to the h-sink node.

The Demultiplexing process demultiplexes all the incoming CSTP messages from the Lower_layer module. It has two major tasks: to remove the bundling header B-header and to send each CSTP message to its corresponding soft state block. Before it sends a message to a state block, it has to query the soft state block process ID (PID) by a remote procedure call. If the destination block is alive, the process will get the PID and then send the incoming message to the corresponding soft state block. If the destination block is dead, the process will get nothing, and have to take the following actions:

- if the incoming message is an xSig(NEW) or an xSig(EVENT) message, the process will create a soft state block for it. Then, this process will send a registration signal to the process StateManager for registering the new state block. After that, the process will forward the incoming message to the newly created state block.
- if the incoming message is an xSig(TEAR), or an xSig(ACK), or an xSig(NACK) message, the process will discard it.
- if the incoming message is an xSig(REFRESH) message, the process will send back a NACK to the xSig(REFRESH)'s originator through the local Multiplexing process.

4.4 ALSP Module

The ALSP module is a test driver, which triggers CSTP to generate various scenarios for validation, and also processes ALSP/CSTP interface calls by simulating ALSP

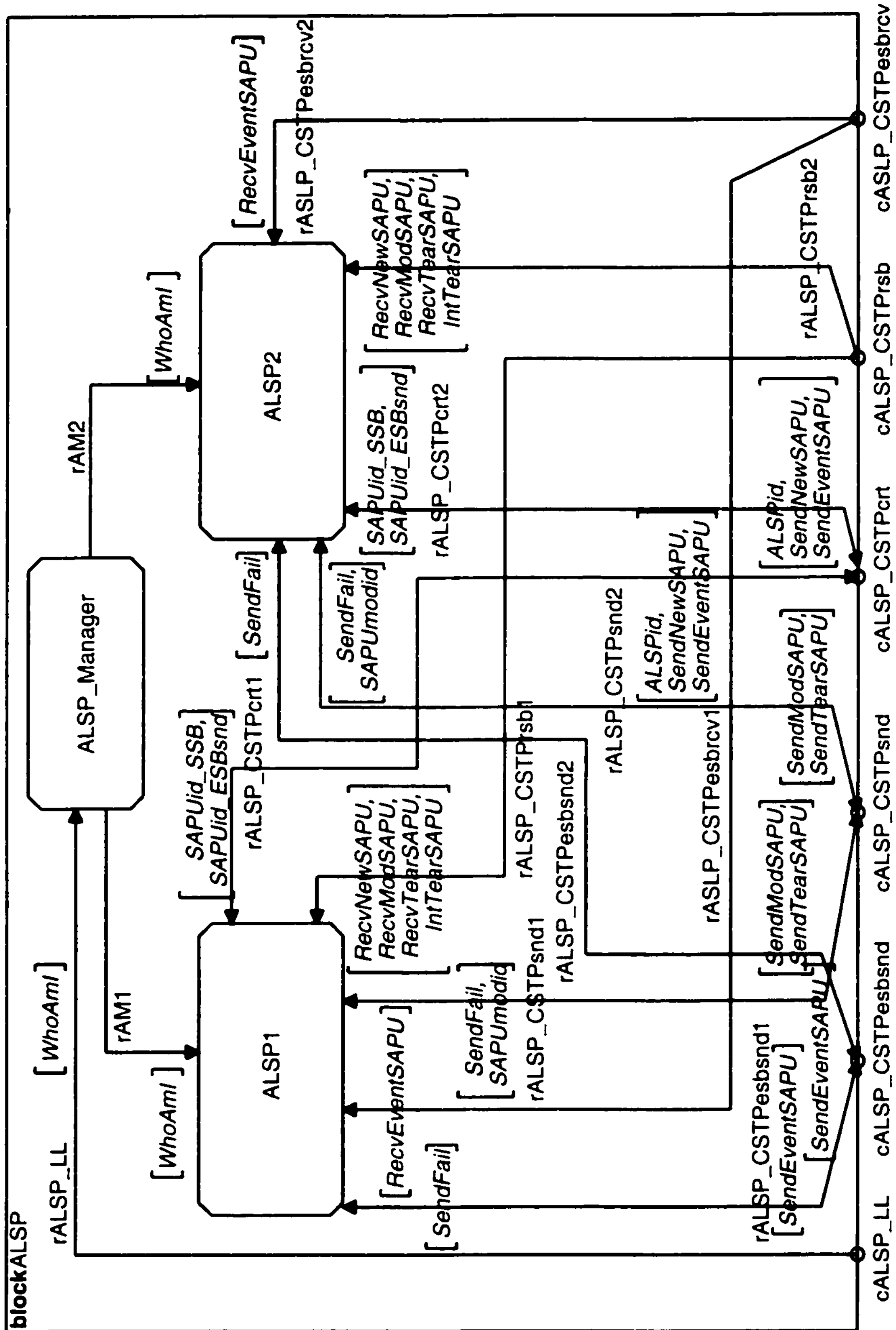


Figure 18: The ALSP Module

services. Our model has three nodes that have different tasks. The node HSrc simulates a sender; the node HSink simulates a router, and the node HDest simulates a receiver. Adapting to the different tasks of the ALSP module at each node, we design three kinds of ALSP modules: the ALSP module at HSrc, the ALSP module at HSink, and the ALSP module at HDest.

Out of 2LAIS, at ALSP level at a node, the ID permits the situation that multiple ALSP 'module's coexist. In order to avoid confusion, we use the concept 'process' in the model to express the concept 'module' specified in the ID. No matter how many ALSP 'process'es there are, each ALSP 'process' has to communicate with CSTP by the standard ALSP/CSTP interface calls. We will use the model to validate two sets of scenarios: one way CSTP scenarios, and round trip CSTP scenarios. The first set is for the study of CSTP general features; the second set is for the study of an ISPS offering the simplest unicast reservation features of RSVP version 1. We use process ALSP1 as the test driver for the first set, and use process ALSP2 as the test driver for the second set. Besides the two ALSP 'process'es, the ALSP module has a process ALSP_Manager. It is designed to manage the two process ALSP1 and ALSP2, as shown in Figure 18. Process ALSP1 and ALSP2 should be able to query an outgoing interface for particular destination IP address. This service is implemented through a remote procedure call in the ALSP 'process's.

Let us survey the design of ALSP1 at each node first. In the model, HSrc is the sender. Thus, the ALSP1 at HSrc has to generate kinds of down interface calls to trigger the local CSTP to transmit messages to the receiver HDest, via the router HSink. The ALSP1 has to generate two sizes of SAPUs for each down call SendNewSAPU, SendModSAPU and SendEventSAPU: a SAPU that needs to be fragmented or a SAPU that does not need to be fragmented.

HSink is a router. Thus, the ALSP1 at HSink has to forward all incoming messages to the HDest. It also generates failures in forwarding the calls RecvNewSAPU, RecvModSAPU, then sends an EVENT SAPU to the incoming SAPU's originating sender. This implies that an error message is sent to the originating sender of the typed incoming SAPU while the ALSP1 processes the typed SAPU and an error is encountered.

HDest is the receiver of the incoming SAPU. Because we only set up a one way virtual path, we do not need the ALSP1 at HDest to do more work. Unlike the ALSP1 at HSink, the ALSP1 at HDest does not need to forward the received SAPUs. Considering that HDEST could not be a host, for general purpose, we won't let HDEST interact with the environment.

The design of ALSP2 at each node is similar to the ALSP1 at each node. As we mentioned, this set of test drivers is to match the simulation of ISPS offering the simpliest unicast features of RSVP version 1. In each ALSP2, we only need to set up necessary soft states for PATH and RESV SAPUs, as well as to take simplified algorithms rather than the real complex algorithms. To implement this goal, we design the following scenario. At HSrc, the ALSP2 composes a PATH SAPU, creates a state for it, and then sends the PATH SAPU to HDest via HSink. Once HSink receives this SAPU, it sets up a state for it, modifies necessary information on the incoming SAPU, then forwards the modified SAPU to HDest. Once HDest gets this SAPU, similar to ALSP2 of HSink, it also sets up a state for this incoming SAPU. Then, it composes a RESV SAPU, creates a state for it, and then sends the RESV SAPU to HSrc via HSink. At HSink, we use a random function to simulate two cases: router HSink has enough resources for the request from the RESV SAPU, or it has not. In the first case, the ALSP2 of HSink will forward the RESV SAPU to HSrc. In the second case, the ALSP2 of HSink will compose an EVENT SAPU and send it back to the HDest. If HSrc receives the RESV SAPU, this means that HDest succeeded in resource reservation; if HDest receives the EVENT SAPU, this means that HDest failed in resource reservation.

According to the orignal ID, a SAPU can be treated as a $(<key>, <value>)$ pair. To manage states at each ALSP2, we design a set of data structures. The following gives the major ones.

```
SAPUKey_t ::= SEQUENCE {
 hsrc               IPADDR_t,
 hsrcport           PORT_t,
 IPtarget           IPADDR_t,
 targetport         PORT_t,
```

```
 SAPUtype           SAPUtype_t
};

UniSAPUid_t ::= SEQUENCE {
 SAPUid            SAPUid_t,
 hsrc              IPADDR_t
};

NEWTYPE SAPUKey_SAPUid_tbl_t
 ARRAY(SAPUKey_t, UniSAPUid_t)
ENDNEWTYPE;

NEWTYPE SAPUid_SAPUKey_tbl_t
 ARRAY(UniSAPUid_t, SAPUKey_t)
ENDNEWTYPE;

NEWTYPE SAPUKey_Set_t
 POWERSET(SAPUKey_t)
ENDNEWTYPE;

NEWTYPE SAPUKey_PHOP_tbl_t
 ARRAY(SAPUKey_t, IPADDR_t)
ENDNEWTYPE;
```

In the above, SAPUKey_t is a type of *key* in a SAPU. We use this data struture to represent a state, and use a powerset SAPUKey_Set_t to manage all the states. When a state is created, it will be added into the powerset; when a state is removed, it will be removed from the powerset. The type UniSAPUid_t identifies a SAPUid globally. The table structures SAPUKey_SAPUid_tbl_t and SAPUid_SAPUKey_tbl_t set up a two-way connection between SAPUKey_t and UniSAPUid_t. To help a SAPU transmitted along a signaled path, the structure SAPUKey_PHOP_tbl_t is necessary.

4.5 Lower_layer Module

The Lower_layer module is an abstraction of OSI lower three layers' services relevant to CSTP. As Figure 19 shows, there are four processes inside. The process Initialization is used for the whole model initialization. It sends routing information and helps each node set up routing and other information. The process Routing_Manager responds to routing queries. The process Transmitter only transmits the internal signals to a particular out-going interface, while the process Rcv_in just receives outside signals from the interface. Moreover, the process Rcv_in simulates packet loss at a given packet loss rate. However, we will not consider all corruption packets, because a router probably can not tell a corruption packet's source address and destination address. In practice, people just discard them. As for the packet delay simulation, we take advantage of function from our simulation tool ObjectGEODE, whose component channel can offer this simulation.

