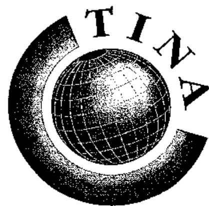


Telecommunications
Information
Networking
Architecture
Consortium

Issue Status: Public Document

Network Components Specification Version 2.2

Abstract: This document contains the computational specification of the components identified by the TINA Network Resource Architecture. The work is part of the TINA-C specification effort and as such an element in the TINA-C architecture. This document is intended to replace the 1995 Connection Management Specifications document.

Keywords: Network Component, Network Resource Architecture, Connection Management, Open Distributed Processing, Telecommunication Management Network, Managed Objects, G.803 Functional Architecture.

Author(s): Chelo Abarca, Takeo Hamada, Hyun Cheol Kim, Carlo Licciardi, Jarno Rajahalme, Raja Rosli, Frank Steegmans, Wataru Takita

Editor: Takeo Hamada, Frank Steegmans

Type: Public Document

Document Label: NCS v2.2_97_12_20

Date: Dec. 20, 1997

Table of Contents

Table of Contents . . . 3
0. Open Issues and Reader's Guide . . . 7
0.1 Objectives . . . 7
0.2 Major Structural Issues of This Document . . . 7
0.3 Major Technical Issues of This Document . . . 8
0.4 Evaluation of Technical Maturity of Sections . . . 9
0.5 Correspondence between NCS components and IDL specifications . . . 9
0.6 Compilation by hidl/hodl and HTML Files . . . 10
1. Introduction . . . 13
1.1 Objective . . . 13
1.2 Audience . . . 13
1.3 How to Read This Document . . . 13
1.3.1 Prerequisites . . . 13
1.3.2 Document Organization . . . 13
1.4 Main Inputs . . . 14
1.5 Feedback . . . 14
2. Specification Methodology . . . 15
2.1 Definition of a Network Component . . . 15
2.2 Component Specification Conventions . . . 16
2.2.1 Introduction . . . 16
2.2.2 ODL Specifications (Prescriptive) . . . 16
2.2.3 Behavior Specifications . . . 16
2.3 Naming Conventions . . . 17
3. Network Component Relationships . . . 19
4. Common Definitions . . . 21
4.1 Generic Type Definitions . . . 21
4.1.1 TINA Naming . . . 21
4.1.2 Management States . . . 21
4.1.3 Management Interface . . . 21
5. Communication Session Related Components . . . 23
5.1 Communication Session Manager (CSM) . . . 23
5.1.1 Related Information Model Fragment . . . 23
5.1.2 Behavior Specification . . . 24
5.2 Terminal Communication Session Manager (TCSM) . . . 27
5.2.1 Related Information Model Fragment . . . 28
5.2.2 Behavior Specification . . . 28
5.3 TINA Communication Session: interfaces descriptions . . . 30
5.3.1 TCSM and CSM interfaces: high level descriptions . . . 31
5.3.2 Components and interfaces . . . 34
6. Connectivity Session Related Components . . . 45
6.1 Overview of Connectivity Session . . . 45
6.1.1 Relationship to ConS -RP . . . 47
6.1.2 Connectivity Session (ConnS) . . . 47
6.1.3 Network Flow Connection (NFC) . . . 47
6.1.4 Network Flow End Point (NFEP) . . . 48

6.1.5 Network Flow Connection Branch 48
6.1.6 Related Information Model Fragment 49
6.2 Connection Coordinator Factory (CCF) 51
6.2.1 Behavior Specification 51
6.3 Flow Connection Controller 54
6.4 TINA Connectivity Session: interfaces description 55
6.4.1 CCF, CC and FCC interfaces: high level descriptions 55
7. Layer Network Related Components 71
7.1 Layer Network Coordinator (LNC) 73
7.1.1 Interface Description 74
7.2 Trail Manager (TM) 75
7.2.1 Interface Description 76
7.3 Terminal Layer Adapter (TLA) 76
7.3.1 Interface Description 77
7.4 Tandem Connection Manger (TCM) 81
7.4.1 Interface Description 82
7.5 Layer Network Binding Manager (LNBM) 83
8. Subnetwork Related Components 85
8.1 Connection Performer 85
8.2 Related Information Model Fragment 86
8.3 Mapping of Information Object to CP 88
8.4 Computational structure 88
8.4.1 General access interface to NRIM objects 88
8.4.2 Connection Performer Interface--Subnetwork Connection Management. 90
8.4.3 Example scenarios 91
9. Accounting Management Components 93
9.1 Overview of Accounting Management in TINA Service 93
9.1.1 Visibility of Billing Context 95
9.2 An Example Scenario of Accounting Management 96
9.3 Accounting Event Management 98
9.4 Essential Accounting Events in Network Resources 99
9.5 Non-essential Accounting Events in Network Resources 100
9.6 Generic Accounting Management Components 102
9.6.1 AmcLadder 102
9.6.2 i_AmcLadderElement 104
9.6.3 Accountable Object 104
9.6.4 Usage Metering Log Manager (UMLog) 104
9.7 Management Domain Related Components 105
9.7.1 Accounting Policy Manager 105
9.8 Relationship to Other Documents 105
10. Fault Management Components 107
10.1 Introduction 107
10.2 Computational Viewpoint 107
10.3 Functions 108
10.3.1 FM functions 108
10.3.2 COs functions 108
10.3.3 Future extensions 110
11. Document History 111

12. Acronyms 115
13. Glossary 119
14. Annex: ODL-specs 125
14.1 ConnectionCoordinator.odl 125
14.2 ConnectionCoordinatorFactory.odl 125
14.3 ConsUserAgent.odl. 126
14.4 ContractProfileManager.odl 127
14.5 FlowConnectionController.odl 127
14.6 InitialAgent.odl 128
15. Annex: IDL-specs 131
15.1 CLNCommonDefs.idl 131
15.2 ComSCommonDefs.idl 133
15.3 Common.idl. 134
15.4 ConnectionPerf.idl 135
15.5 NRACommonDefs.idl. 140
15.6 NrimObjectConf.idl 141
15.7 NrimRelConf.idl. 147
15.8 PLATyToolsFix.idl 156
15.9 RACommon.idl 156
15.10 Security.idl 162
15.11 States.idl 162
15.12 UMLogManager.idl 165
15.13 accPolicyManager.idl 167
15.14 attribute.idl 168
15.15 capability.idl 168
15.16 csm.idl. 171
15.17 exceptions.idl 175
15.18 lnc.idl 176
15.19 lncfed.idl. 183
15.20 media.idl. 187
15.21 naming.idl 188
15.22 nfep.idl 189
15.23 sfep.idl. 191
15.24 sfepcoms.idl. 191
15.25 tcsm.idl 192
15.26 tla.idl. 197

0. Open Issues and Reader's Guide

0.1 Objectives

First of all, this document is INCOMPLETE. Although the network component specification (NCS) was planned to be made a baseline of TINA documents, some interruptions both in terms of work schedules and human resources of the core-team prevented it from making the final goal. The current document is delivered "as is", i. e. this document is not reviewed by the stream or the core-team, or even adequately edited. It is more of less a compilation of final drafts of the last generation of core-teamers, with marginal editing efforts.

The sole purpose of this version is to provide a starting point for the continuing NCS work, which is planned to be done by a yet-to-be-identified working group. This work is supposed to have critical importance of the TINA resource architecture work, which is the culmination of 5 years' resource architecture works from NRIM to NRA, done in the core-team.

In this section, we present our analysis of the current status. Then we list open issues, works to be done to complete the document.

0.2 Major Structural Issues of This Document

Overall, we think that major topics of NRA has been covered in NCS. Though some elements (e. g. configuration, fault management) are still missing. Major structural changes are still necessary, in particular

- *Readability of IDL/ODL Files:* readability of IDL/ODL files are still low. Some files rely on modules, whereas other files rely on inheritance mechanism of nested interfaces. In short, there is no consistent style in IDL files. There is no naming convention consistently followed among current set of IDL files. Even though the introduction part of this document states as if there is a naming convention in this NCS document, too many of the files (in fact most of them) do not follow it. A major restructuring of IDL files with a consistent naming + modular scheme is necessary.
- *ODL Specification:* even when IDL specifications are done fairly well, ODL specifications are still very immature. The current ODL specifications (idl/objects) only covers a small portion of the objects provided in NCS, and the objects provide only small portion of interfaces which they are expected to provide.
- *IDL Specification:* some sections of this document cite IDL files. They are, however, not necessarily most up-to-date. The IDL specification in this document may differ from those in idl/modules. If they differ, those in original IDL files (idl/modules) should have precedence over those in this document.
- *Readability of Document:* this document can be made readable, only when IDL/ODL themselves are readable, since this document is intended to explain the IDLs. By the reasons stated above, this document itself needs a major re-organization, to better reflect the structure of the specifications. The structure of this document itself assimilates that of NRA v3.0, which is still missing some elements to be a complete architecture document (e. g. performance monitoring).

- *Separation of ConS:* ConS reference point is described as a part of this NCS document (IDL file is under idl/ConS directory). Although IDL files is separated from other NCS specifications, the contents of the document has not been updated to reflect the change. Old IDL files for connectivity session components (cc.idl, fcc.idl, ccf.idl in ver. 2.1) are now replaced by ConS.idl. There are, however, interfaces of the relevant components (CC, FCC, CCF) not exactly in ConS (e. g. i_Notify). Those interfaces will be subsequently added to respective IDL/ODL files of respective components in the future update.

0.3 Major Technical Issues of This Document

60% to 80% of the interfaces and components of TINA connection management are already covered in this document. It means that most of the topics covered in NRIM and much of connection management part of NRA are covered by this document, though they have never been officially reviewed. Some recent advances since last release of NRA (v3.0, Feb. 10, 1997) have been added, or planned to be added but not realized.

- *Internet Transport:* good basic concepts of connectionless network and use of internet transport in TINA network resource layer have been established. The results are published in TINA'97 (Santiago de Chile, Nov. 1997) by resource streamers ("Managing TINA Streams that use Internet Transport", Hyun Cheol Kim et al.). The current version of NCS have already taken the proposal into its specification (Sec. 7 Layer Network Related Components). A new computational component called Layer Network Binding Manager (LNBM) has been added, which manages technology-specific, connectionless network.
- *QoS negotiation:* ideas have been proposed in TINA'97 (Santiago de Chile, Nov. 1997) by resource streamers ("Quality of Service Negotiation in TINA" by Jarno Rajahalme et al.). The paper mainly discusses negotiation of QoS parameters at terminal level, and their mapping onto transport layer. The proposal is not taken into the current NCS specification, and is yet to be reviewed.
- *Configuration Management:* a section is devoted for configuration management in NRA. Though the topic is important, the whole material is missing in this version of NCS.
- *Fault Management:* a section is devoted for fault management in NRA. In this NCS document, a section (Section 10) is spent for fault management. Some basic concepts are presented, but the concepts are still immature and many parts are still missing to be useful.
- *Security Issues:* security issues are only awkwardly handled in this document, and inconsistent throughout this document. All interfaces exposed at reference points (ConS, TCon) need at least have basic security measures (authentication, authorization) based on DPE security (i. e. CORBA security++). In this NCS document, some interfaces in connectivity session components (CC, CCF, FCC) require a set of security parameters represented by 't_SecHandle'. Though its idea is correct, its semantics is yet to be clarified, and it is also likely that the parameter is not necessary in regular operational environment using CORBA security (corresponding parameter is passed as part of DPE security operations, thus they do not appear at connectivity components API).

0.4 Evaluation of Technical Maturity of Sections

In this section, we evaluate the current status of each section. These evaluation can not

Table 0-1. Evaluation of Technical Maturity of Sections

	Basic Concept	IDL/ODL Specification
2. Specification Methodology	7	N.A.
3. Network Component Relationships	5	N.A.
4. Common Definitions	3	3
5. Communication Session Related Components	7	4
6. Connectivity Session Related Components	6	4
7. Layer Network Related Components	7	5
8. Subnetwork Related Components	7	5
9. Accounting Management Components	7	6
10. Fault Management Components	2	0

help being subjective, but we feel they are fairly accurate estimation. The point represents the maturity level of work item represented by respective section, from 0 (no work done) to 10 (complete). Each point can roughly translate into a week's work of an expert in the appropriate area. For example, section 6 (Connectivity Session Related Components) may require four more weeks to complete basic concepts (texts of this document) and 6 more weeks to finish all relevant IDL/ODL files. There are other work items not listed in the current NCS document, though, our expectation is that 6 mo. or 1 year of intensive work is necessary to finish all the work items to a satisfactory level.