This module can help us examine the relationship among packet loss rate, CSTP message fragmented pieces, resending timer and resending counter limits.

4.6 Modeling Constraints

4.6.1 Probabilistic Decisions

Standard SDL does not provide a direct method for the description of probabilistic actions. Simple description of probabilistic actions can be done in SDL by choosing a random number out of a defined range of integer numbers (using the *any* construct) and then using it to take decisions based on probabilities. Obviously, this method is hard to understand. More complex probability distribution functions are much harder to describe and the description using SDL may not represent the real function.

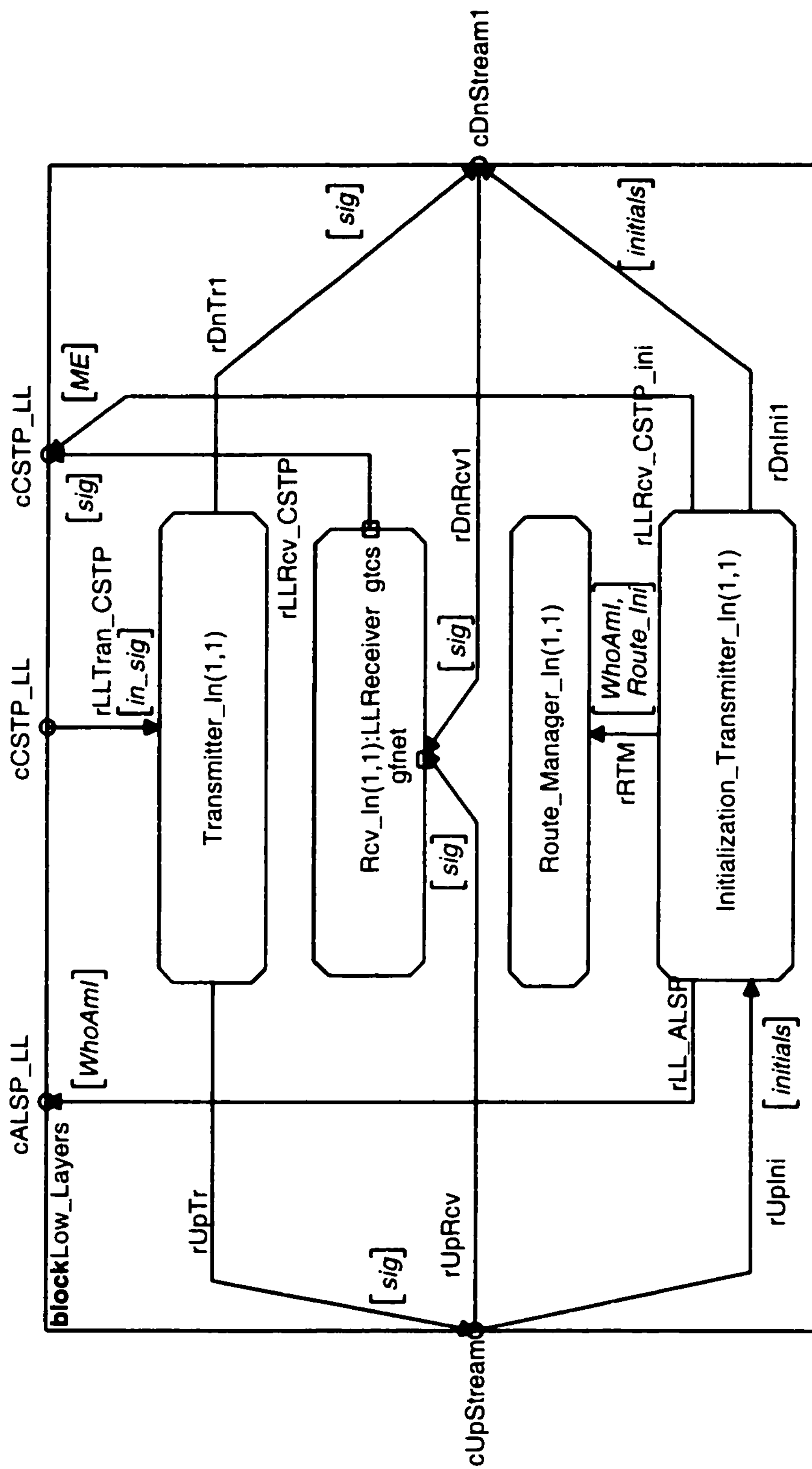


Figure 19: The Lower_layer Module

4.6.2 Timing

SDL does not allow for the definition of probabilistic time intervals and hence we cannot model complex scheduling problems for CSTP in SDL. Although messages may be sent in a certain sequence through a channel, there is no way to specify an upper bound on the the delay bound that they will occur in the channel. Moreover, there is no guarantee that they will arrive in the same sequence.

4.6.3 Resource Specified

As a high level language, SDL does not specify resources, which are assumed unlimited. SDL process have unlimited queues and SDL channels are either assumed to have random delay or no delay at all. Unlimited queues simplify modeling, but they make it difficult to define behaviours that are based on resource overflow. There is no definition what will happen if the queues are full or if the channels are blocked or have no available bandwidth. Unlimited queues also make verification more difficult as they cause the generation of a larger number of states. SDL has no means to specify some very specific resource characteristics such as channel error rate or delay jitter.

4.6.4 Signal Priority

SDL has the capability to define the system stimuli as a signal, but it provides no means to describe the timing characteristics of this signal. In addition, SDL provides only two levels of priorities to the signals, which makes it difficult to describe systems with multiple levels of priority. This may cause the obtained system behaviour to be incomplete.

Chapter 5

Validation of CSTP

5.1 Tasks and Techniques

Historically, the meaning of *verification* and *validation* is often confused in the field of protocol simulation. Some define the *verification* as to verify protocol general properties, such as the absence of deadlock, unspecified reception and live locks. And they define the *validation* as to validate specific properties of a protocol against the specification requirements [1]. Some just reverse the above definitions of *verification* and *validation* [27]. Practically speaking, *verification* and *validation* use nearly identical techniques and the boundary between them becomes somewhat vague. In his book, Holzmann does not distinguish between the two definitions and speaks only of *validation* [19]. Basically, we prefer to take this approach.

The design of a protocol is an iterative work. Currently, the design of CSTP is just at the beginning. Thus, the major tasks for the CSTP simulation are to detect any ambiguities in an informal protocol description in English, to validate certain scenarios, and at the same time to verify the correctness of general properties, such as deadlock, livelock, unexpected signals, etc.. As we mentioned before, we will validate two sets of scenarios. One set includes the basic CSTP scenarios, such as SendNewSAPU. The other set implements the simplest unicast features of RSVP version 1 by CSTP and its drivers.

Different tasks need different techniques. Generally speaking, the techniques used in validation are compilation, interactive simulation, random simulation and exhaustive simulation with the help of MSC observers.

The static correctness can be verified by means of the compilation of the SDL specification with ObjectGEODE Simulation Builder. Since SDL is a strongly typed language, all static errors can be detected during compilation. The absence of syntax errors does not necessarily imply that the protocol will do what it is supposed to do. With the help of interactive and random simulation, run-time errors can be quickly discovered but the logic of the protocol is usually not taken into account. In order to prove that certain undesirable properties are absent, exhaustive simulation is the major technique. It can detect deadlocks, livelocks, the parts of the model that are never executed, unexpected signals and queue overflow. With the help of MSC observers, which describe the scenarios that the specification must respect, exhaustive simulation can prove that an SDL model supplies the service (expressed in the requirements specification) correctly.

Writing observers consists in translating a service and the properties expected from the system. This service and properties generally are expressed informally in the requirements specification. This phase is essential as it allows design errors to be detected very early, and it can make sure whether the SDL specification can correctly fulfil the requirements. The ObjectGEODE simulator permits us to assign various observers at the same time. The properties in MSC observers are checked automatically at each simulation step. If the simulation ends without errors detected, the properties are never violated for the all paths explored in the model. The service described is therefore validated for this set of paths. Otherwise, interactive simulation will be used to replay error scenarios, which are generated by exhaustive simulation, in order to detect the errors. In the following sections, we will introduce the works to validate the two sets of scenarios as well as the validating results.

5.2 Validation of Set One Scenarios of CSTP

In this part, we mainly use exhaustive simulation with the help of MSC observers to validate basic CSTP scenarios. Based on the analysis of CSTP, we conclude four basic CSTP scenarios:

- *Send New SAPU*
- *Send Mod SAPU*
- *Send Tear SAPU*
- *Send Event SAPU*

Essentially, the first three scenarios need not only reliable delivery, but also refreshment. The last one only needs reliable delivery. As CSTP behaviours are hop-by-hop, we only need to validate these scenarios between the CSTP neighbour pair HSrc and HSink. Thus, HSrc is the h-src node, and HSink is the h-sink node. We are going to illustrate these scenarios and give the validation plans.

5.2.1 Scenario: *Send New SAPU*

This scenario starts from the ALSP1 issuing a down call SendNewSAPU to the process StateManager at CSTP level of an h-src node. Triggered by this call, StateManager generates an unique SAPUid for the SAPU carried in this call. Then it creates a trigger sending soft state block TrgSendFSM for this SAPU, and forwards this call to the state block created. Finally, it returns the SAPUid to the ASLP1.

After being created, the local trigger sending soft state block TrgSnedFSM builds and sends the trigger message xSig(NEW) to the target IP address. Then it sets the resending timer and resending counter. If the SAPU size is less than an MTU size, it will directly be composed into a CSTP message xSig(NEW). Otherwise, the SAPU will be fragmented and composed into individual CSTP messages, which are sent out

later. If the resending timer goes off before the xSig(NEW) message is acknowledged, the local CSTP will transmit the trigger message again. This procedure will be repeated until either an ACK is received or the resending counter reaches its limit. In the latter case, the state block will issue an up call SendFail to the ALSP and kill itself. If the state block receives an xSig(ACK) message, it will stop retransmission and start to send a periodic refresh message xSig(REFRESH) to the IP-target. During the refreshment, if the state block receives an xSig(NACK) message, it will return to the step to retransmit the trigger message.

When the xSig(NEW) message is received at the h-sink node, the local Demultiplexing module at CSTP of the h-sink node will create a local trigger receiving soft state block TrgRecvFSM. The state block will pass the SAPU to the local ALSP by an up call RecvNewSAPU; then it will return an ACK message to the CSTP at HSrc.

To fully validate the scenario *Send New SAPU*, we let the driver ALSP1 of HSrc produce two sets of scenarios: SAPU without fragmentation and SAPU with fragmentation. For the scenario without fragmentation, we assign the resending counter limit to three. Resulting from this, there are four cases:

1. the transmission succeeds in sending xSig(NEW) the first time;
2. the transmission succeeds in sending xSig(NEW) the second time;
3. the transmission succeeds in sending xSig(NEW) the third time;
4. the transmission fails in sending xSig(NEW) after the third time delivery.

Among the above cases, the first three are successful scenarios, and the last one is a failure scenario. For each of them, we assign two level observers: at block level and at process level. To simulate the state lost at an h-sink node, we let the process Demultiplexing generate the CSTP message xSig(NACK) only at the first time it receives the xSig(REFRESH). Figure 20 depicts a block level observer of the first successful scenario, and Figure 21 depicts a block level observer of the failure scenario. As the real signals in MSC observers have long signature, they are not read easily. We use the simplified signals instead of the whole signals in the figures.

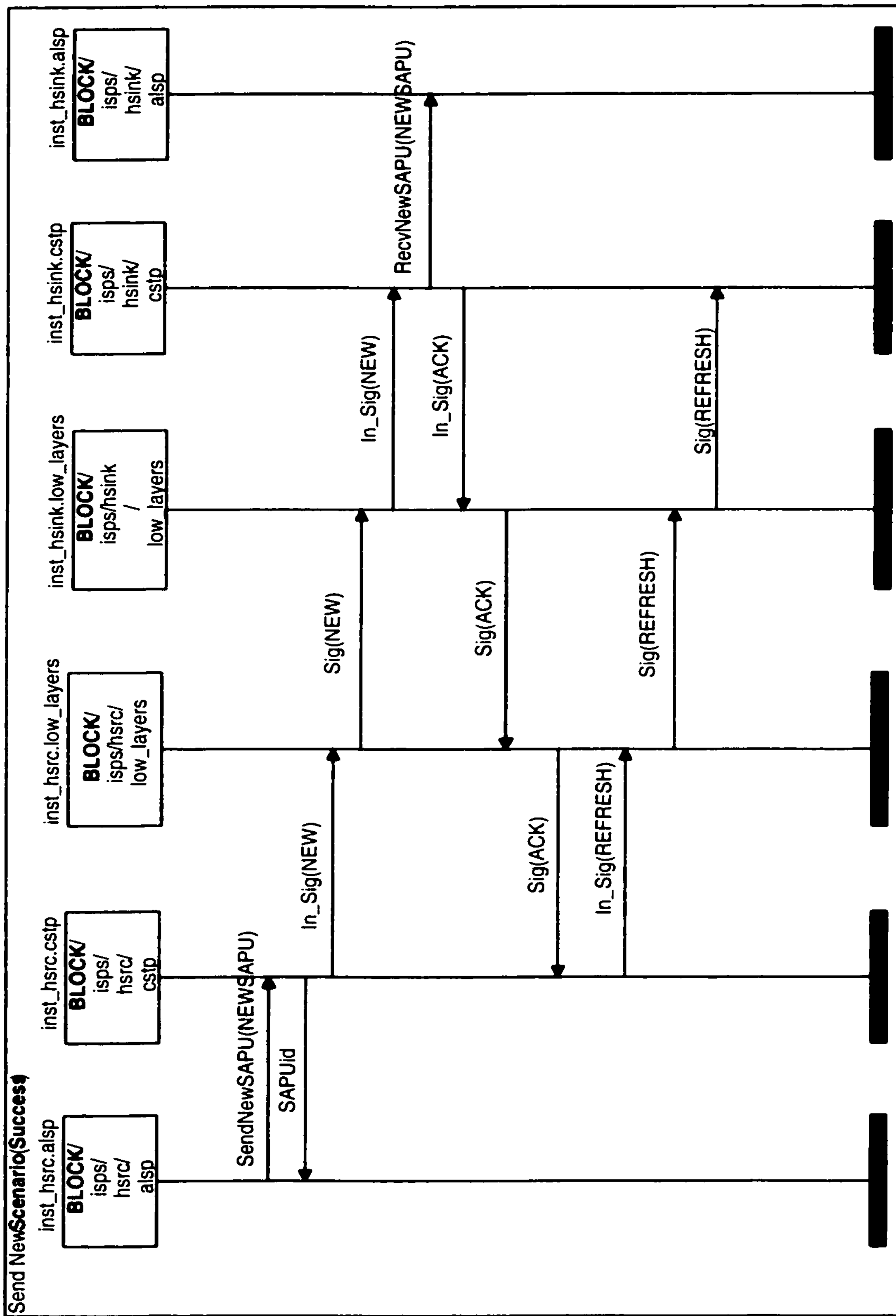


Figure 20: A Successful Scenario for Scenario *Send New SAPU*

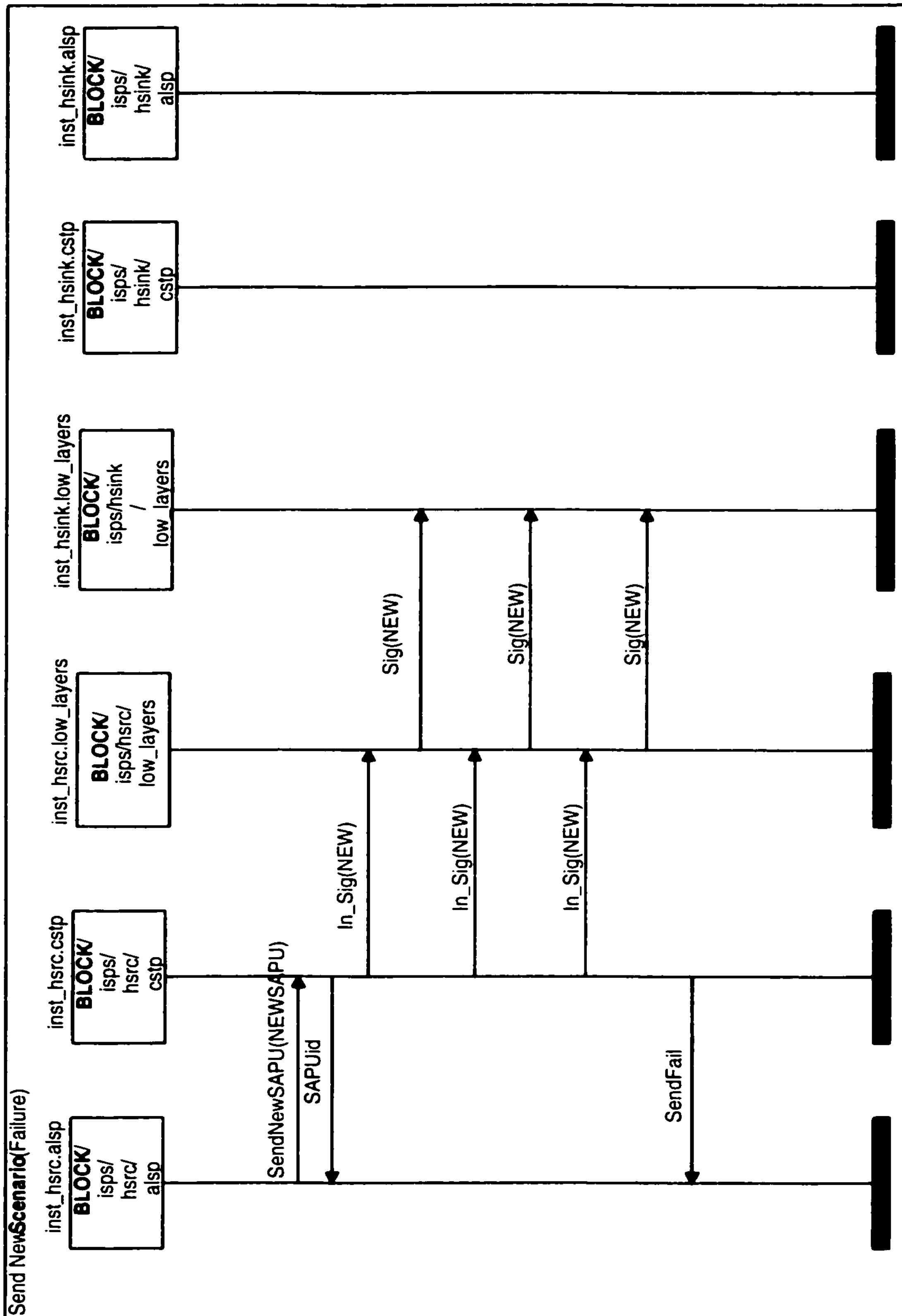


Figure 21: A Failure Scenario for Scenario *Send New SAPU*

Compared to the scenario without fragmentation, the scenario with fragmentation is much more complex at process level at HSink, but the rest is almost the same. This is because of packet loss. At the h-sink node, CSTP may not receive all the fragments belonging to the same SAPU during a certain period. Let us say that the SAPU is fragmented into three pieces. Resulting from this, there are eight cases at the h-sink node. For example, the CSTP at the h-sink node receives all the three fragmentation messages or nothing during a certain period. To validate these cases, we need to assign a different observer for each case.

5.2.2 Scenario: *Send Mod SAPU*

The authors of CSTP proposed a discussion topic: "whether CSTP needs xSig(MOD)? Is there any advantage to keep MOD, instead of using NEW only?" Actually, the scenarios of delivery NEW and MOD are extremely similar. The major difference is the happening time. At the very beginning, CSTP delivers NEW. After that, CSTP could deliver MOD or NEW based on the trigger conditions.

Thus, to validate the scenario *Send Mod SAPU*, we will mainly use the similar techniques and observers discussed in the above subsection. We also let the driver ALSP1 of HSrc produce two sets of scenarios: SAPU without fragmentation and SAPU with fragmentation. For the scenario with or without fragmentation, scenario *Send Mod SAPU* has the same cases as the scenario *Send New SAPU*. To avoid repetition, we omit the repeated parts. As a reference, we give two figures of this scenario. Figure 22 depicts a block level observer of the first successful scenario, and Figure 23 depicts a block level observer of the failure scenario. For clearness, we list all the cases of this scenario below:

1. the transmission succeeds in sending xSig(MOD) the first time;
2. the transmission succeeds in sending xSig(MOD) the second time;
3. the transmission succeeds in sending xSig(MOD) the third time;
4. the transmission fails in sending xSig(MOD) after the third time delivery.