0.5 Correspondence between NCS components and IDL specifications

In the following, we show the correspondence between NCS components and IDL specifications (idl/modules). Although it should be shown throughout this document and ODL specifications, which specifies relationships between computational object and interfaces, both are immature at this point, the correspondence between NCS components and IDL specification are not clearly shown.

Table 0-2. Correspondence between NCS components and IDL specifications

Section	Component Names	IDL File Names
2. Specification Methodology	None.	None.
3. Network Component Relationships	None.	None.

Table 0-2. Correspondence between NCS components and IDL specifications

Section	Component Names	IDL File Names
4. Common Definitions	None.	Common.idl NRACommon.idl RACommon.idl States.idl attribute.idl exceptions.idl naming.idl
5. Communication Session Related Components	CSM TCSM	CLNCommonDefs.idl capability.idl csm.id media.idl sfep.idl sfepcoms.idl tcsm.idl
6. Connectivity Session Related Components	CCF CC FCC	ConS.idl nfep.idl
7. Layer Network Related Components	LNC LNBM TM TCM TLA	lnc.idl lncfed.idl tla.idl
8. Subnetwork Related Components	CP	ConnectionPerf.idl NrimObjectConf.idl NrimRelConf.idl
9. Accounting Management Components	AmcLadderElement	UMLogManager.idl accPolicyManager.idl
10. Fault Management Components	AM FC TDS	None.
Miscellaneous	None (for debugging).	PLATyToolsFix.idl

0.6 Compilation by hidl/hodl and HTML Files

All the IDL/ODL files are successfully compiled using hidl compiler from OmniBroker and hodl compiler by TINA-C, respectively. To apply hidl compiler, an aggregated IDL file is prepared (idl/ncs.modules.idl). All the browsable HTML files are under idl/html directory.

- _top_.html: the result of hidl compiler, which translated the aggregated top-level IDL file (ncs.modules.idl). The HTML file contains index of modules, interfaces, and so on. Its URL is:
  http://tinac.com:4070/97/resources/network/docs/ncs/v2.2/idl/html/_top_.html

- _top_o_.html: the result of hodl compiler. The HTML file contains index of objects. Object HTML files are generated along with it, which shows its behavior, and it points to HTML files of supported interfaces. Its URL is:
  http://tinac.com:4070/97/resources/network/docs/ncs/v2.2/idl/html/_top_o_.html

1. Introduction

1.1 Objective

The objective of this document is to specify the network components as described by the Network Resource Architecture (NRA) [4]. Components are specified using TINA ODL [8], and the behavior of the components is specified with state diagrams and event traces.

This document is based on other TINA baseline documents. However, new considerations regarding Network Resource Architecture or Network Resource Information Model may be included. These will be introduced in upcoming releases of the other TINA documents.

This work is part of the ongoing Network Resource Architecture activity within the TINA-C Core Team and aims at its part to harmonize the work done on the NRA and the relevant TINA reference points effort.

1.2 Audience

The intended audience is the whole TINA community at large, but especially parties interested in the Network Resource Architecture in:

- The TINA Core-Team
- TINA auxiliary projects
- The TINA Trial projects

1.3 How to Read This Document

This document is part of a set of documents issued by the Core Team of the TINA Consortium. In particular, this document further explains the interfaces and behavior of components defined in the Network Resource Architecture.

1.3.1 Prerequisites

This document assumes the reader is familiar with at least the following TINA documents:

- *Network Resource Architecture* [4]
- *Network Resource Information Model Specification* [5]
- *Computational Modeling Concepts* [1]
- *TINA Object Definition Language Manual* [8]

1.3.2 Document Organization

A brief description of the contents of each major section in this deliverable follows.

- Section 2 defines the concept of network component and describes the use of TINA ODL and other methodological considerations applied in the development of the specifications. The structure and requirements for the specifications in the following sections is also given.

- Section 3 provides an overall presentation of the components of the network resource architecture, including description of component relationships.
- Section 4 deals with common definitions used in NCS, such management states and attributes.
- Section 5 deals with communication session related components such as CSM and TCSM.
- Section 6 covers connectivity session related components such as CCF, CC, and FCC.
- Section 7 covers layer network related components such as LNC, TM, TLA, TCM, and LNBM.
- Section 8 covers subnetwork related components such as connection performer (CP) and element management objects (NRIM objects).
- Section 9 covers accounting management components.
- Section 10 covers fault management components.
- Section 11 presents the history of this document.

1.4 Main Inputs

The main inputs to this work are:

- *Connection Management Specifications* (1995) [2].
- *Network Resource Architecture* version 3.0 [4].
- *Network Resource Information Model Specification* [5].
- Reference Points [6].
- *Connection Management Specifications for VITAL 2nd phase* [11].

1.5 Feedback

Please send your comments via e-mail to: ncs@tinac.com. You can also access the web page http://tinac.com:4070/97/resources/www/ncs.html for more information on the NCS review process etc.

2. Specification Methodology

This section describes the specification methods used in this document. A definition for a network component is given first, followed by the structure for individual component specifications. TINA naming conventions are also summarized to facilitate consistent naming throughout the document.

2.1 Definition of a Network Component

The current NRA [4] does not define the concept of the *network component*. For the purpose of this specification, we define a network component as follows:

> *A (TINA) component is a computational object[1] whose internal decomposition or distribution pattern is not visible[2]. The component is characterized by its supported and required interfaces and the semantics of those interfaces. The interfaces are bound to the defined inter- and intra-domain reference points[3] as well as to TINA defined architectural separations[4].*
>
> *A network component is defined as a component within the scope of the Network Resource Architecture.*

Components are building blocks for the architecture, providing segmentation of the functionality of TINA systems. Component specifications provide a high level view of the segmented functionality, focusing only to the external interfaces and behavior. This allows the components to be provided by different vendors and still interoperate. Component specifications may also suggest some internal decomposition of the component, but those parts of the specifications are only descriptive.

Since components are defined as objects, they have all the characteristics of objects, especially:

1. Components collaborate through the defined interfaces.
2. Encapsulation: Neither internal state nor internal behaviour is visible to outside of the component.
3. Specialization: Components can be refined through inheritance to allow vendor-specific added-value functionality, different distribution patterns, etc.
4. The initial interface of the component provides the contact point and component management functions.

---

1. Note that this definition requires relaxation of the definition of the computational object (CO) in the current Computational Modeling Concepts [1], namely to allow a CO to composed of component COs. See also Ch. 7.1 in [9] for related discussion.
2. Each instance of an interface will be bound to one DPE node, though.
3. See [7] for definition of TINA reference points.
4. E.g. separation between access, primary and ancillary service usage, and communication [9].

2.2 Component Specification Conventions

This section specifies the structure in which the individual component specifications are given in the following sections.

Each component specification consists of following three parts:

1. Introduction
2. ODL specification
3. Behaviour specification

The content of each of these is described in the following sections.

2.2.1 Introduction

In this section the component is shortly introduced with a computational model fragment showing the component in relation to other components. An internal subdivision can also be suggested.

Since each object deals with a fragment of the information model, relevant mappings from information to computational models are shortly presented, referring to [3] when necessary.

Each component has an initial interface. This interface serves as the entry point to the component. Other interface references may be given through the initial interface or as return values from other interfaces. These include the common management interfaces for the components.

2.2.2 ODL Specifications (Prescriptive)

ODL specifications are used to specify the interfaces and related data types of the component object. Common interfaces and data types are defined separately from the component specifications.

OMG IDL is used to prescribe the interfaces and data types. TINA ODL is used to specify the computational objects making use of the interface and data types. The file extension **.idl** is used for OMG IDL files, and **.odl** is used for ODL files. This is to make use of OMG CORBA tools easier.

2.2.3 Behavior Specifications

Behavior is specified at three levels:

1. At the component level: Generic description; how required and supported interfaces are interrelated, how the component in question relates to other components.
2. At the interface level: General purpose and operations; how the operations in this interface relate to each other.
3. At the operation level: Pre-conditions, sequence of actions and post-conditions. In pre- and post-conditions, the state attribute values of all the objects involved are indi-

cated.

2.2.3.1 State Diagrams (Prescriptive)

State diagrams are used to specify the semantic relationships between interface operations. Each operation invocation typically causes a state change after which a given set of operations can be invoked. State diagrams provide precise means for this kind of specification without long natural language descriptions.

State diagrams belong to the prescriptive part of the architecture.

2.2.3.2 Event Traces (Descriptive)

Event traces are examples of possible operation sequences in graphical format. An event trace can never completely specify component behavior, but can be used to describe e.g. the most common use of the component. Event traces are valuable examples for understanding component usage, but provide little support for the component implementation. Therefore event traces are descriptive.

2.3 Naming Conventions

General conventions used in IDL and ODL specifications:

- Object names appear in the format of 'XxxXxxXxx'.
  - E.g. InitialAgent, UserAgent.
- Interface type names appear in the format of 'i_XxxXxx'
  - E.g. i_ProviderInitial
- Operation names appear in the format of 'xxxXxxXxx', and the first word (xxx) is typically a verb.
  - E.g. requestAccess(...) is a method on the i_ProviderInitial interface of the InitialAgent object
  - It may also be written as InitialAgent::i_ProviderInitial::requestAccess(...).
- Data type names appear in the format of 't_XxxXxxXxx'.
- Constant names appear in the format of 'XxxXxxXxx'.
- Exception names appear in the format of 'e_XxxXxxXxx'.
- Parameter names appear in the format of 'xxxXxxXxx'.
- Comments are started with '//' and continue to the end of the line.
- Comments spanning multiple lines may use '/* ... */'.

3. Network Component Relationships

The definition of TINA business roles, separation principles and conceptual layering dictate the outer limits for the network components. While it would be tempting to go further in the functionality subdivision, such subdivisions can not be conformance requirements. Therefore, we define the network components using the most outer limits. This allows maximum freedom for vendors to design the actual distribution patterns and functional subdivisions of the components, while still maintaining the interoperability provided by the reference points. Note, however, that a specification of an individual component is free to *suggest* further subdivisions, especially if they make the specification more readable.

The reference points [7] divide the Network Resource Architecture functionality into a set of categories. Figure 3-1 summarizes the reference points and the components they bind. The lines in the figure represent the different reference points, individual computational interfaces are not shown. The components are:

- Communication session related components (Section 5):
  - Communication Session Manager (CSM)
  - Terminal Communication Session Manager (TCSM)
- Connectivity Session related components (Section 6):
  - Connection Coordinator (CC)
- Layer network related components (Section 7):
  - Layer Network Coordinator (LNC)
  - Trail Manager (TM)
  - Layer Network Binding Manager (LNBM)
  - Tandem Connection Manager (TCM)
  - Terminal Layer Adaptor (TLA)
- Subnetwork related components (Section 8):
  - Network Management Layer Connection Performer (NML-CP)
  - Element Management Layer Connection Performer (EML-CP)

Note that reference points below Layer Network (LNW)-RP are technology dependent, i.e. they need to be specialized for different network technologies.