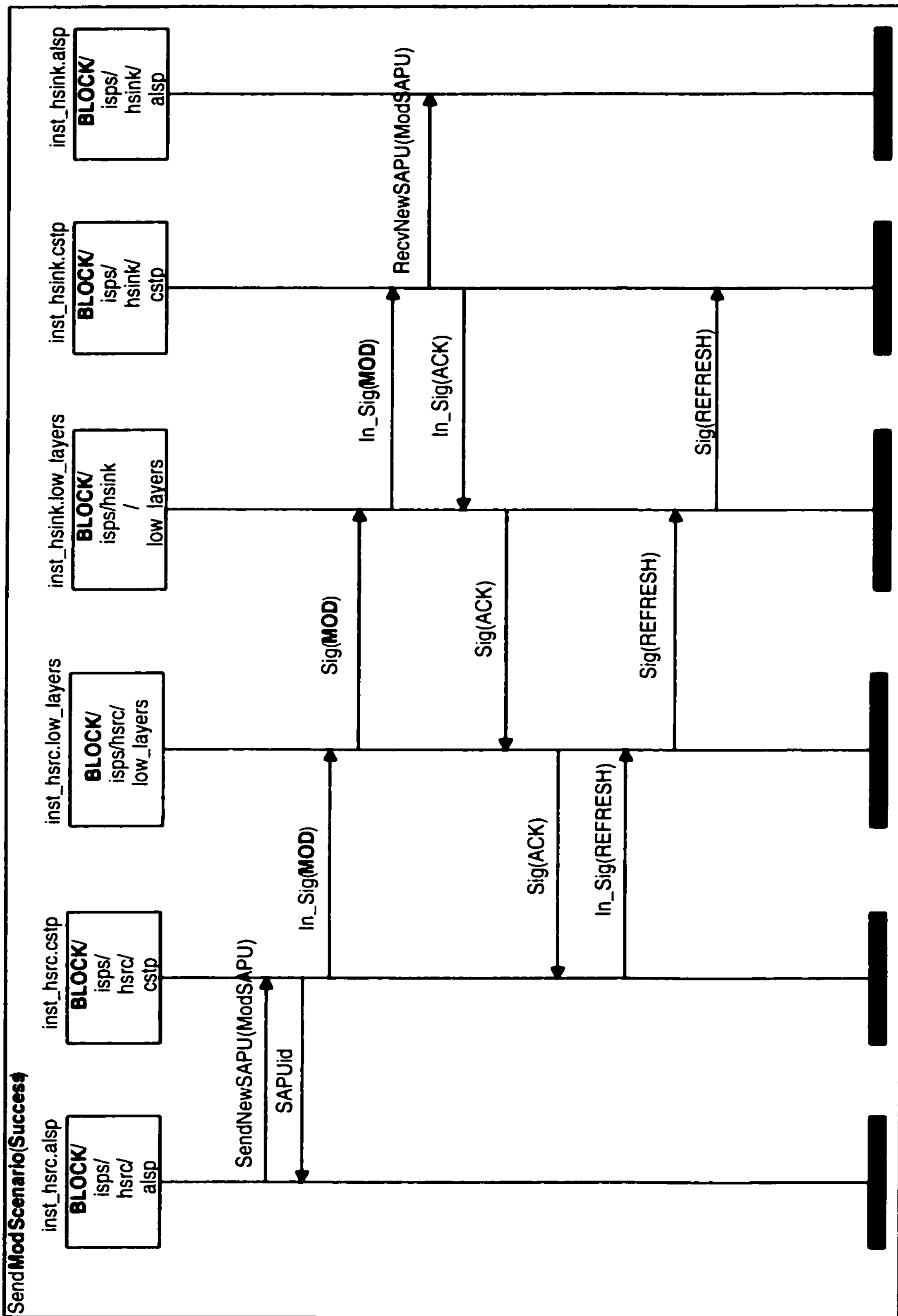


Figure 22: A Success Scenario for Scenario *Send Mod SAPU*

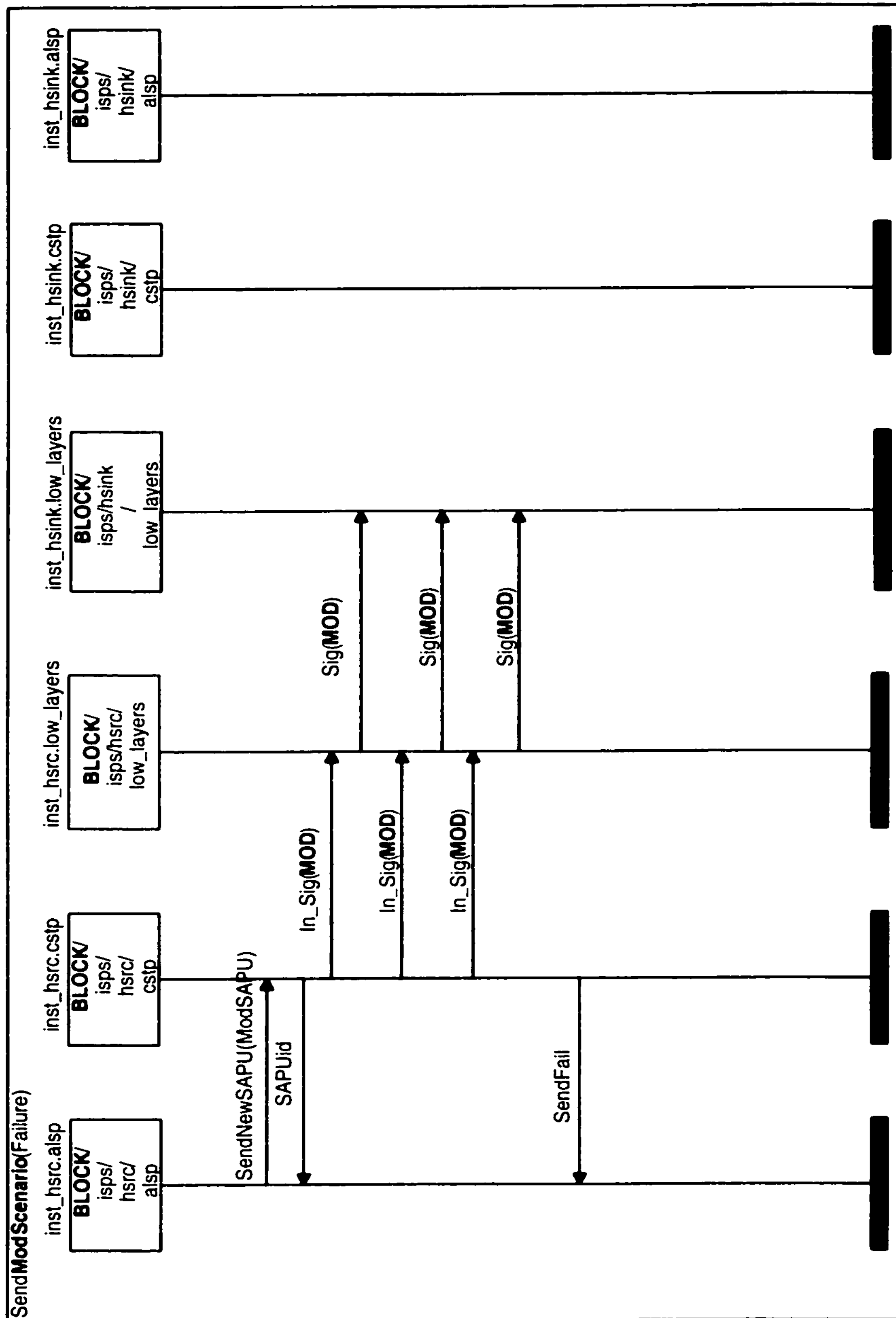


Figure 23: A Failure Scenario for Scenario *Send Mod SAPU*

5.2.3 Scenario: *Send Tear SAPU*

After the driver ALSP1 at HSrc issues a down call SendNewSAPU, it can issue a down call SendTearSAPU at any time. Once the trigger sending soft state block at the CSTP of HSrc receives a SendTearSAPU, it composes a CSTP message xSig(TEAR) and sends it to HSink. Then it starts its tear timer. At any of the situations that it receives an ACK, or the tear timer expires, or the resending counter reaches its limits, it will kill itself immediately.

At the h-sink node HSink, if the trigger receiving soft state block receives a CSTP message xSig(TEAR), it sends back an xSig(ACK), sends an up call RecvTearSAPU to the local ALSP1, and then it kills itself immediately; if the state life timer goes off, it kills itself immediately.

From the above analysis, we can refine the following cases for this scenario on the condition that the resending counter limit is three:

1. after the CSTP of HSrc sends xSig(TEAR) the first time, the CSTP receives an ACK from HSink;
2. after the CSTP of HSrc sends xSig(TEAR) the second time, the CSTP receives an ACK from HSink;
3. after the CSTP of HSrc sends xSig(TEAR) the third time, the CSTP receives an ACK from HSink;
4. after the third time delivery, the resending timer expires and the resending counter reaches its limit;
5. the state life timer of the receiving soft state block at HSink expires.

The above first three are successful cases, and the last two are failure cases. Figure 24 is the observer of the first case; Figure 25 is the observer of the fourth case, and Figure 26 is the fifth case.