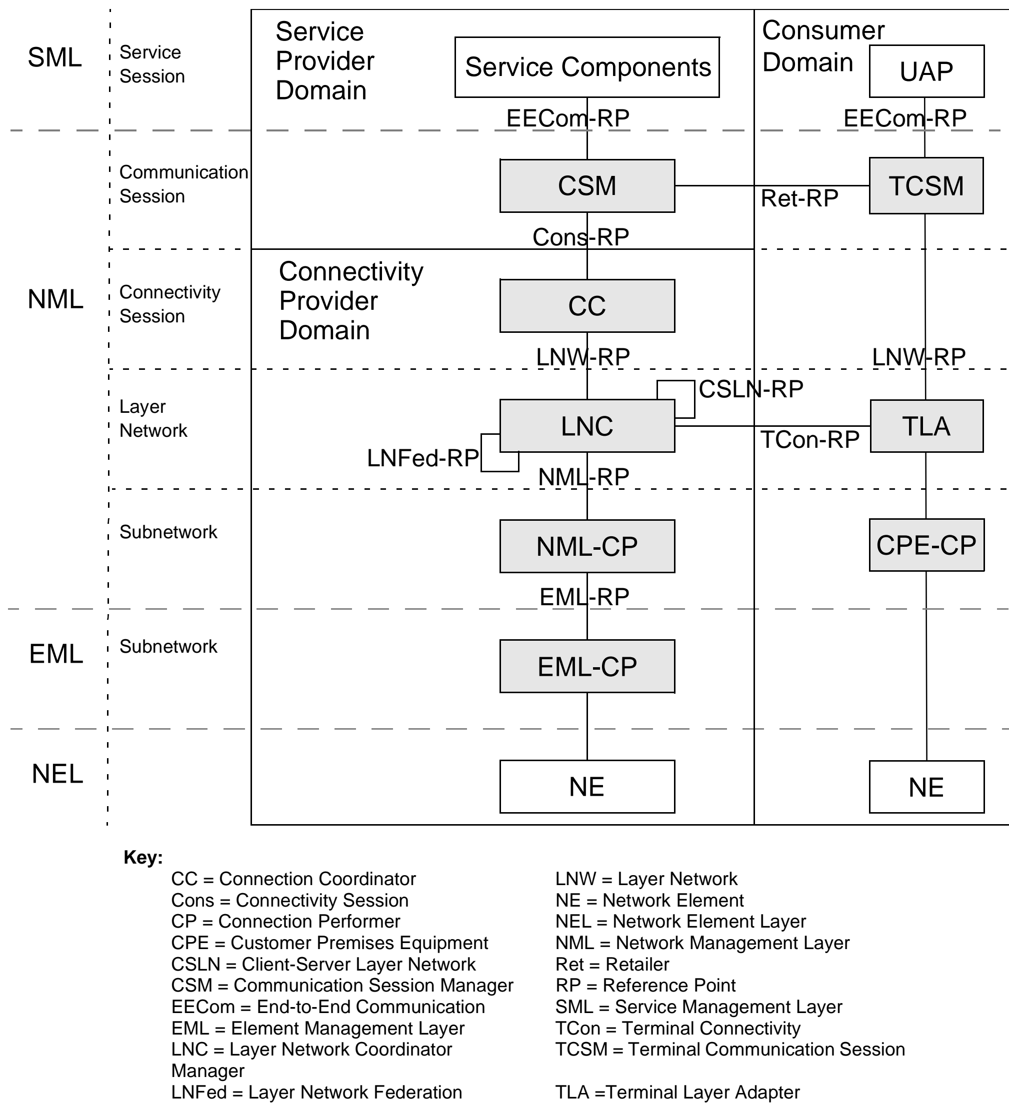


Figure 3-1. Overview of the Network Components and Reference Points

4. Common Definitions

4.1 Generic Type Definitions

4.1.1 TINA Naming

TINA naming is explained in naming.idl (Section 15.21).

4.1.2 Management States

Management states are explained in States.idl (Section 15.11)

4.1.3 Management Interface

Management Interface allows operations on information object attributes as defined in NRIM. In addition to the Get/Replace operations, object may support configuration management (Create/Delete operations).

One important portion of management interface is state management. This interface will be used to control the states of the information model objects supported by components. States of all the information objects are managed through this interface.

Usage of state management interface is normally not needed by the actual service use, since setup operations allow specification for the initial states.

Information objects are addressed by names, so it is possible to implement one of the interfaces for the whole component. Tina naming conventions are defined, and the component uses them to consistently name all the created information objects.

State management interface has operations to both query and set the state of an object, or group of objects. All operations are subject to policies and access rights.

5. Communication Session Related Components

5.1 Communication Session Manager (CSM)

The CSM computational object offers to service level (i.e. SSM) a logical view of connectivity. Connectivity resources at this level are information streams. Information streams flow from an endpoint to another endpoint. These endpoints are modelled as Stream Flow Endpoints (SFEPs) and information streams as Stream Flow Connections (SFCs). So, connectivity at this level must be requested in terms of SFCs. This means that a client of CSM can setup, modify and release a Communication Session that is made up of one or more stream flow connections.

Figure 5-1. CSM CO

5.1.1 Related Information Model Fragment

Figure 5-2. OMT diagram for EECom-RP

5.1.1.1 Communication Session (CommS)

An instance of this object type contains one or more stream flow connections. It represents the total set of communication resources involved in a communication session. A client of a communication session can establish, release and manipulate stream flow connections.

Attributes:

- Communication Session Name
- Success Criterion (AllorNone, BestEffort)
- Stream Flow Connection Description List

5.1.1.2 Stream Flow Connection (SFC)

An instance of this object represents a stream flow connection. A stream flow connection transports information from a source stream flow endpoint to one or more sink stream flow endpoints. Stream flow connections are always unidirectional. A stream flow connection can be established, released, activated and deactivated. New branches can be added or deleted to/from a stream flow connection. A stream flow connection is always either point-to-point or point-to-multipoint unidirectional (when non specified it is assumed to be point-to-multipoint by default).

Attributes:

- Stream Flow Connection Name
- Success Criterion (AllorNone, BestEffort)
- Stream Flow Connection Topology {PointToPoint, PointToMultipoint}

- Stream Flow End Point Description List

5.1.1.3 Stream Flow End Point (SFEP)

This class represents an endpoint of an information flow. Stream flow end points are always unidirectional; a SFEP is either a source or a sink. SFEPs already exist before they can be used in a SFC; they are considered to be created by the applications or by a SFEP Manager (an entity that would be able to keep track on the resources within the terminal) and registered within the terminals. The administrative state can be specified as locked or unlocked. Unlocked means that the requested connection should be immediately ready for receiving information. Locked status means that the needed resources have just been allocated, they can not be activated. If no status is specified unlocked is assumed. High level QoS parameters should be specified depending on the type of the flow (not considered in VitalV2).

Attributes:

- Stream Flow Endpoint Name
- Stream Flow EndPoint Direction (Source or Sink)
- Administrative State (Locked or Unlocked)
- Media Description {High level requested QoS parameters}

5.1.1.4 Stream Flow Connection Branch

An instance of this association represents a branch of a stream flow connection. It relates a stream flow connection object and a sink stream flow endpoint.

5.1.2 Behavior Specification

The CSM computational object offers two interface types to the SSM. One that enables the creation (or release) of communication sessions and another one that enables the creation (activation, deactivation and release) of stream flow connections within each one of the created communication sessions.

Assumptions:

- The mapping between Stream Binding and communication session is one-to-one.
- The logical view of connectivity given by the stream binding mechanism only supports point-to-point and point-to-multipoint unidirectional connections since SFCs are always unidirectional.
- The idl specification includes only what Vital has considered as mandatory at this interface.
- Only simple exception handling has been considered.
- The QoS at the CSM level will take values that are independent from any specific technology. For Vital V2 it has been assumed that simple values are passed to the CSM (strings). A more complex structure is foreseen for Vital V3.

- The Retailer and the Connectivity Provider will be within the same administrative domain (typically an operator). This means that no parameters have been included to allow the CSM to choose the Connectivity Provider where the connectivity session is instantiated. In the future the NP might be deliberately indicated by the user or it can be chosen based on the Network Address given by the TCSM.

In order to illustrate the behaviour of the CSM some event sequences are shown in the following more representative scenarios:

- Communication Session setup
- (more will be added in the final version)

5.1.2.1 Communication Session Setup

Figure 5-3. Communication Session Setup

1. SSM requests the CSM for creation of a communication session. This request should include as parameters:
    - The Success Criterion that indicates the criterion for a successful completion of the session set operation. The possible values are AllorNone or BestEffort. In the first case the operation will be considered successful if all the SFCs are setup successfully while in the second case the operation will succeed if at least one SFC is successfully setup.
    - A list containing the SFCs that belong to the requested communication session. A least one SFC should be specified. In order to describe properly a SFC a Name should be given by the SSM to each of them, a particular Success Criterion (indicating a successful SFC if all the sinks are successfully bound or at least one), the topology, and the SFEPs. For each SFEP the reference should be indicated, the type (source or sink), the media description (i.e. the QoS) and the administrative state; locked means that the information can not flow and unlocked means that it can. The locked status can be modified to activated by requesting to the CSM an activation operation.
2. The reference of the ComSCtrl Interface will be sent back to the SSM together with the SFC response parameters (i.e. SFC Name and respective SFepRefList of the bound SFEPs).

There is only one ComSSetup interface per CSM. It is the factory of instances of ComSCtrl interfaces. The CSM will instantiate one ComSCtrl Interface per communication session. This interface allows the SSM to control the Session by requesting new stream flow connections, deleting existing ones, activate and deactivate them. Additionally new branches can be added (or deleted) to the SFCs and activated (or deactivated).

The CSM will issue a similar request to the CC; a request for a connectivity session setup. But in order to do that CSM has to perform some actions to meet the following objectives:

1. Find the TCSMs.

2. The parties must agree on the capability sets (i.e. CODECs) they will use at the application level (In Vital only one option will be considered, not list of options).
3. From each SFepRef CSM has to find out the correspondent Rfep (in VitalV2 it is characterised by only one Network Address, one LNW type and one TLA Reference).
4. The parties must agree on the network protocol they will use at network transport level. (In VitalV2 only one option will be considered).
5. Generate a CorrelationId per SFEP. This parameter will be used by the TLA to resolve the RFEP (or NFEP) within the terminal once the NWTPs have been decided by the CPs (e.g. VCI number 10).

Figure 5-4. CSM-TCSM Relationship

- This information is negotiated between the CSM and the TCSM. The TCSMRef encapsulated in each SFEP will be used to locate the TCSM in each terminal.
- CSM performs the mapping between SFEPs and NFEPs (or REFPs). By default the mapping has been assumed to be one-to-one; the same is valid for the SFC-NFC mapping.
- A unique correlation identifier is created per SFEP. This value will be used by the terminal to resolve the NFEP when e.g. the VPI/VCI values are chosen by the lower levels.
- Then the CSM object will send a request to the TCSM (correlate operation) in order to find out the characterization of the correspondent RFEP. As parameters the CorrelationId and the SFEPId are included. In return the RFEP information is sent, i.e the LNWType, the TLARef, the correspondent network address (in Vital V2 it has just been considered one network address per LNW / TLA) and the protocol information at the application level (Codecs) and at the transport network level (For Vital V2 only one combination as been considered). In Vital V3 the connectivity provider information and the LNW type might be used by the CSM to choose the overall connectivity provider. In Vital V2 the CSM will just follow this information to the CC since the Retailer and the Connectivity Provider business roles are within the same administrative domain, i.e. each Network Operator.
- Then the CSM will send a request to the TCSM (i.e. modifyProtocol operation) in order to negotiate the protocols at both the application level and the transport network level (Vital V2 offers the possibility of negotiating both in one operation or separately in two independent operations). Note that in this scenario it is assumed that one operation is sent per SFEP but the CSM object can deal with several SFCs at the same time. At this phase it is assumed that if the protocol information received form all the SFEPs is the same the CSM will proceed, if not it will impose the protocol information received by the SFEP source to the SFEPs sinks. Note that the same operation can be used to impose the source protocol to the sinks since “protocol” has been included as an inout parameter.
- The CSM can now request the already identified NFCs to the CC (see section 5).

5.2 Terminal Communication Session Manager (TCSM)

The Terminal Communication Session Manager (TCSM) is responsible for establishing the nodal (or TFC) of a SFC. It manages the connection setup for a node and does not participate in actual connections. It cooperates with CSM responsible for the SFC.

TCSM in VITALv2 knows the details of the nodal binding in a technology independent manner, which are hidden from CSM. For this reason it interacts with user applications in order to resolve the TFC to media flow connection between the terminal devices and the software modules that manipulate the information that flows between them.

There is one TCSM per terminal, special resource, or any node that terminates a stream flow connection. A TCSM can manage multiple TFCs.

At last, TCSM acts as SFEP manager. It is responsible for the creation/deletion of a SFEP and manages its life-cycle. TCSM assigns (raises) SFEP to (from) user applications, under the request of the last ones, as well as it associates (deassociates) a SFEP to a TFC.

Before requesting for a streambinding, a user application has to request the TCSM to assign to the application the needed SFEPs. The SFEPs are identified uniquely by a SFEP identifier (in VITAL v2 is of type t_SFEP_CId). Later, when the CSM establishes a SFC, it needs a way of associating the network part of the connection with its terminal(s) part(s). To do this, it generates a unique identifier, called correlation identifier, and passes this identifier to each TCSM associated with the SFC, plus the SFEP identifier, that will be associated to the TFC as root FEP. This identifier identifies both the associated SFC and NFC. This correlation identifier is also passed from CSM to the connectivity session and from connectivity session to a trail in a layer network. Once a unique NFEP is either created or selected, during the establishment of a trail, the correlation identifier is passed to the corresponding TLAs. Then, each of the TLAs inform its corresponding TCSM of the chosen NFEP, using the correlation identifier, in order to identify the specific TFC to which that NFEP will be assigned as leaf FEP.

Figure 5-5. TCSM CO

TCSM provides to its clients the following interfaces:

- TCSM provides to UAP-SS a I_tcsmSFEPManager interface for the manipulation of the SFEPs.
- It provides the I_rettcsmCoord interface to CSM in order to be the CSM able to resolve the LCG to its nodal part and to establish a NCG at each terminal that is involved in a communication session. This interface is part of the Ret-RP
- It provides the I_tcsmReport interface to TLA in order to be the TCSM able to complete a NCG.
- It provides the I_tcsmNotification interface to UAP-CS and TLA in order to be the TCSM notified by application-driven and network-driven changes to the state of the NCG.

- It also provides an I_rettcsmNotifyCtrl interface to CSM, in order to get to CSM the chance to cotrol the emissions of notifications from TCSM to itself.

5.2.1 Related Information Model Fragment

The TCSM as has been already noted, maintains the nodal part of the communication sessions that affect the terminal that belongs. This is realised in terms of Nodal Connection Graph (NCG).

The Nodal (or Terminal) Connection Graph (NCG) concept is used to specify the terminal connectivity needs of a communication session in a technology independent manner.

The connectivity requirements specified at the node are almost similar to the Logical Connection Graph (LCG), but there are some differences with LCG as in the node it should differentiate what binding are resolved locally in the node and which have to be associated with a remote node.

The NCG contains a number of Terminal Flow Connections (TFC) that interconnect Stream Flow End Points (SFEPs) and Network Flow End Points (NFEPs). In the current solution these TFCs only support a point-to-point connection, either between two SFEPs or between a SFEP and a NFEP.

In order to resolve a TFC it is necessary to take into account engineering view concerns and include terminal specific information on terminal devices and operating systems.

Each Terminal Flow Connection is initially identified by a SFEP identifier that correspond to a pre-existing resolved SFEP. The communication session associates each terminal flow connection (identified by the SFEP), with a correlation identifier. The correlation identifier identifies a stream flow (and possibly also a network flow) connection.

NCG is used for negotiation between network and terminal for establishing the correspondence between stream interface addresses and network addresses.

Figure 5-6. Class Diagram of NCG

5.2.2 Behavior Specification

5.2.2.1 Assign SFEP to an User Application

1. UAP-SS invokes the operation GetTypedIdleSFEPList of I_tcsmSfepManager interface of TCSM. In case there are available SFEP is the pool of the available SFEPs, the list of the SFEP identifiers that correspond to the available SFEPs of the specified type is returned to UAP-SS.
2. UAP-SS invokes the operation AssignSFEP of I_tcsmSfepManager of interface of TCSM in order to request from TCSM to assign to itself (UAP-SS) a list of SFEPs, whose identifiers are passed as in parameters. In case these SFEPs do not exist or they are already assigned to an UAP-SS, TCSM throws an exception.

Figure 5-7. Assign a SFEP to a UAP-SS.

5.2.2.2 Set-up a Terminal Flow Connection

1. CSM request from the TCSM to correlate a TFC with a correlation identifier. TCSM identifies the existence of the SFEP. If there isn't any SFEP identified by the specific identifier, or in case that this SFEP is available in the pool or is already assigned to a TFC, TCSM throws an exception. TCSM communicates with application in order to specify the protocols that we be negotiated and give to the application the appropriate connectivity information.
2. TCSM, invokes the GetuserNfepPools operation of all TLAs that exist in terminal.
3. TCSM invokes the SetupNfep operation of the appropriate TLA.
4. TCSM invokes the NfepEnableOperations of the appropriate TLA.
5. CSM invokes the ModifyPotocols operation of TCSM, in order to confirm the protocol stack. TCSM communicates with application in order to be instantiated a media flow.
6. At a later time, TLA invokes the AssociateNfep Operation of TCSM.
7. TCSM invokes the OpenChannel operation of CSAP.

Figure 5-8. Establishment of a TFC

5.2.2.3 Removal of a TFC

1. CSM request from the TCSM to invalidate a TFC with a correlation identifier. If the correlation identifier is not valid, TCSM throws an exception.
2. TCSM communicates with application in order to be the corresponding media flow deactivated.
3. TCSM invokes the RemoveMF operation of the corresponding (to the SFEP) CSAP.
4. TCSM invokes the ReleaseNfep operation of the appropriate TLA.

Figure 5-9. Release of a TFC.

5.2.2.4 Activation of a TFC

1. CSM invokes the tcsmActivateTFC operation of TCSM
2. TCSM invokes the NfepDisableOperations operation of the appropriate TLA.
3. TCSM i communicates with application in order to be the corresponding media flow activated.

Figure 5-10. Activating a TFC

5.2.2.5 Deactivation of a TFC

1. CSM invokes the tcsmDeactivateTFC operation of TCSM
2. TCSM communicates with application in order to be the corresponding media flow deactivated.

3. TCSM invokes the NfepEnableOperations operation of the appropriate TLA.

Figure 5-11. Deactivation of a TFC.

5.3 TINA Communication Session: interfaces descriptions

The TINA communication session interfaces support communication session level interactions across the Ret reference point.T he communication session is separate from the service session. However, as it is part of Ret Usage it needs to be considered.

The communication session does not directly allow requests for connections between domains: these types of requests are handled by the stream binding feature sets of the TINA service session. Instead, the communication session supports lower level requests to set up SFCs that support stream bindings initiated by service sessions.

Figure 5-12 shows the Computational objects that are involved in managing and controlling the Communication Session and it shows the relevant interfaces. Two objects are drawn in

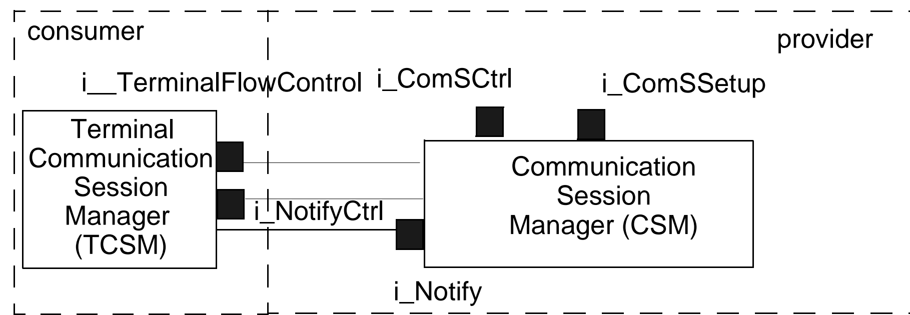


Figure 5-12. Communication Session components

the figure:TCSM (Terminal Communication Session) in the Consumer Domain and CSM (Communication Session Manager) in the provider/retailer domain.

The provider part of the communication session coordinates with the terminal parts to establish SFCs and Terminal Flow Connections (TFCs), and associated the TFCs with the appropriate SFCs and Network Flow Connections (NFCs). It is supported by these components: the Communication Session Manager (CSM) and the Terminal Communication Session Manager (TCSM).

The TINA communication session interfaces specification make the following assumptions:

- Service level stream binding support specifies the stream flow connections.
- All SIs and SFEPs that it is passed are already in existence and are known by a TCSM.
- The TFC is not yet fully established.

The CSM can request the TCSM to associate an SFEP with a TFC identified by a correlation identifier. The CSM can also make requests to determine and set the capabilities and session protocols that can be associated with an SFEP. It can ask for the Resource Flow End Points that are associated with a SFEP, once the capabilities and protocols are selected.

When an NFC is established, the TCSM completes the TFC which it identifies by the correlation identifier associated with the NFEP selected by the NFC. If the communication session is supported by the ConS and TCon reference points, correlation identifiers can be passed over the TCon reference point. The Terminal Layer Adapter, the consumer component supporting the TCon reference point, uses the correlation identifier and the selected NFEP to request the TCSM to complete the TFC. As the communication session needs to cope with non-ConS reference points, the functionality for completing a TFC is also supported between the TCSM and CSM.

As well, the TCSM notify the CSM of faults or problems with the TCSM and TFC changes or failures, including changes to SFEP QoS or removal of SFEPs. This allows the consumers to maintain control of their own terminals. However, TCSMs only act on behalf of TFCs and SFEPs located on its terminal. (I.e. it can not set up, pull down or modify the overall SFC, it can only modify or remove the branch or branches of an SFC that relate to its terminal).

In the following sections interfaces supported by CSM and TCSM are described.

5.3.1TCSM and CSM interfaces: high level descriptions

5.3.1.1 i_TerminalFlowControl

i_TerminalFlowControl: This interface allows the coordination between nodal (TFC) and physical (network - NFC) parts of a stream flow I connection.

- Correlate:
  - Relate a TFC, locally identified in the node by an SFEP, to a unique correlation identifier that identifies the network part of the connection with which it is to be associated.
- Invalidate:
  - Invalidate a correlation identifier. When a SFC is released, its correlation identifier is no longer valid and must be removed. To do this, the CSM must invalidate the SFC at each associated node. When an SFC branch is released, its correlation identifier is no longer valid for the node terminating that branch. The CSM must then invalidate the correlation identifier with the associated TCSM.
- Associate:
  - Associate a TFC, specified by a correlation identifier, with a particular NFEP. The TCSM can complete a TFC on receiving this notification.[1]
- Disassociate:

- Disassociate a nodal binding, specified by a correlation identifier, from a particular NFEP. This allows an SFC to migrate from one NFC to another if required. The CSM may also use this command when release an SFC or SFC branch.
- Resolve:
  - Set the correlation identifier, required capabilities[2] and required session protocols for an SFEP and acquire the RFEPs that can support the SFEP for these requirements
- Get RFEPs:
  - Ask the terminal for RFEPs that may be associated (connected by a TFC) with a given SFEP.
- Query supported capabilities:
  - Ask the terminal for capabilities a group of SFEPs may support.
- Set required capabilities:
  - Set the capabilities an SFEP is required to support.
- Query supported session protocols:
  - Ask the terminal for session protocols a group of SFEPs may support.
- Set required session protocols:
  - Set the session protocols an SFEP is required to support.
- Activate: Activate a TFC. This allows the CSM to activate the SFC
- Deactivate: Deactivate a TFC. This allows the CSM to deactivate the SFC
- QueryTFC:
  - Report on state of a given TFC (identifier by an correlation identifier). This allows the CSM to establish the overall state of an SFC if necessary.
- QuerySFEP:
  - Report on state of an SFEP. This allows the CSM to establish the overall state of an SFC if necessary.

5.3.1.2 i_NotifyCtrl

i_NotifyCtrl: This interface is used to control notifications. It need not directly belong to the object being managed, but could belong to a notification server of object group manager. It is used to enable and disable notifications, and to set destinations of notifications. Multiple destinations for events are supported.

1. This operation is equivalent to the i_tcsmReport Associate operation. It allows a TFC to be completed at the request of the CSM if necessary. Normally we avoid this due to the overhead involved.
2. Capabilities and session protocols indirectly determine the transport QoS an NFC branch needs to support

- enable notifications
  - Enable distribution of notifications to a given list of destinations.
- Disable notifications
  - Disable distribution of notifications to a given list of destinations.
- Set notification destinations
  - Associate one or more destinations, specified by an i_Notify interface reference with a set of specified (possibly all) notifications types. This command can also be used to modify the set of notifications desired.
- Remove notification destinations
  - Disable distribution of notifications to a the given list of destinations, identified by registration id, and remove these destinations from further use.

5.3.1.3 i_Notify

This interface receives notifications from any source.

- Events:
  - Receive one or more notifications. A generic notification format is assumed allowing notifications from many different sources to be accepted. This generic format includes basic notification type and instance identifier attributes, and an extensible list of attributes of any type.