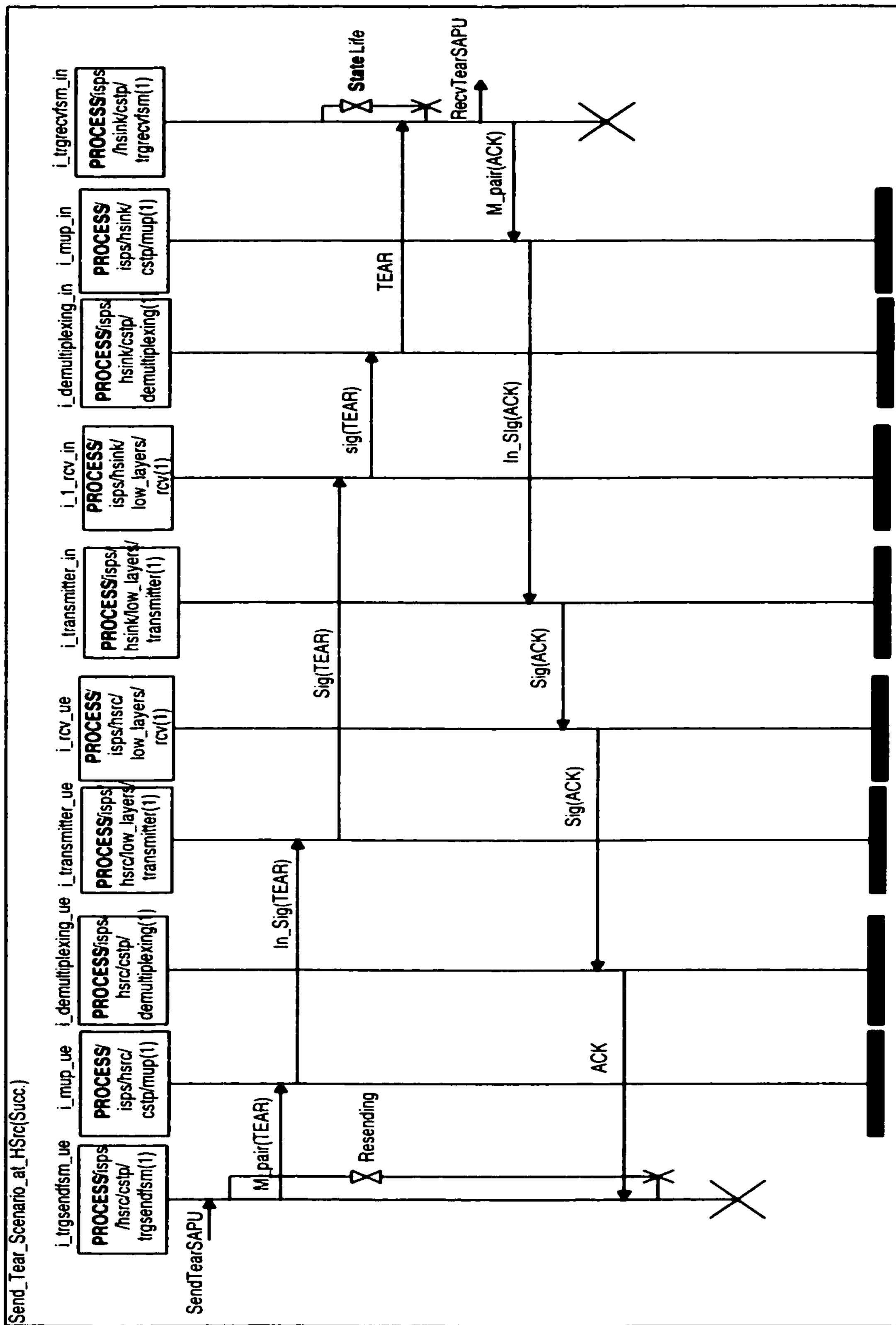


Figure 24: A Successful Scenario for Scenario *Send Tear SAPU*

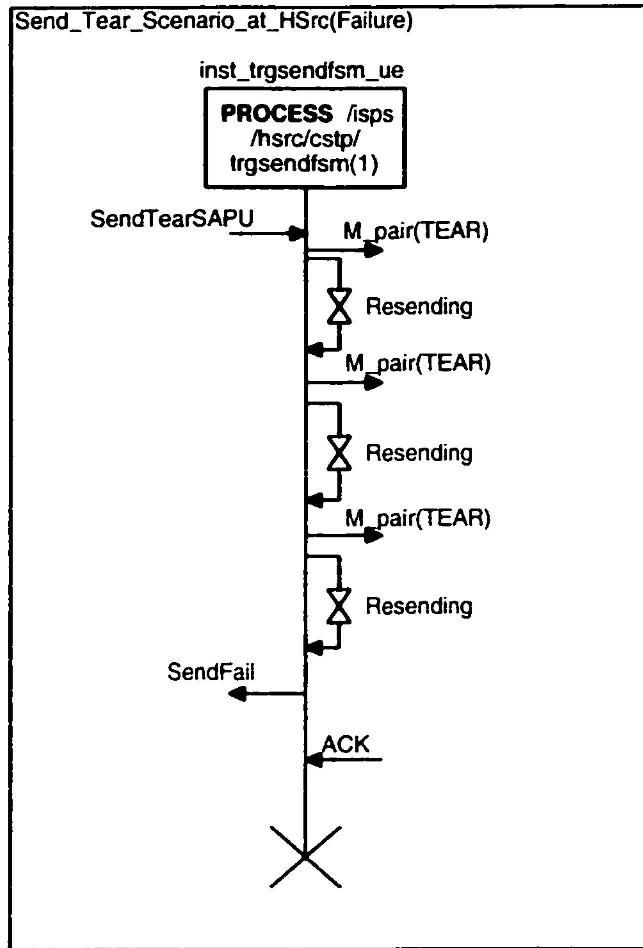


Figure 25: A Failure Scenario for Scenario *Send Tear SAPU*: SendFail

5.2.4 Scenario: *Send Event SAPU*

A CSTP EVENT message needs reliable delivery, but it does not need refreshing. Thus, the scenario of delivery EVENT is similar to deliver CSTP message xSig(NEW). The original CSTP ID does not give the possible size of an EVENT message. To completely simulate this scenario, we assume that the size of an EVENT SAPU could be larger than an MTU size. Thus, the fragment algorithm is used for EVENT SAPU as well.

Like the validation of scenario *Send Mod SAPU*, in order to verify the scenario *Send Event SAPU*, we still mainly use the techniques and similar observers discussed

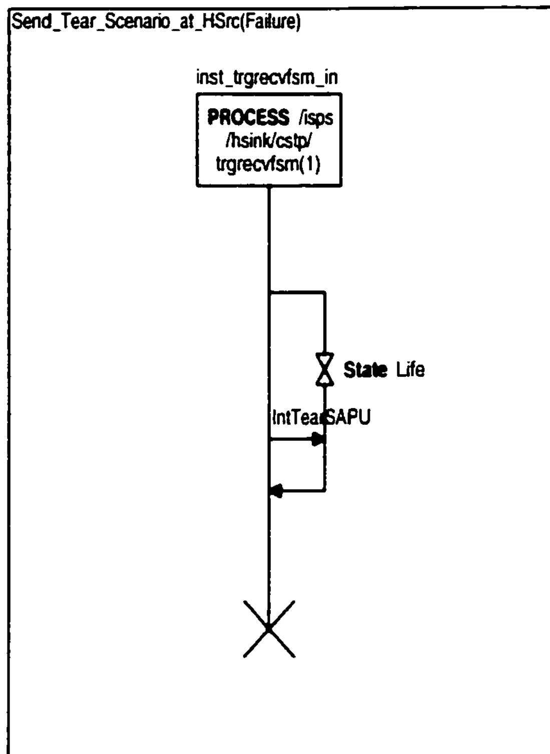


Figure 26: A Failure Scenario for Scenario *Send Tear SAPU*: State Life Timer Time-Out

in Subsection 5.2.1. For clearness, we list all the cases for this scenario below.

1. after the CSTP of HSrc sends xSig(EVENT) the first time, the CSTP receives an ACK from HSink;
2. after the CSTP of HSrc sends xSig(EVENT) the second time, the CSTP receives an ACK from HSink;
3. after the CSTP of HSrc sends xSig(EVENT) the third time, the CSTP receives an ACK from HSink;
4. after the third time delivery, the resending timer expires and the resending counter reaches its limit;

Figure 27 is the observer of the first case with fragmentation; Figure 28 is the observer of the fourth case with fragmentation.