5.3.1.4 i_ComSSetup

This interface is used to setup/release a communication session. These operations are performed through this interface:

- Setup a new communication session
- Release an existing one
- list all communication sessions
- activate suspended communication sessions (previously deactivated)
- deactivate existing communication sessions (previously created)

5.3.1.5 i_ComSCtrl

This interface is used to control SFC and branches of SFCs within an assigned Communication Session. This interface supports three kind of operations:

- operations on SFCs: they allow to create, release, activate and deactivate SFCs
- operation on Branches of SFCs: they allow to add/delete branches to/from SFCs and to activate and deactivate branches of a particular SFC.

- operations to get information: they allow to list all existing SFCs and to get information on a specified SFC.

5.3.2Components and interfaces

5.3.2.1 TCSM

The TCSM supports a user setting up SI and associated SFEPs. These must be registered with the TCSM before they are made available to the service session. The TCSM interacts with the CSM to support requests to aid the setup, modification, and release of SFCs. In particular it can associate SFEPs and their associated TFCs with particular SFCs and NFCs, modify TFCs and associated SFEPs, and activate and deactivate TFCs. The TCSM is responsible for the completion of the TFC and may interact with the TLA and/or the TCSM to achieve this. TLA and UAP interaction functionality is described in Section 7.3.

5.3.2.1.1. Supported Interfaces

- **i_TerminalFlowControl:** This interface allows the coordination between TFC and network (NFC) parts of a stream flow connection. See Section 5.3.1.1
- **i_NotifyCtrl:** Lets a client control notifications and send events to required destinations. See Section 5.3.1.3

5.3.2.1.2. Required Interfaces

- **i_Notify:** Lets a client receive notification events. See Section 5.3.1.3

5.3.2.2 CSM

The CSM supports the establishment of SFCs. It allows its clients (service sessions) to add, activate, deactivate or remove SFCs via the associated session control interface. It also allows clients to manipulate individual stream flows. Each stream flow connection has an interface related solely to it. Similarly, each communication session control interface is dedicated to a single CSM.

To establish a stream flow connection, the CSM must coordinate with each associated TCSM to correlate the nodal part of the connection with the network connection and overall stream flow connection (SFCs are uniquely by the session, and this name may not be known to the user part of the session during establishment). It may also interact with them to modify SFCs or support changes in the nodal part of a connection.

Finally, the CSM interacts with connectivity level components. We usually assume that these components conform to ConS. However different components could be used. To ensure that the communication session can function independently of the underlying connectivity level, we have included operations for completing, modify and removing TFCs at this level.

5.3.2.2.1. Supported Interfaces

- **i_Notify:** Lets the CSM receive notification events. See Section 5.3.1.3. Actually this interface may be placed on other components.

- **i_ComSSetup**: It allows Service Session Manager to setup a Communication Session.
- **i_ComSCtrl**: It allows Service Session Manager to control the Communication Session and the creation/deletion/activation/deactivation of SFCs and branches.

5.3.2.2.2. Required Interfaces

- **i_TerminalFlowControl:** This interface allows the coordination between TFC and network (NFC) parts of a stream flow connection. See Section 5.3.1.1
- **i_NotifyCtrl:** Lets the CSM control notifications and which events it wants to be notified about. See Section 5.3.1.3

5.3.2.2.3. Required Interfaces for service session interactions

- **i_Notify:** Lets the CSM send notification events. Actually this interface may be placed on other components.

Editor's note: the following part need to be updated according to the modification in Ret RP. Operations here listed might not be fully compliant with IDLs.

5.3.2.3 Operations on the i__TerminalFlowControl Interface

5.3.2.3.1. Correlate TFC with SFC and NFC operation

```
void correlate(
        in t_SFEPName sfep,// SFEP identifier
        in t_CorrelationId relatedCon)
            // Correlation ID: SFC and NFC identifier
        raises (e_UserDomainError, e_SIQueryError,
        e_ResrcError);
```

This operation allows the communication session to correlate a SFEP with a terminal flow connection (TFC) a correlation identifier, that identifies the SFC and an associated NFC. (The SFEP is used to identify the initial partial TFC.) This identifier is used later to take actions on the TFC. In particular, it is used to help associate an NFEP with a particular TFC. If the operation is successful it returns. Otherwise, it raises an exception. This operation, or the resolveSFEP operation, is required to setup each branch of an SFC.

SFEPs are identified by the t_SFEPName passed as the sfep parameter. This is the local identifier of the SFEP (unique for the terminal and currently a string). The t_CorrelationId passed as the relatedCon parameter is the correlation identifier which identifies the NFC and SFC to which the TFC is related. It must be unique for each terminal involved in the SFC as well as for the service and connectivity providers. (This can be achieved by combining a service provider identifier with session, SFC and NFC identifiers.) Currently, it is implemented by a sequence of strings.

5.3.2.3.2. Invalidiate operation

```
void invalidate(
        in t_CorrelationId tfc)
        raises (e_UserDomainError, e_CorrelError);
```

This operation allows the communication session to invalidate a correlation identifier and related TFC. The correlation identifier is passed by the t_CorrelationId used as the tfc parameter. After this operation, the identifier can no longer be used to identify a TFC, and the terminal can invalidate the TFC and release its associated resources. If there is an error an exception is raised. Otherwise the operation just returns. This operation is required to remove branches of an SFC.

5.3.2.3.3. Get Rfeps related with an SFEP operation

```
void getRelatedRfeps(
        in t_SFEPName sfep, // SFEP name
        out t_RFEPDescList rfeps);
        raises (e_UserDomainError, e_SIQueryError,
                e_ResrcError);
```

This operation allows the communication session to ask for RFEPs that may be associated with an SFEP. These RFEPs form the pool from which NFC can be set up. An SFEP may be associated with many RFEPs. The RFEPs it may be associated with can be restricted by the capabilities, session protocols and transport requirements associated with the SFEP. If the operation is successful, it returns the RFEPs descriptions. If there is an error, it raises and exception. This operation, or the resolveSFEP operation, is required to setup each branch of an SFC, if no RFEP information is passed in the initial SFEP description.

SFEPs are identified by the t_SFEPName passed as the sfep parameter. This is the local identifier of the SFEP. The t_RFEPDescList passed as the rfeps output parameter describes NFEPs or NFEPPools which can be associated with the SFEP. For the communication level, these need to include attributes identifying local connectivity providers (i.e. whose network is the RFEP connected to), the layer network technology and other information need to determine the connectivity provider. For connectivity and layer networks they also need to include transport protocols requirements and transport quality requirements. (Transport quality requirements may be included in the RFEP descriptions by the terminal or set by the CSM from data returned from setCapability and setSProtocol operations).

5.3.2.3.4. Resolve SFEP capabilities, protocols and RFeps operation

```
void resolveSFEP(
        in t_SFEPName sfep, // SFEP name
        in t_CapabilityDesc capability,
        in t_SessionProtocolDescList protocols,
        in t_CorrelationId,
        out t_RFEPDescList rfeps)
        raises (e_UserDomainError, e_SIQueryError,
        e_ResrcError,     e_SProtocolSelectError,
```

```
                    e_CapabilitySelectError);
```

This operation allows the communication session to set capabilities, session protocols, correlation identifier and ask for RFEPs that may be associated with an SFEP in one combined operation. This operation is included for efficiency reasons, so that the communication session can set up the SFEP and associated partial TFC in one operation. (The full TFC can not be set up until it is associated with a single NFEP.) If successful, this operation returns the list of RFEPs that are available to this SFEP. The RFEPs should include transport quality and protocol requirements.

SFEPs are identified by the t_SFEPName passed as the sfep parameter. This is the local identifier of the SFEP. The capability and protocols parameters describe the capabilities (e.g. codec capabilities) and session protocols to be used with the SFEP. Both capabilities and session protocols are described by a key (local identifier for the terminal's capability or session protocol table), an identifier (agreed unique identifier, relating either to standard or proprietary capability or protocol). The correlation identifier is described in Section 5.3.2.3.1. The operation returns a set of RFEPs if it is successful, as described in Section 5.3.2.3.3, which should contain related transport quality requirements. If there is an error, an exception is raised.

This operation, or some combination of the correlate, setCapability, setSessionProtocol and getRFEPs operations is required to setup each branch of an SFC. In addition, the queryCapabilities and querySessionProtocols may be required to determine compatible SFEPs on each branch. (Compatible means that there is some means to interwork the capability and session protocols selected for each SFEP).

5.3.2.3.5. Associate SFEP with NFEP operation

```
void associate(
          in t_CorrelationId tfc,
          in t_FepName finalNFep,
          raises (     e_UserDomainError, e_CorrelError,
                       e_ResrcError,e_NFEPIdError );
```

This operation allows the communication session to associate a NFEP with a TFC, identified by the correlation identifier, completing the TFC set up. The TFC may be completed at the TCon level, assuming a similar operation between the TLA and TCSM. This operation is included here to allow use of non ConS/TCon providers. It also allows SFCs to be multiplexed over the one NFC by associating multiple TFCs with the one NFC. The t_CorrelationId passed as the tfc parameter gives the correlation identifier, and the t_FepName passed as the finalNFEP identifies the NFEP that has been selected at the layer network (or equivalent) to terminate the associated NFC. If the operation is successful it returns and completes the TFC (and hence the NFC). Otherwise, it raises an exception. This operation is optionally required to setup each branch of an SFC.

5.3.2.3.6. Disassociate SFEP from NFEP operation

```
void disassociate(
          in t_CorrelationId tfc,
          in t_FepName finalNFep,
```

```
raises (   e_UserDomainError,e_CorrelError,
e_NFEPIdError);
```

This operation allows the communication session to disassociate a NFEP with a TFC, identified by its correlation identifier. This can be used to start deleting an SFC or SFC branch. It could also be used to swap an SFEP from connection with one NFEP to connection with another. This operation does not remove the entire TFC (though it is no longer operational). Rather it prepares the TFC for a change in setup. The t_CorrelationId passed as the tfc parameter gives the correlation identifier, while the t_FepName passed as the finalNFep identifies the NFEP to be removed from the TFC. If the operation is successful it returns. Otherwise, it raises an exception. This operation is required to remove branches of an SFC or change the NFEP with which they are associated.

5.3.2.3.7. Query capabilities that the SFEP can support

```
void queryCapabilities(
        in t_SFEPQueryDescList sfeps,
        out t_SFEPCapDescList sfepCaps,
        out t_CapabilityDescList detailedCaps,
        out t_CapabilityRelationList restrictions)
        raises (e_UserDomainError, e_SIGeneralError );
```

This operation allows the communication session to query a terminal for the capabilities that a particular SFEP or SFEPs can support. If successful, this operation returns a list of SFEPs and their associated capabilities indicated by a key, a table of capability descriptions, and a list of restrictions that apply to these capabilities. Otherwise, it raises an exception. This operation will be required for setting up each branch of an SFC, unless capability information is included in the SFEP description.

The t_SFEPQueryDescList passed as the sfeps identifies the SFEPs by their local SFEPName, and includes a mediaParameters list that may restrict the capabilities to be selected. The t_SFEPCapDescList returned as the sfepCaps parameter lists each of the submitted SFEPs with list of keys that identify the precise capabilities that may be supported. Each key may be associated with a priority to indicate preferred capability sets.

The keys relate to the t_CapabilityDescList passed as the detailedCaps parameter. This parameter is a table of detailed descriptions of the capabilities, and included a unique identifier, the terminal’s key, and a set of attributes describing the capabilities (e.g. codec specification) and possible restrictions (e.g. quality requirements, or required session (or transport) protocols).

Finally, t_CapabilityRelationList passed as the restrictions parameter details relations between the capabilities that may restrict what combination of capabilities may be used. (Because of these relations, submitting a group of SFEPs is desirable so the CSM can determine what may be used together.) These relations are described by lists of simultaneous and alternative capabilities identified by keys.

5.3.2.3.8. Set capabilities that the SFEP is required to support

```
void setCapability(
```

```
in t_SFEPName sfep,
in t_CapabilityDesc selection,
out t_TransportQuality reqQual)
raises (        e_UserDomainError, e_SIGeneralError,
                e_CapabilitySelectError);
```

This operation allows a communication session to set the particular capabilities the SFEP should support. This operation is needed to ensure that SFEPs support compabilitible capabilities. In particular, these capabilities can map to particular CODECs and support capabilities. If successful the operation returns a list of transport quality requirements. Otherwise, an exception is raised. This operation, or the resolveSFEP operation, is required to setup each branch of an SFC.

The SFEP is identified by the t_SFEPName passed as the sfep parameter. The capabilities to be set are described by the t_CapabilityDesc passed as the selection parameter. This description is the same as Section 5.3.2.3.7. The returned quality requirements are described by the t_TransportQuality passed as the reqQual parameter. This quality is described by a list of attribute tag value pairs.

5.3.2.3.9. Query session protocols that the SFEP can support

```
void querySessionProtocol(
        in t_SFEPQueryDescList sfeps,
        out t_SFEPSProtocolDescList sfepProtocols,
        out t_SProtocolDescList detailedProtocols,
        out t_SProtocolRelationList restrictions)
        raises (e_UserDomainError, e_SIGeneralError );
```

This operation allows the communication session to query a terminal for the session protocols that a particular SFEP or SFEPs can support. If successful, this operation returns a list of SFEPs and their associated session protocols indicated by a key, a table of session protocol descriptions, and a list of restrictions that apply to these session protocols. Otherwise, it raises an exception. This operation will be required for setting up each branch of an SFC, unless session protocol information is included in the SFEP description.

The t_SFEPQueryDescList passed as the sfeps identifies the SFEPs by their local SFEP-Name, and includes a mediaParameters list that may restrict the session protocols to be selected. The t_SFEPSProtocolDescList returned as the sfepProtocols parameter lists each of the submitted SFEPs with list of keys that identify the precise session protocols that may be supported. Each key may be associated with a priority to indicate preferred session protocols.

The keys relate to the t_SProtocolDescList passed as the detailedProtocols parameter. This parameter is a table of detailed descriptions of the capabilities, and included a unique identifier, the terminal's key, and sets of attributes describing the session protocols (e.g. codec specification) and possible restrictions (e.g. quality requirements or required transport protocols).

Finally, t_SProtocolRelationList passed as the restrictions parameter details relations between the capabilities that may restrict what combination of session protocols may be used. These relations are described by list of simultaneous and alternative session protocols identified by keys. Note that some protocols may be required to be used together, as they may form part of stack.

5.3.2.3.10. Set session protocols that the SFEP is required to support

```
void setSessionProtocol(
        in t_SFEPName sfep,
        in t_SProtocolDescList selection,
        out t_TransportQuality reqQual)
        raises (    e_UserDomainError, e_SIGeneralError,
                    e_SProtocolSelectError);
```

This operation allows a communication session to set the particular session protocols the SFEP should support. This operation is needed to ensure that SFEPs support compabilitible session protocols. If successful the operation may returns a list of transport quality requirements. Otherwise, an exception is raised. Session protocols may sit both above and below the capability set (but are always above transport related protocols that are determined at the layer network level). An SFEP may be associated with a stack of such protocols, so a number may be set at once. This operation, or the resolveSFEP operation, is required to setup each branch of an SFC.

The SFEP is identified by the t_SFEPName passed as the sfep parameter. The session protocols to be set are described by the t_SProtocolDescList passed as the selection parameter. This description is the same as Section 5.3.2.3.9. The returned quality requirements are described by the t_TransportQuality passed as the reqQual parameter as described in Section 5.3.2.3.8

5.3.2.3.11. Activate a TFC

```
void activate(
        in t_CorrelationId tfc)
        raises (    e_UserDomainError, e_CorrelError );
```

This operation allows the communication session to activate a TFC identified by a correlation identifier. The tfc parameter gives the correlation identifier. The operation requests a change in the administrative status of the TFC and associated resources (including the SFEP) to unlocked. If the operation is successful, it returns. Otherwise an exception is raised. This operation is required to activate branches of a SFC.

5.3.2.3.12. Deactivate a TFC

```
void deactivate(
        in t_CorrelationId tfc)
        raises (    e_UserDomainError, e_CorrelError );
```

This operation allows the communication session to deactivate a TFC. identified by a correlation identifier. The tfc parameter gives the correlation identifier. The operation requests a change in the administrative status of the TFC and associated resources (including the SFEP) to locked. If the operation is successful, it returns. Otherwise an exception is raised. This operation is required to deactivate branches of an SFC.

5.3.2.3.13. Query the state of the TFC

```
void queryTFCs(
        in t_CorrelationIdList tfcs,
        out t_TFCDescList state)
        raises (    e_UserDomainError, e_CorrelError );
```

This operation allows the communication session to query the status of a given TFCs. The TFCs are identified by the t_CorrelationIdList passed as the tfcs parameter. If successful, this operation returns a list of TFC descriptions which describe their current state (administrative and operational) and other details. Otherwise an exception is raised.

5.3.2.3.14. Query the state of the SFEP

```
void querySFEPs(
        in t_SFEPNameList sfep,
        out t_SFEPStatusDescList state)
        raises (    e_UserDomainError, e_SIQueryError );
```

This operation allows the communication session to query the status of a terminals SFEPs. The SFEPs are identified by the t_SFEPNameList passed as the sfep parameter. If successful, this operation returns a list of SFEP descriptions which describe their current state (administrative and operational) and other details. Otherwise an exception is raised.

5.3.2.4 Operations on the i__ComSSetupI Interface

```
 void  setup_communication_session
      (in  t_SuccessCriterioncriterion,
 in  t_SFCDescList sfcDescList,
 out t_ComSessionName comSessionName,
 out i_ComSCtrlcomSCtrl,
 out t_SFCRespList  sfcRespList)
raises (Error);

 void release_com_session
(in t_ComSessionName comSessionName)
raises (Error);

 void  activate_communication_session
(in t_ComSessionNamecomSessionName)
raises (Error);

 void  deactivate_communication_session
```

```
(in t_ComSessionNamecomSessionName)
raises (Error);

 void list_all_communication_sessions
(in t_ComSessionNameListcomSessionNameList)
raises (Error);
```

5.3.2.5 Operations on the i_ComSCtrl Interface

Operations on Stream flow connections:

```
 void setup_flow_connections
(in  t_SuccessCriterioncriterion,
 in  t_SFCDescList sfcDescList,
 out t_SFCRespList sfcSetList)
raises (Error) ;

void release_flow_connections
 (in  t_SuccessCriterioncriterion,
  in t_SFCNameList sfcList,
  out t_SFCRespList sfcRelList)
 raises (Error);

void activate_flow_connections
 (in  t_SuccessCriterioncriterion,
  in  t_SFCNameList     sfcList,
  out t_SFCRespList sfcActList)
 raises (Error);

void deactivate_flow_connections
 (in  t_SuccessCriterioncriterion,
  in  t_SFCNameListsfcList,
  out t_SFCRespList sfcDeactList)
 raises (Error);
```

Operations on branches

```
void add_branches
 (in  t_SuccessCriterioncriterion,
  in  t_SFCName sfcName,
  in  t_SFepDescList sfepDescList,
  out t_SFepRefList boundList)
 raises (Error);

void delete_branches
 (in  t_SuccessCriterioncriterion,
  in  t_SFCName sfcName,
  in  t_SFepRefList list,
  out t_SFepRefList dellist)
 raises (Error);
```

```
void activate_branches
  (in  t_SuccessCriterioncriterion,
   in  t_SFCName sfcName,
   in  t_SFepRefListlist,
   out t_SFepRefList actList)
  raises (Error);

void deactivate_branches
  (in  t_SuccessCriterioncriterion,
   in  t_SFCName sfcName,
   in  t_SFepRefListlist,
   out t_SFepRefList deactList)
  raises (Error);
```

Operations to retrieve information on SFCs

```
void list_all_SFCs
  (out  t_SFCNameList  sfcList)
  raises (Error);

void get_flow_conn_info
  (in  t_SFCName sfcName,
   out t_SFCDesc sfcDesc)
  raises (Error);
```

6. Connectivity Session Related Components

This view is used by the communication level(CSM) to establish the network part of its stream flow connections. The network requirements information for the Stream Flow Connections which was previously determined based on the network capabilities exchange negotiation (between the CSM and TCSM) is passed from the CSM to this level.The function provided at this level is the setup and management of connectivity sessions and network flow connections. The interaction between the connectivity level and the communication level above it is part of the TINA Connectivity Session (ConS) Reference Point.

The Connectivity session is related to a technology independent abstraction of network connections. A connectivity session is a context in which one or more network flow connections can be established and managed as a unit. Setting up a network flow connection involves setting up one or more trails in the layer networks that make up a connectivity layer network. More than one trail needs to be created to realize a network flow connection if the end points of the network flow connection (called Network Flow Endpoints) are on different layer networks. Figure 6-1 illustrates the connectivity level computational objects namely the Connection Coordinator Factory (CCF), theConnection Coordinator (CC), and the Flow Connection Coordinator (FCC). They are illustrated with objects from other groups with which they interact.

As shown, the connectivity session objects (part of the connectivity provider domain) communicates with the communication level (CSM) in the service provider domain and communicates with the Layer Network Components below it. In relation to the ConS Reference Point, the three Connectivity session objects correspond to the usage part.

6.1 Overview of Connectivity Session

In Figure 6-1, connectivity session related components are shown. There are three components in the connectivity session.

- Connection Coordinator Factory (CCF)
- Connection Coordinator (CC)
- Flow Connection Coordinator (FCC)

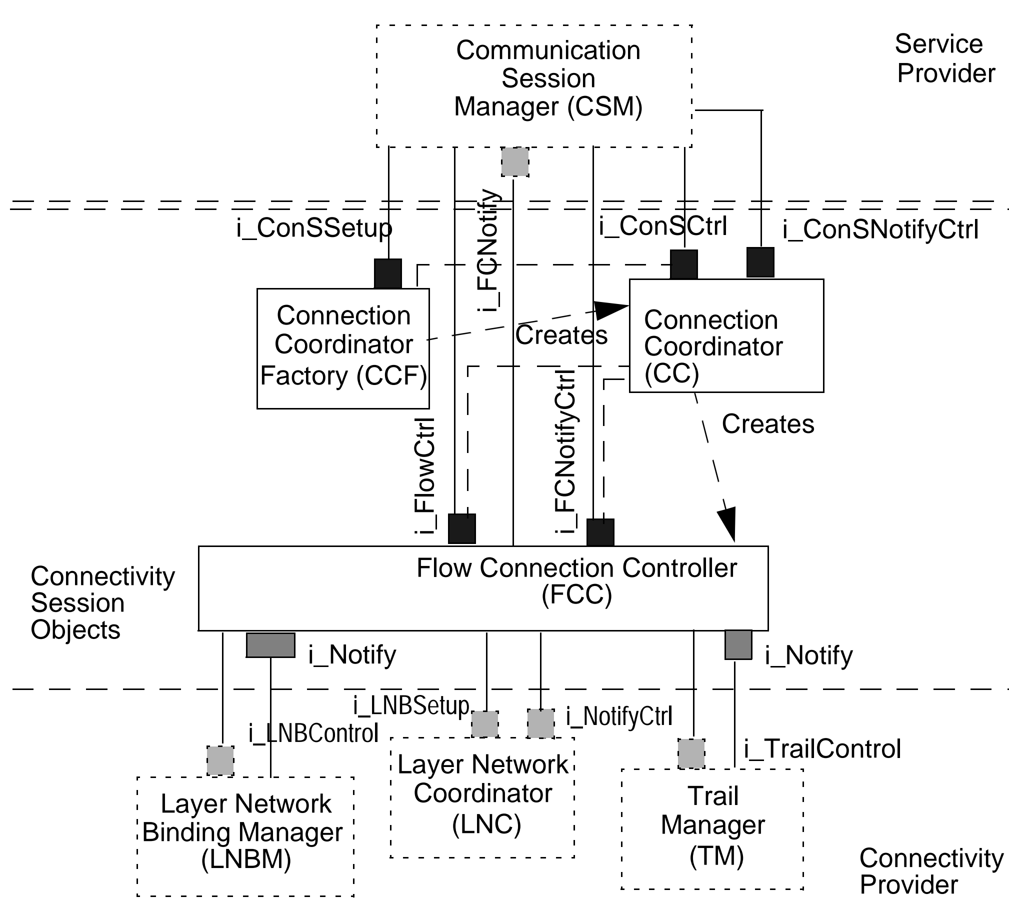


Figure 6-1. Connectivity session related objects

The object Connection Coordinator Factory(CCF) provides an operation for the setting up of a connectivity session.It serves as the factory object for connectivity sessions. It provides access to the connectivity level and its technology independent network view. At connectivity session setup time, one or more network flow connections may also be setup.