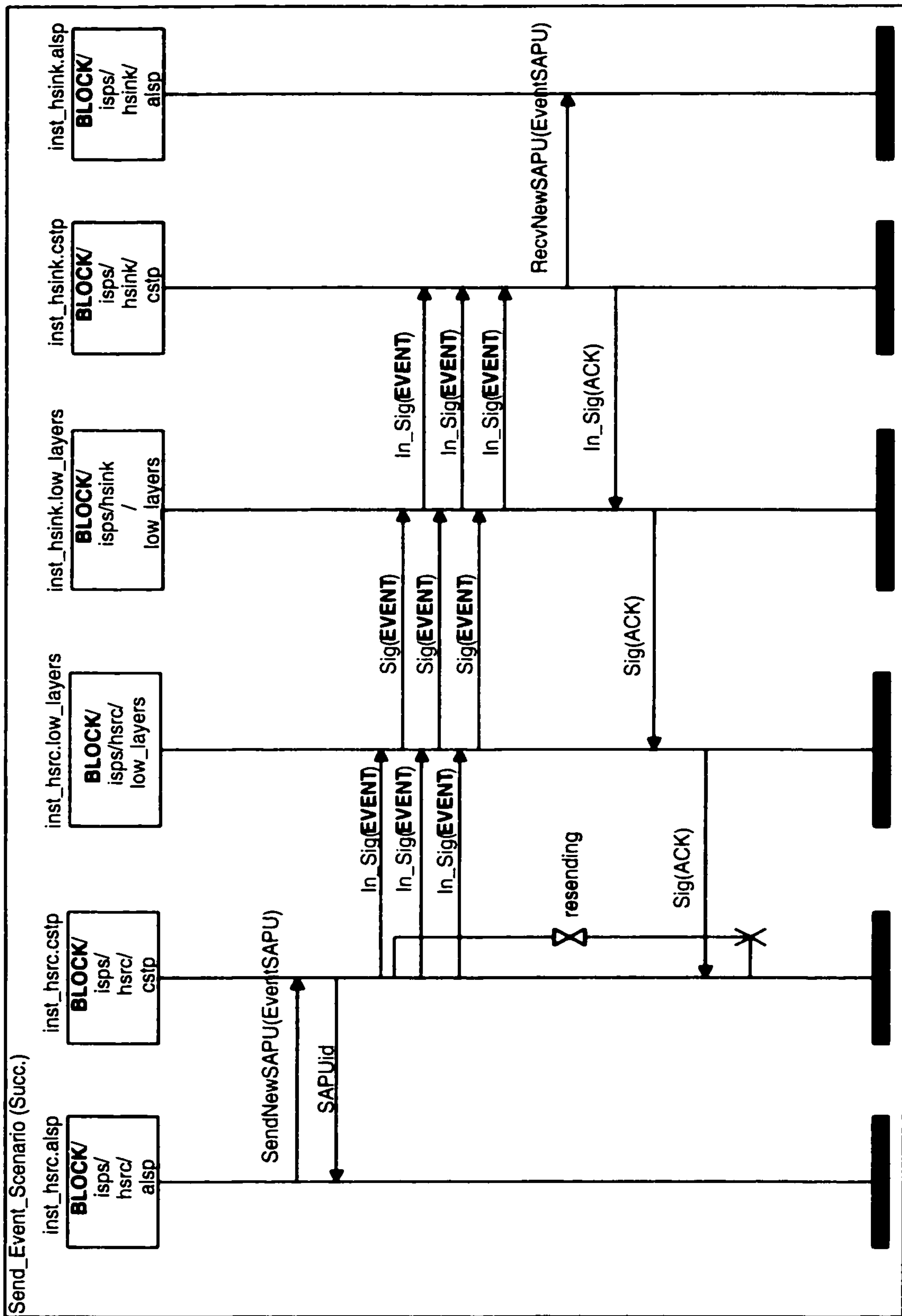


Figure 27: A Successful Scenario for Scenario *Send Event SAPU*

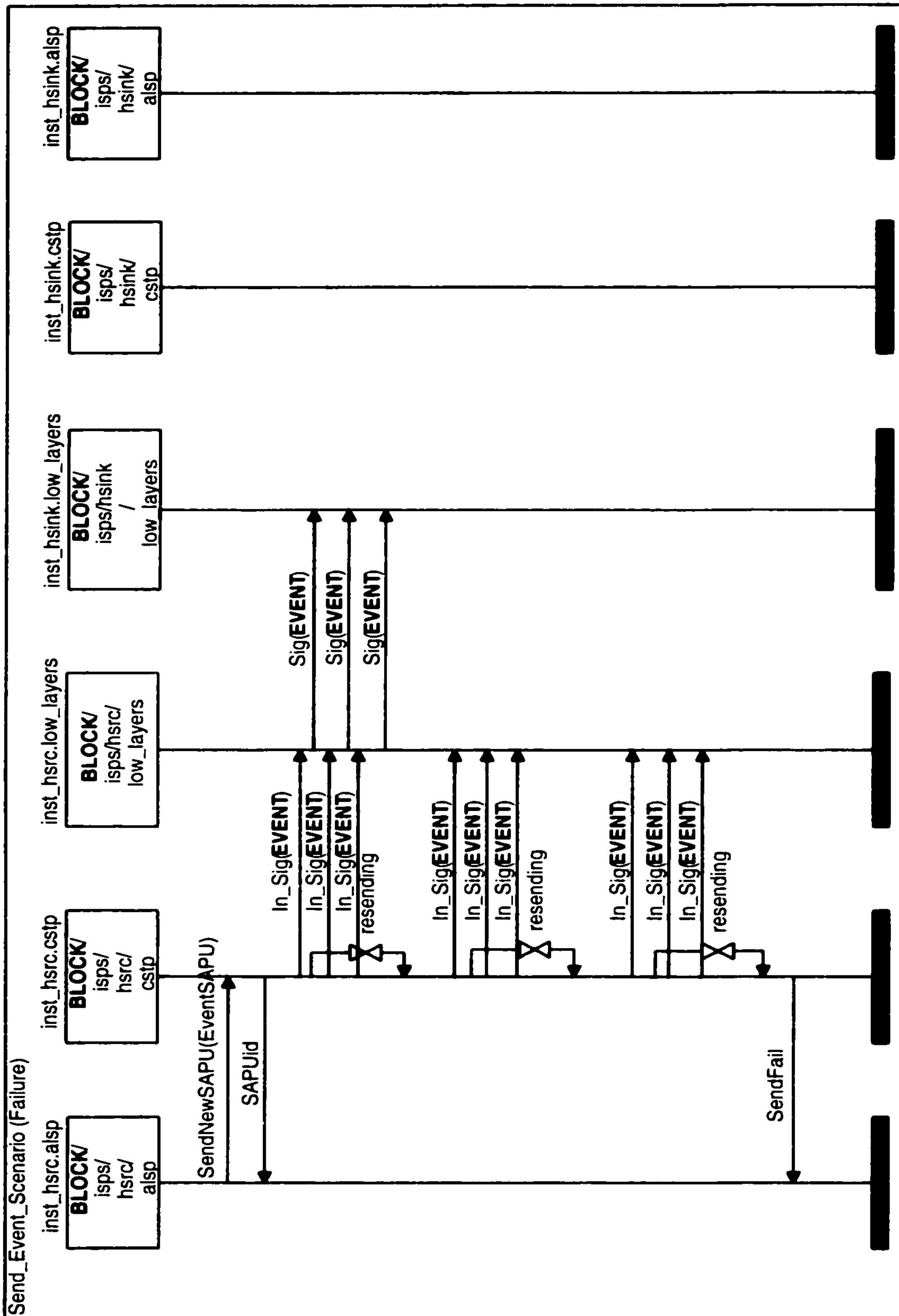


Figure 28: A Failure Scenario for Scenario *Send Event SAPU*

5.3 Validation of Set Two Scenarios of CSTP

As we mentioned before, this scenario is to simulate the behaviours of ISPS offering the simplest unicast features of RSVP version 1. Therefore, the ALSP2 of HSrc has to generate a PATH SAPU and send it to the ALSP2 of HDest, via HSink. When the ALSP2 of HSink receives the SAPU, the following actions will take place: if the PATH SAPU passes the relevant check, the ALSP2 will forward this SAPU to the HDest; otherwise, the ALSP2 will send an EVENT SAPU to the PATH SAPU's originator. Upon the receipt of the PATH SAPU at HDest, the ALSP2 will generate an RESV SAPU and send it to the HSrc by a down call SendNewSAPU. The RESV SAPU will be transmitted following the same path that the PATH SAPU came on. Upon the receipt of the RESV SAPU at HSink, similar to the results for processing the PATH SAPU, we randomly let the ALSP2 generate two results: if the RESV SAPU passes the relevant check, the ALSP2 will forward this SAPU to the HDest; otherwise, the ALSP2 will send an EVENT SAPU to the PATH SAPU's originator. The case that the RESV SAPU fails in checking means the reservation fails. When either the HSrc receives the RESV SAPU or HDest receives the EVENT SAPU, the scenario ends.

To focus on observing the behaviours we describe above, we let all the SAPU sizes be less than an MTU, and assign the packet loss rate to zero. Resulting from these conditions, there is no retransmission in this validation. Thus all possible scenarios are:

1. HSrc succeeds in delivering a PATH SAPU to HSink; HSink succeeds in forwarding the PATH SAPU to HDest; HDest then succeeds in delivering a RESV SAPU to HSink; HSink succeeds in forwarding the RESV SAPU to the HSrc;

2. HSrc succeeds in delivering a PATH SAPU to HSink; HSink succeeds in sending back an EVENT SAPU to HSrc because of the failure of PATH SAPU check;

3. HSrc succeeds in delivering a PATH SAPU to HSink; HSink succeeds in forwarding the PATH SAPU to HDest; HDest then succeeds in delivering a RESV SAPU to HSink; HSink succeeds in sending back an EVENT SAPU to HDest because of the failure of RESV SAPU check;

Figure 29 is the successful scenario described above; Figure 30 depicts the third scenario described above.

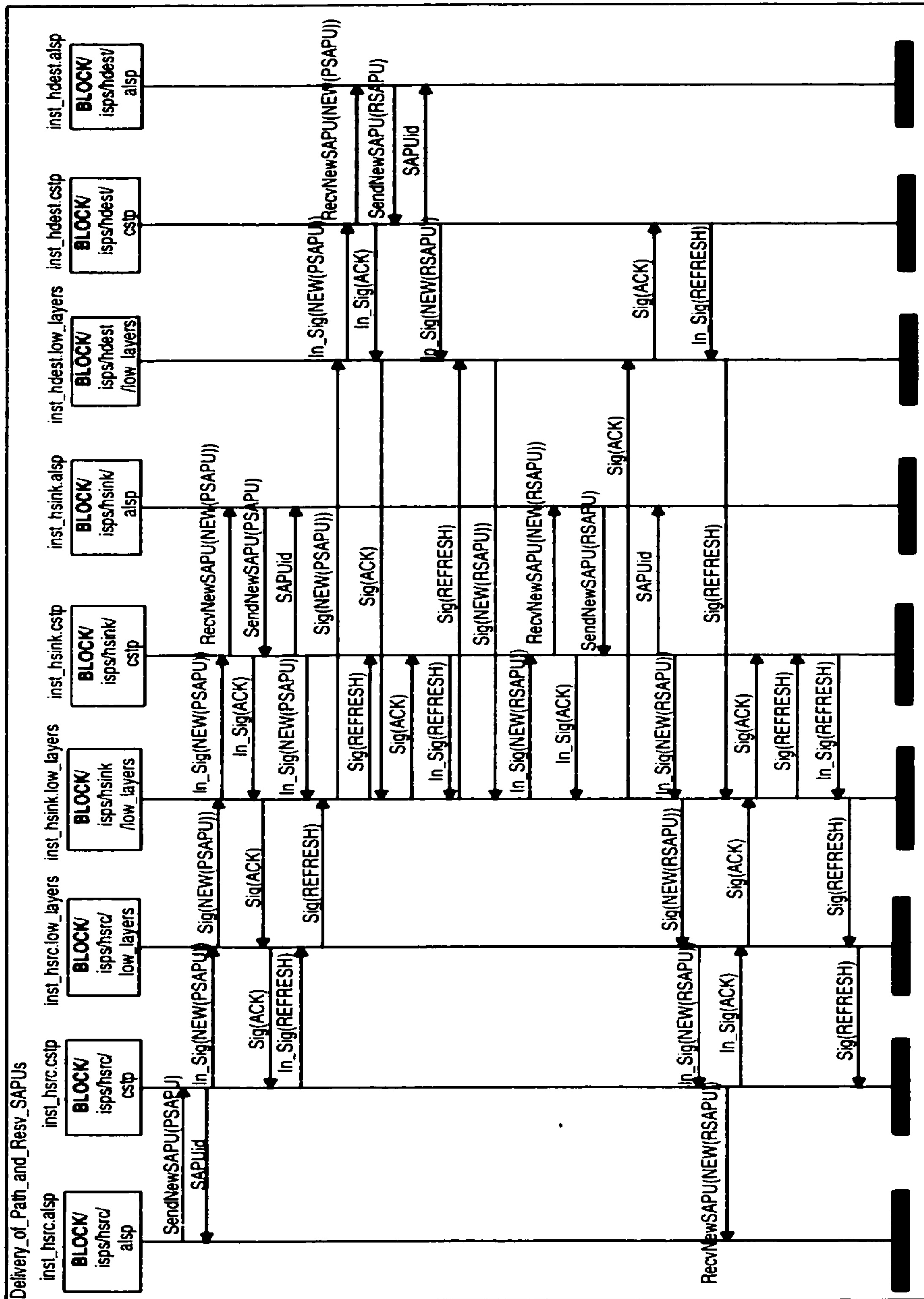


Figure 29: A Successful Scenario for Scenario *Offering Unicast Reservation Features of RSVP Version 1 by CSTP*