The Connection Coordinator is created by the CCF. It is recommended that an instance of this object exists for each connectivity session that is setup. The CC is the session control object and provides an operation for the management of an individual connectivity session including addition, removal and modification of network flow connections that are contained in the connectivity session. In complying with the ConS Reference Point, it is recommended that a CC is associated with a single connectivity session. When a Network Flow Connection setup is requested, the CC creates a Flow Connection Controller(FCC) object and delegates the setup request to that object.

The FCC object provides management operations for the network flow connections associated with it including addition of branches, removal of branches and network flow connection release.The Connectivity Providers Flow Connection Coordinator determines that both the Network Flow End Points point to network termination points in different layer networks. As a result the Flow Connection Coordinator locates an interworking unit that is capable of handling the interworking between two layer networks and requests an interworking connection. After successfully setting up the interworking connection, the interworking unit will return Network Flow End Points pointing to one network termination point and another Network Flow End Point pointing to the other Network Termination point of the interworking connection. The connectivity session components interact with layer network related COs and support the interfaces required by them.

6.1.1 Relationship to ConS -RP

The Connectivity Service Reference Point (ConS-RP) is defined between business administrative domains that provide connectivity services (network transport services) and business administrative domains that are using services on behalf of their customers.

The provider of the connectivity service (Connectivity Provider) enables its clients (connectivity users) to setup, modify and release a connectivity session that is composed of one or more NFCs. The connectivity service also allows a connectivity user to manage (i.e. setup, release, and modify) a group of network flow connections as an aggregated unit(ie.in a connectivity session, a connectivity user can release all network flow connections that are part of the connectivity session in one operation) making the exchange of information more efficient thus reducing setup delay.

6.1.2 Connectivity Session (ConnS)

An instance of this object type contains one or more stream flow connections. It represents the total set of communication resources involved in a communication session. A client of a communication session can establish, release and manipulate stream flow connections.

Attributes:

- Communication Session Name
- Success Criterion (AllorNone, BestEffort)
- Stream Flow Connection Description List

6.1.3 Network Flow Connection (NFC)

An instance of this object represents the resource that transfers information across the connectivity layer network of a TINA network. The information is transported between a group of Network Flow End Points. The connection topology may be point-to-point bidirectional, point-to-point unidirectional, or point-to-multipoint unidirectional.

A network flow connection can be established, released, activated and deactivated. New branches can be added or deleted to/from a stream flow connection. A network flow connection is always either point-to-point or point-to-multipoint unidirectional (when non specified it is assumed to be point-to-multipoint by default).

Attributes:

- Network Flow Connection Name
- Success Criterion (AllorNone, BestEffort)
- Network Flow Connection Topology {PointToPoint, PointToMultipoint}
- Network Flow End Point Description List

6.1.4 Network Flow End Point (NFEP)

This class represents an endpoint of a connectivity layer network. Network flow end points can be uni or bidirectional. A NFEP can be a source or a source/sink in case of being bidirectional. The NFEPs are technology independent representation of a network termination points. They represent the required Termination points (existent in the terminal) to accomplish the connection.

Attributes:

- Network Flow End Point Name
- Network Flow Endpoint Direction (Source, Sink,SourceSink)
- Connectivity Provider
- Network Address List(+ eventually restrictions (eg. VPI/VCI)
- Administrative State (Locked or Unlocked)

Import NFEP.idl

6.1.5 Network Flow Connection Branch

An instance of this association represents a branch of a network flow connection. It relates a stream flow connection object and a sink stream flow endpoint.

6.1.6 Related Information Model Fragment

The information structure that represents such information about all network flow connections of a connectivity session is called a Physical Connection Graph. Figure 6-2 shows the OMT diagram for a Physical Connection Graph.

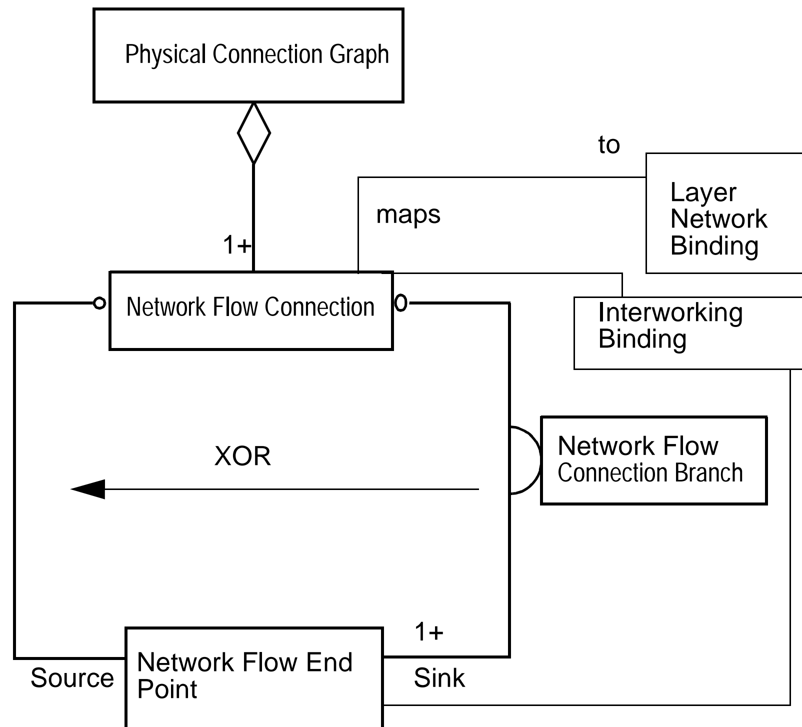


Figure 6-2. . OMT diagram for ConS-RP

The objects provide the vocabulary for both domains to control and manage network connections. The connectivity layer network is a container of all the network EndPoints and is an identifier for the connectivity domain. it represents the networks owned by the connectivity provider and can be used to perform high level maintenance on a network and check its state. The connectivity layer network contains pools of flow end points which represent groups of endpoints of these network connections. A pool can be the end of a link on which ATM packets are carried with a single wall socket or a group of connections of different technology eg. a computer connected to both a telephone network and an Ethernet network using multiple sockets. The endpoints is an abstract concept, since it will always be either a root (generating traffic or a leaf consuming traffic).

The capability to carry traffic between one root EndPoints and one or several leaf Endpoints is labelled as flow connection and allows the identification for manipulation of all the EndPoints involved in the flow connection. Flow Connections can be grouped together into a

connectivity session. To express preferences about routing (eg. to provide lip-synch when video and voice are carried on separate connections through the network) the route related information object is used.

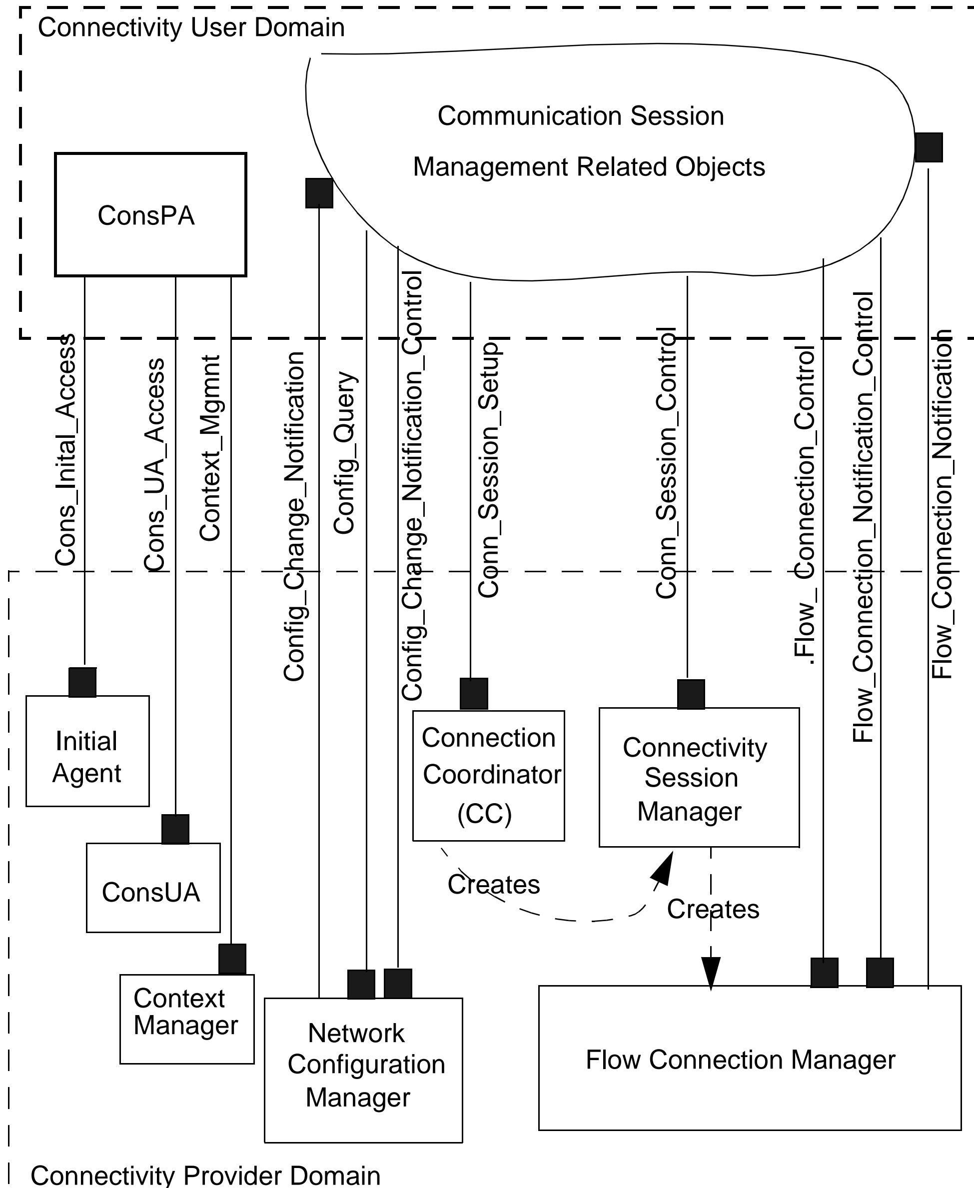


Figure 6-3. Relationship to ConS

6.2 Connection Coordinator Factory (CCF)

An instance of this type exists in a connectivity provider domain. It provides connectivity session establishment and release functions. At the time of a connectivity session setup, a client can request setup of one or more network flow connections, the following information is provided by the client:

Network Flow Connection Name

Root NFEP descriptors

Correlation identifier

A NFEP descriptor includes NFEP Pool identification and the bandwidth and QoS parameters associated with the NFEP. Upon receipt of a connectivity session setup request, the CCF either creates a new CC object for controlling the new connectivity session or assigns the control responsibility to an existing CC, depending on the scheduling policy used by the CCF. In either case, the CCF delegates to the CC object the communication setup request and passes along the connectivity session information received from the client.

Figure 6-4. CCF CO

6.2.1 Behavior Specification

Assumptions:

- A communication session maps one-to-one in a connectivity session.

- A SFC maps one-to-one in a NFC. Two SFC can map into one NFC.

- A SFEP maps one-to-one in a NFEP.

- The previous assumptions lead to the fact that only unidirectional connections are supported at the service and network levels. The default mapping is one-to-one.

- Only simple exception handling has been considered.

In order to illustrate the behavior of the CC some event sequences are shown in the following more representative scenarios:

- Connectivity Session setup

- (more will be added in the final version)

6.2.1.1 Connectivity Session Setup

1. CSM requests the CC for creation of a connectivity session. This request should include as parameters:

- The Success Criterion that indicates the criterion for a successful completion of the session set operation. The possible values are AllorNone or BestEffort. In the first case the operation will be considered successful if all the NFCs are setup successfully while in the second case the operation will succeed if at least one NFC is successfully setup.