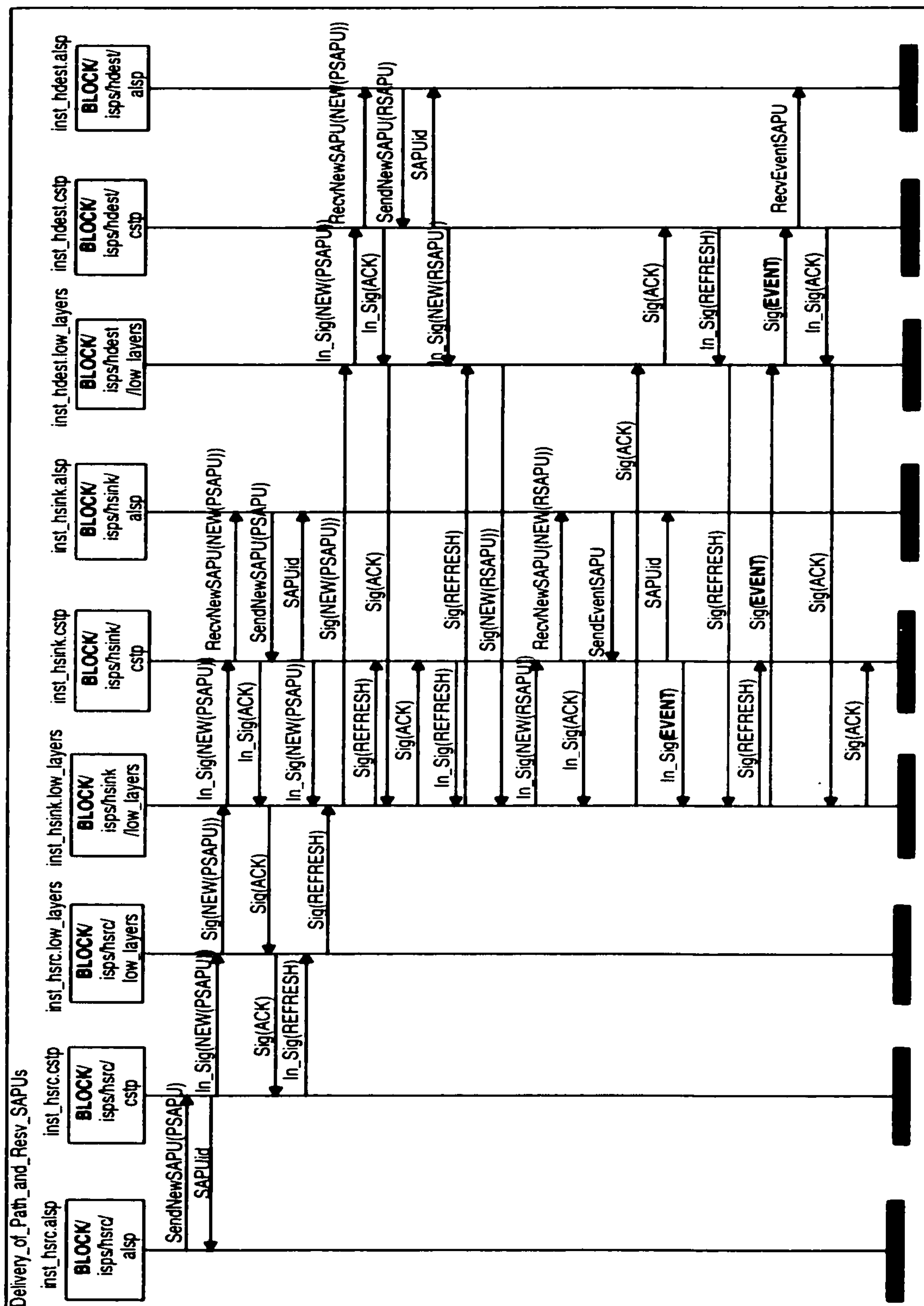


Figure 30: A Failure Scenario for Scenario *Offering Unicast Reservation Features of RSVP Version 1 by CSTP*

5.4 Validation Results

This model has passed static checks; it has no syntax error and no run time error. In addition, it has the absence of deadlock and livelock. However, we detected several unexpected signals. In addition, the design of the specification avoids many unexpected signals at an h-sink node. For example, the CSTP at an h-sink node drops incoming CSTP messages xSig(MOD) that cannot find their corresponding soft state blocks; as the result, those messages can not be treated as unexpected signals. The following subsections give the validation details.

5.4.1 Validation of Set One Scenarios of CSTP

Scenario *Send New SAPU* We succeeded in observing all the cases of the scenario, with or without fragmentation. For cases with fragmentation, it is hard for the model to succeed in reliably delivering the CSTP message xSig(NEW) at 5% packet loss rate. In order to observe the scenario under the fragmentation cases, we assigned the resending counter limit to 5. Therefore, we observed the success cases and the failure case of this scenario.

Scenario *Send Mod SAPU* We succeeded in observing all the cases of the scenario, with or without fragmentation. For cases with fragmentation, it is hard for the model to succeed in reliably delivering the CSTP message xSig(MOD) at 5% packet loss rate. In order to observe the scenario under the fragmentation cases, we assigned the resending counter limit to 5. Therefore, we observed the success cases and the failure case of this scenario. While the ALSP1 of HSrc sends two successive interface down calls SendModSAPU between 10-unit time delay, the second down call is observed as an unexpected signal at the state MOD. We will have a discussion on this unexpected signal in Subsection 6.1.3.

Scenario *Send Tear SAPU* We succeeded in observing all the cases of the scenario. When the ALSP1 of HSrc sends two successive interface down calls SendTearSAPU

between 10-unit time delay, the second down call is observed as an unexpected signal at the state TEAR. When the ALSP1 of HSrc sends two successive interface down calls SendTearSAPU and SendModSPAU between 10-unit time delay, the second call is observed as an unexpected signal at the state TEAR. We will have a discussion on these unexpected signals in Subsection 6.1.3.

Scenario *Send Event SAPU* We succeeded in observing all the cases of the scenario, with or without fragmentation. For cases with fragmentation, it is hard for the model to succeed in reliably delivering the CSTP message xSig(EVENT) at 5% packet loss rate. In order to observe the scenario under the fragmentation cases, we assigned the resending counter limit to 5. Therefore, we observed the success cases and the failure case of this scenario.

5.4.2 Validation of Set Two Scenarios of CSTP

This set only has one scenario with three cases. We succeeded in observing all the success and failure cases of this scenario.

Chapter 6

Discussion

The contents of this chapter are based on the work in the thesis.

6.1 CSTP Design Faults

6.1.1 CSTP Bundling Message Definition

The ID gives a definition for a bundled CSTP message:

```
<B-header> { <M-header-MF> <SAPU> }* <M-header> [SAPU]
```

According to the ID, "Multiple ($< M - header >$, [SAPU]) pairs, prefixed by a single $< B - header >$, can be sent in a single MTU-sized datagram; this is bundling." However, the above expression tells us only one ($< M - header >$, [SAPU]) pair rather than many in a bundling message. Without doubt, the expression is wrong.

The ID also says: "On the other hand, if the basic packet format is too large to fit into a single datagram, the ($< M - header >$, SAPU) pair can be fragmented into datagrams that are each prefixed with a $< B - header >$." That means that

one $< B - header >$ is only followed by one fragmented ($< M - header - MF >$, SAPU) pair. However, the above expression permits that one $< B - header >$ can be followed by multiple fragmented ($< M - header - MF >$, SAPU) pairs. Obviously, the expression is wrong.

A revised CSTP bundled message definition is:

```
<CSTP Bundled Message> ::=   <B-header> < Bundling datagram>
< Bundling datagram>   ::=   <M-header-MF-pair> | <M-header-pair-list>
<M-header-MF-pair>     ::=   <M-header-MF> <SAPU>
<M-header-MF>          ::=   <M-header>
<M-header-pair-list>   ::=   <M-header> [<SAPU>] | <M-header-pair-list>
```

For the undefined items see Chapter 2. When making up a $< M - header - pair - list >$, the CSTP must check its length.

6.1.2 Session Distinction

Suppose there are two CSTP sessions between a pair of CSTP neighbours an h-src node and an h-sink node, and these two sessions' messages have the same ALSPid. While CSTP messages from the h-src node arrive at the h-sink node, it is impossible for the CSTP at the h-sink node to distinguish these two sessions' messages. This is because we cannot distinguish them from the incoming SAPUs, which are opaque to the CSTP. Moreover, from the $< B - header >$ and $< M - header >$, we can get the h-src's IP address, SAPUid, ALSPid, message type, and etc., which are not enough to distiguish them. For example, at a certain time, two messages from the two sessions can arrive at the h-sink node. They both have the same h-src IP address, ALSPid, and their types are both MOD. The only difference is the SAPUid. However, SAPUid can be frequently changed during a session's life. As a result, we cannot distinguish between the two. Without this, CSTP would pass these messages into the same soft state block. This result cannot be accepted. Thus, we have to develop a way to tell the difference.

As a solution, we suggest adding a field 'session' in the $< M - header >$. Maybe this could be the easiest way. The value of the field 'session' is only valid between CSTP neighbour nodes since CSTP message transmission is a hop-by-hop behaviour.

6.1.3 The Unexpected Signal xSig(NACK)

The unexpected signal xSig(NACK) comes from an incoming CSTP message. At the state of TEAR, the finite state machine could receive an unexpected xSig(NACK). This could happen in the following situation: after the finite state machine sends an xSig(REFRESH) at the state of ESTABLISHED, it sends an xSig(Tear) immediately. Then its state transits into the state of TEAR. Due to the route change, the CSTP at the h-sink node returns an xSig(NACK). When the xSig(NACK) arrives at the h-src node, it will be treated as an unexpected signal for the state of TEAR.

As a solution, we suggest the finite state machine should discard the incoming xSig(NACK) at the state of TEAR. Because the state machine at the h-src node has decided to end the session, it will terminate when the timer expires or the state machine receives an xSig(ACK). The reception of the xSig(NACK) should not affect the decision of the finite state machine.

6.1.4 Interface Calls

Table 2 lists all CSTP messages and ALSP/CSTP interface calls. From the table we can easily find that the down calls SendEventSAPU and SendInfoSAPU need their corresponding up calls RecvEventSAPU and RecvInfoSAPU. The SendInfoSAPU also needs a corresponding CSTP message xSig(INFO) to transmit the Info SAPU.

Besides the above interface call, we also need to introduce some new interface calls. For example, SAPUid is generated at CSTP layer. The ID requires an independent up call to pass the newly generated SAPUid to the ALSP, but it does not specify this call clearly. Another interface call to be introduced is IntTearSAPU developed by ourselves. The up call IntTearSAPU is mandatory for the hop-by-hop refreshing

CSTP signaling message	**Interface Downcalls**	**Interface Upcalls**
xSig(NEW	SendNewSAPU	RecvNewSAPU
xSig(MOD)	SendModSAPU	RecvModSAPU
xSig(TEAR)	SendTearSAPU	RecvTearSAPU
xSig(REFRESH)		
xSig(ACK)		
xSig(NACK)		
xSig(EVENT)	SendEventSAPU	
xSig(CHALLENGE)		
xSig(RESPONSE)		
	SendInfoSAPU	

Table 2: The Table of CSTP Messages and ALSPs/CSTP Interface Calls

mechanism. Once a receiving state block's state life timer expires, the block must send this call to inform the ALSP.

6.1.5 Hop-by-Hop Refreshment Mechanism

The CSTP refreshment mechanism is a hop-by-hop behaviour. It works in the following way. At a h-src node, the soft state block sends xSig(REFRESH) periodically to its h-sink node. At a h-sink node, the soft state block has to stay alive by periodically receiving refresh messages from the h-src node. Once its state life timer goes off, the state will kill itself without informing the ALSP. Thus, the ALSP does not know what happens at this time and continues to maintain the soft states related to the session that the killed state belonging to. From the other point, if the h-sink node is an intermediate node, it has to maintain a sending state block belonging to the same session to refresh its h-sink node. This does not make sense, as the signaled path has been corrupted but the resources are still taken.

To resolve this problem, we revised the hop-by-hop refreshment mechanism by introducing an interface up call IntTearSAPU(SAPUid, hsrc). Once the state life timer expires, the state block at the h-sink node will issue the up call IntTearSAPU to the ALSP. When the ALSP receives this call, it will send a down call SendTearSAPU or a similar call to the sending soft state at the CSTP of the current session. Once the soft state gets the call, it kills itself. Then, the ALSP will kill the soft states at the ALSP of the session to release resources. Some further research needs to be done to find out if the sending state block at the CSTP should send an xSig(TEAR) to its h-sink node before the block is killed.

6.2 Suggestions for CSTP Design

6.2.1 Adding a SAPUid in the Signature of RecvNewSAPU and RecvModSAPU

At an h-sink node, if the receiving state block receives an xSig(NEW) or an xSig(MOD), it will retrieve the SAPU and the h-src IP address, and pass them to the ALSP, without the SAPUid in the call signature. Once the block receives an xSig(TEAR), it will retrieve the SAPUid and the h-src IP address, and pass them to the ALSP. In the ALSP, all the incoming messages have to find their soft states by matching the $< key >$. As we know, a SAPU can be treated as a $(< key >, < value >)$ pair. Therefore, all the incomining SAPUs can easily find their soft state by the $< key >$. However, an incoming SAPUid has neither a $< key >$ to find the soft state nor a lookup table to find the $< key >$. This is because when its corresponding SAPUs arrive at the ALSP, the ALSP does not have enough information to set up this table. To resolve this problem, we suggest modifying the signature of RecvNewSAPU and RecvModSAPU by adding the parameter SAPUid, hence the ALSP can set up the lookup table for the pair of $(< key >, < SAPUid >)$ or the pair of $(< SAPU >, < SAPUid >)$ before the SAPUid arrives.

6.2.2 Sending an xSig(EVENT) at an Intermediate Node

Let us suppose there are three nodes. The intermediate node has succeeded in receiving the xSig(NEW) CSTP message from the previous node. Then it sends back an ACK to the originator of the message. At the same time, it sends an up call RecvNewSAPU to the ALSP. Once the ALSP receives this up call, it will issue a down call SendNewSAPU to the CSTP. Upon the receipt of this down call, CSTP sets up a state block to reliably deliver the incoming SAPU to its h-sink node. However, due to a failure in delivering the SAPU, the state block has to send an up call SendFail to the ALSP, and then it kills itself. From the other point, the receiving soft state block at the intermediate node can be refreshed periodically by the

xSig(REFRESH) incoming from the first node, as the first node does not know what happened in the intermediate node. Moreover, the third node might never know the first node wants to signaling the third node via the second node. As a result, this sender cannot take the right actions to deal with this failure.

At this situation, we suggest that if the ALSP receives an up call SendFail, it must send a down call SendEventSAPU to the first node and let the sender decide on further actions.

6.2.3 Adding a Context for the Hsrc State Transit Diagram

The original ID does not give any application context for the Hsrc State Transit Diagram. Through the simulation, we got three unexpected signals. At the state of MOD, if the finite state machine receives a down call SendModSAPU, it will treat the call as an unexpected signal. At the state of TEAR, if the finite state machine receives a down call SendTearSAPU or SendModSAPU, it will treat it as an unexpected signal. These three unexpected signals are issued by a local ALSP module. In order to eliminate the unexpected signals, we suggest to give a reasonable context for the finite state machine. For example, we restrict the ALSP to send the down call SendTearSAPU only once. After sending a SendTearSAPU, the ALSP should not send the down call SendModSAPU or SendTearSAPU again. The unexpected signals SendModSAPU and SendTearSAPU cannot appear at the state of TEAR. However, we still need to study if we should forbid the local ALSP to successively send down calls SendModSAPU.

6.2.4 Denoting the Generation of a Modified SAPUid in SendModSAPU Clearly

Only CSTP can generate and manage SAPUids. The signature of this down call may confuse users:

```
SendModSAPU(mod-SAPUid, mod-SAPU, old-SAPUid, burst-flag)
```

Before the ALSP sends this down call, it is difficult for readers to see how and when the ALSP knows the mod-SAPUid. Actually, similar to the down call SendNewSAPU, the ALSP first sends SendModSAPU without the mod-SAPUid. Later it will receive the mod-SAPUid from the CSTP generation. We suggest the CSTP authors should denote this issue explicitly.

Chapter 7

Conclusion

7.1 Conclusion of Work

A specification of CSTP is mandatory for examining CSTP behaviour. The thesis introduced our work on the Specification and Validation of CSTP in SDL. We chose SDL as the specification language because of its dominant characteristics, such as formalization. In order to write CSTP specification, we first revised some errors and ambiguities in the ID. We wrote the CSTP specification in detail, developed soft state management and hop-by-hop refreshment mechanisms, and gave an algorithm on reassembly. To simulate the CSTP specification, we set up an abstract CSTP model, which can hold most CSTP requirements. Moreover, we designed various simulation drivers. We assumed the model works in an environment where all nodes are CSTP capable and the network produces packet losses and delay. By simulation, we verified general safety properties, and validated certain scenarios. We have found some problems, such as session setup, and unexpected signals on trigger sending states. We believe that our results will benefit the CSTP design.

7.2 Contributions

With the thesis, we have made contributions in the following ways:

1. revision of the CSTP message definition and the Hsrc State Transition Diagram;
2. specification of more CSTP state transition diagrams;
3. completion of a CSTP specification. Inside, we developed a successful CSTP internal structure and working mechanisms, such as state management mechanism;
4. validation of the CSTP specification;
5. suggestions for the CSTP future design;
6. simulation of the simplest resource reservation feature of RSVP version 1 by simulating the combination of CSTP and ALSP driver;
7. the facilitation of understanding CSTP.

7.3 Future Work

Our future works will match ISPS research progress. Obviously, it would be divided into three term goals:

The short-term goal is to complete CSTP design. It could be divided into several subtasks, such as:

- taking part in the CSTP community work to complete an enhanced CSTP model by adding some other services, such as a Congestion Manager.
- validating the new functions.

The middle term goal is to develop ASLP modules, transplant RSVP version 1 and its extensions' services into individual ALSP, hence to set up an ISPS.

The long-term goal is to map current RSVP version 1 and its extensions with the ISPS by gateways, in order to protect the current investment.

Bibliography

[1] Telelogic AB. *SDL Simulator.* ObjectGEODE 4.2 Manual, 2000.

[2] Baker, F., et. al. *Aggregation of RSVP for IPv4 and IPv6 Reservations.* RFC 3175, IETF, September 2001.

[3] Baker, F., Lindell, R., and M. Talwar. *RSVP Cryptographic Authentication.* RFC 2747, IETF, January 2000.

[4] Balakrishnan, H. and S. Seshan. *The Congestion Manager.* RFC3124, IETF, June 2001.

[5] Berger, L., et. al. *RSVP Refresh Overhead Reduction Extensions.* RFC2961, IETF, April 2001.

[6] Bernet, Y., et. al. *A Framework for Integrated Services Operation over Diffserv Networks.* RFC2998, IETF, November 2000.

[7] Braden, R. and Lindell, B. *A Two-Level Architecture for Internet Signaling.* draft-braden-2level-signal-arch-00.txt, IETF, November 2001.

[8] Braden., R. Ed., et. al. *Resource ReSerVation Protocol(RSVP) - Version 1 Functional Specification.* RFC 2205, IETF, September 1997.

[9] Braden, R., et. al. *The Design of RSVP Protocol.* Final Report for Contract DABT63-91-C-0001, 1995.

[10] Inc. Cable Television Laboratories. *PacketCable(tm) Dynamic Quality-of-Service Specification.* PKT-SP-DQOS-I01-991201, Cable Television Laboratories, Inc., 1999.

[11] Hemant Chaskar. *Requirements of a QoS Solution for Mobile IP.* draft-ietf-mobileip-qos-requirements-01.txt, IETF, August 2001.

[12] Choi, J., et. al. *Mobile IPv6 support in MPLS Network.* draft-choi-mobileip-ipv6-mpls-02.txt, IETF, 2001.

[13] CIP Working Group. *Experimental Internet Stream Protocol, Version 2 (ST-II).* RFC1190, IETF, 1990.

[14] ITU-T Rec. E.800. *Terms and Definitions Related to the Quality of Telecommunication services.* ITU-T Blue Book, 1988.

[15] SDL Forum. http://www.sdl-forum.org.

[16] Gai, Silvano, et. al. *RSVP Proxy.* draft-ietf-rsvp-proxy-02.txt, IETF, July 2001.

[17] Gene Gaines and Marco Festa. *A Survey of RSVP/QoS Implementations.* http://www.isi.edu/rsvp/DOCUMENTS, July 1998.

[18] S. Herzog. *RSVP Extensions for Policy Control.* RFC2750, IETF, 2000.

[19] G.J. Holzman. *Design and Validation of Computer Protocol.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[20] Geneva ITU. *Message Sequence Charts.* ITU Z.120 MSC, 1994.

[21] Geneva ITU. *Specification and Description Language (SDL).* ITU Z.100 SDL-92, 1994.

[22] Keaton, M., Lindell, R., Braden, R., and S. Zabele. *Active Multicast Information Dissemination.* submitted to conference, April 2001.

[23] Ooms, D., et. al. *Framework for IP Multicast in MPLS.* draft-ietf-mpls-multicast-08.txt, IETF, 2002.

[24] Ping Pan, et. al. *Fast Reroute Extensions to RSVP-TE for LSP Tunnels.* draft-ietf-mpls-rsvp-lsp-fastreroute-00.txt, IETF, 2002.

[25] Quittek,J., et. al. *Requirements for QoS Signaling Protocols.* draft-brunner-nsis-req-00.txt, IETF, November 2001.

[26] B. Rajagopalan. *LMP, LDP and RSVP Extensions for Optical UNI Signaling.* draft-bala-uni-signaling-extensions-00.txt, IETF, October 2001.

[27] Harry Rudin. *Protocol Development Success Stories: Part I.* Proceedings of the IFIP TC6/WG6.1. Twelth International Symposium on Protocol pecification, Testing, and Verification, Florid, North-Holland, Amsterdam, pp. 149-160, June 1992.

[28] Schrader,Andreas, et. al. *Requirements for QoS Signaling Protocols.* draft-brunner-nsis-req-00.txt, IETF, November 2001.

[29] Swallow, G., et. al. *RSVP-TE: Extensions to RSVP for LSP Tunnels.* draft-ietf-mpls-rsvp-lsp-tunnel-09.txt, IETF, September 2001.

[30] IETF NSIS WG. www.ietf.org/html.chapters/nsis-chapter.html.