- A list containing the NFCs that belong to the requested connectivity session. A least one NFC should be specified. In order to describe properly a NFC, a Name should be given by the CSM to each of NFC, a particular Success Criterion (indicating a successful NFC if all the leafs are successfully bound or at least one) and the topology of the network connection should be defined. Additionally, a correlation identifier should be defined for each NFC and the NFEPs information should be given. As attributes the NFEP Name, the administrative status, the NFEP Direction, the LNW Type, the TLA Reference and the list of possible network addresses should be defined for each End Point together with the transport protocol.

2. The references of the ConSCtrl Interface will be sent back to the CSM together with the NFC response parameters (i.e. NFC Name, the respective NFepRefList of the bound NFEPs and the FCCtrl interface reference).

There is only one ConSSetup interface per CC. It is the factory of instances of ConSCtrl interfaces. The CC will instantiate one ConSCtrl Interface per connectivity session. This interface allows the CSM to control the session by requesting new network flow connections, deleting existing ones, activate, deactivate and modify them. Additionally in order to add (delete) new branches and activate (or deactivate) them separate interfaces are created for each NFC. This mechanism will allow to speed up the process of adding and deleting branches to the NFCs. Conceptually it is in line with the concepts below the CC (i.e. LNC) where the concept of session does not exist anymore. The NFC will be mapped one-to-one in a trail.

The CC will issue a similar request to the LNC; a request for a trail setup.

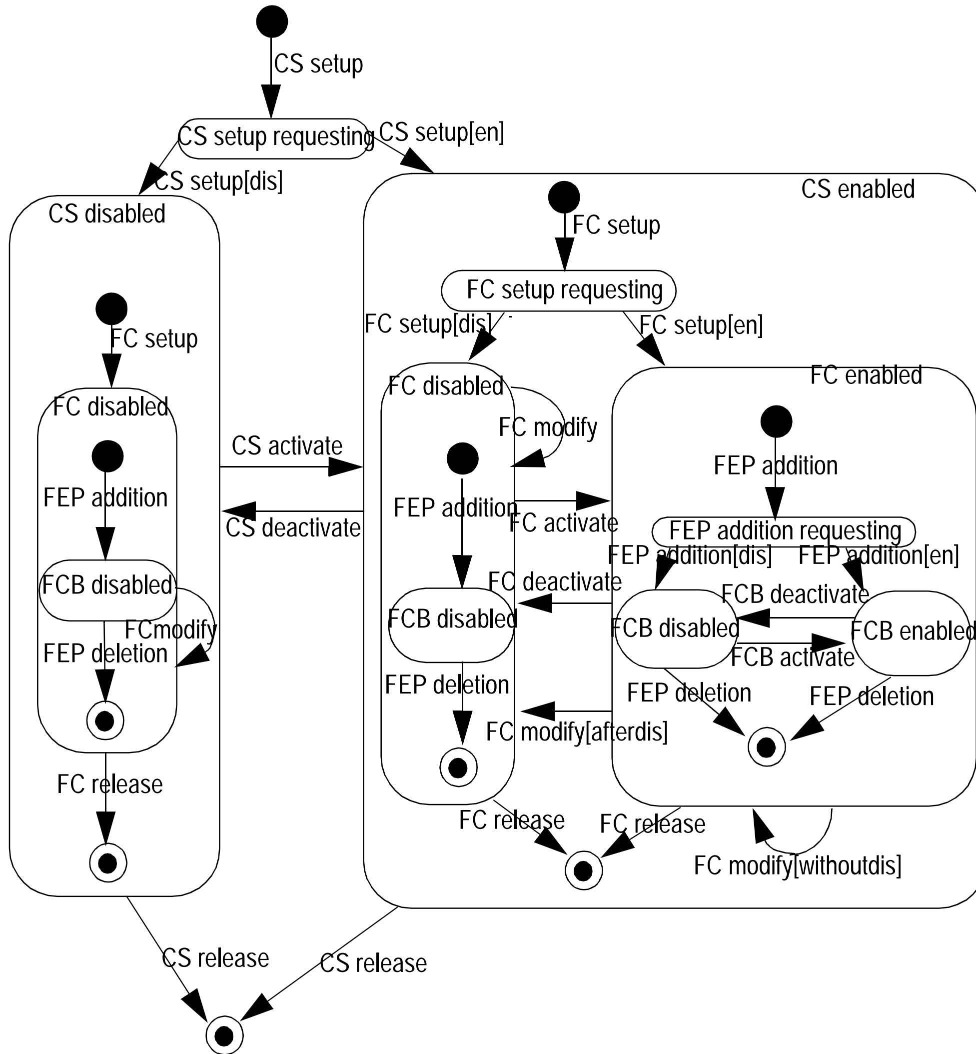
5. State Diagram

The behavior of the ConnectivityService Provider and the interactions on the ConS-RP are explained by the state diagram below. A connectivity session and its component flow connections have separate administrative states. however, these administrative states are subject to the following rules.

- If the administrative state of a connectivity session is unlocked, then the administrative state of each component flow connection may be either locked or unlocked, it can be individually changed.
- If the administrative state of a connectivity session is locked, the administrative state of each component flow connection is Locked.
- Similar rules hold between the administrative state of a flow connection and the administrative state of its branches.If the administrative state of a flow connection is unlocked, then the administrative state of a branch of the flow connection

may be either unlocked or locked; it can be individually changed.If the administrative state of a flow connection is locked, the administrative state of each branch of the flow connection is Locked.

Usage part



6.2.1.2 Connection Coordinator (CC)

One or more instance of this object type exists in a connectivity provider domain. Each CC object manages one or more connectivity sessions. For each connectivity session under its control, the CC offers the following management capabilities:

Network Flow Connection setup

Addition/removal of branches to/from

6.3 Flow Connection Controller

One instance of this object type is associated with each NFC. The Flow Connection controller(FCC) is created by the CC associated with the connectivity session of which the NFC is a part. The FCC is responsible for translating the NFC, with its technology dependent view to a particular layer Network, or networks should be used. A selected Layer network should support the requested QoS and provide suitable termination for at least one branch of the NFC. If such a connection cannot be established across a single layer network, then the FCC would have to setup connections across two different layer networks. The FCC would have to find a suitable adapter and setup trails across both layer networks in the case of connection oriented networks. The FCC uses the LNC interface from the Layer Network to setup an end-end connectivity across a Layer Network(connection oriented or connectionless Layer Network). This is because the concept of trails is only usable for Connection oriented Networks while for connectionless the trail concept is unsuitable. To provide for this transparent mapping, the concept is realised by using a Layer Network Binding and managed by the Layer Network Binding Manager(LNBM). A FCC object provides the following management operations for the NFC under its control:

Network Flow Connection setup

Addition/Removal of branches if the NFC is point to multipoint connection

Activation of Bandwidth and Qos parameters of the network flow connection

Activation of the Network Flow Connection or its branches

Deactivation of the network flow connection or its branches

Network flow connection release

6.2.1 Related Information Model fragment

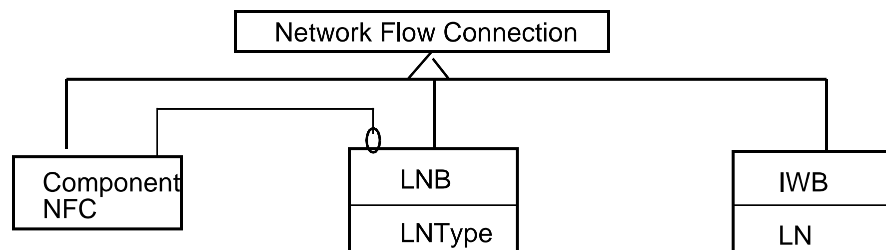


Figure 6-5. FCC related Information Model fragment

In the information model, for interworking between two different Layer Network the FCC makes use of the information for Interworking using the Interworking Binding(IWB) with parameters about the Layer Networks, the Layer Network Binding with parameters about the Layer Network Type and and the information on the Component NFCs.

6.4 TINA Connectivity Session: interfaces description

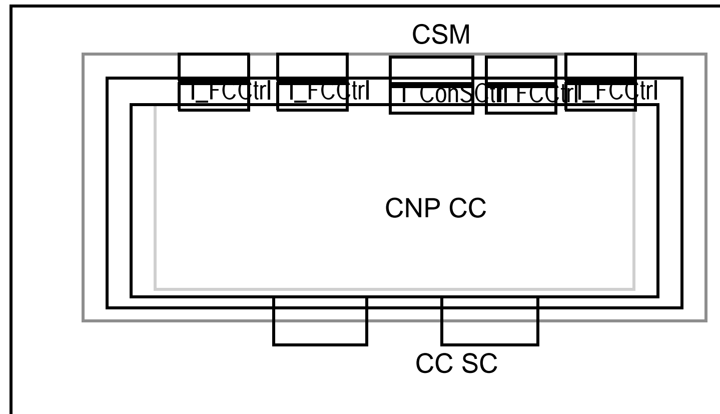


Figure 6-6. Connectivity session components interfaces

6.4.1 CCF, CC and FCC interfaces: high level descriptions

6.4.1.1 i_ConSSetup

This interface allows the establishment of connectivity sessions and their associated computation objects. It also allows some general connectivity level management operations. These operations are primarily included to aid management, such as recovery and transfer of connectivity session control.

- Establish a connectivity session

  -This operation is used for setting up a connectivity session. If the invocation is successful, this operation creates a CC object which provides operations for manipulating the newly created connectivity session. When a connectivity session set up is requested, multiple NFCs can be established at the same time. Control interfaces associated with each NFC are returned.

- List connectivity sessions:

  -Return a list of connectivity session identifiers. Some filtering may be used to manipulate which connectivity sessions are listed.

- Get a connectivity session:

-Return the control interface reference to a connectivity session associated with the given connectivity session identifier.

- Delete a connectivity session:

  -Delete the connectivity session(s) associated with the given identifiers. This operation is intended for management.

6.4.1.2 Required Interfaces

The following interface are suggested for initialization of the CCs that support a single connectivity session. The creation of objects depends on DPE services.

i_ConSCtrl: Once the CC is created, the CCF can use this interface to establish any requested NFCs.

6.4.1.3 i_ConSCtrl

This interface provides operations for setup, activation, deactivation, and release of one or more flow connections of the connectivity session associated with the Connection Coordinator object. It also provides a connectivity session query operation and an operation to acquire a notification control interface associated with the connectivity session.

This interface provides the following operations:

- Setup flow connections

  -This operation is used for setting up one or more NFCs within the connectivity session associated with the CC. If the invocation is successful, this operation creates one or more FCC objects which provides operations for manipulating the newly created NFCs. When a NFC set up is requested, the client should specify at least one branch of the NFC. NFCs can be established in an active or inactive state.

- Activate flow connections

  -This operation is used for activating one or more (possibly all) NFCs of the connectivity session associated with the CC. One of the output is List of triples of the form <FC, FEP, Status> where FC is the name of a NFC, FEP is the name of an endpoint (root or leaf) of the NFC referenced by FC, and Status is either “Activated” or “UnableToActivate”.

- Deactivate flow connections

  -This operation is used for deactivating one or more (possibly all) NFCs of the connectivity session associated with the CC.

- Release connectivity session

  -This operation is used for releasing the connectivity session associated with the CC. When the connectivity session is released, all component NFCs are also released and the CC is deleted.

- Release flow connections

-This operation is used for deactivating one or more (possibly all) NFCs of the connectivity session associated with the CC.

- Get notification control interface

  -Return the i_ConSNotifyCtrl interface reference associated with this connectivity session. This interface allows a client to modify enable or disable notifications at a session level or change the default destination.

- Get connectivity session info

  -This operation is used for retrieving the connectivity session information. Outputs are State of the connectivity session and List of names of NFCs that are components of the connectivity session.

- Get flow connection control interfaces

  -This operation is used for obtaining the references to the i_FlowCtrl and i_FCNotifyCtrl interfaces associated with one or more (possibly all) flow connections of the connectivity session associated with the CC.

6.4.1.4 i_ConSNotifyCtrl

This is a suggested interface for controlling notifications at the session level. This allows notifications associated with the connectivity session and its associated NFCs to be controlled together or individually, as desired.

- Enable flow connection notification

  -Enables notifications associated with either listed NFCs or the connectivity session (i.e. all NFCs in the connectivity session).

- Disable flow connection notification

  -Suspends sending notifications associated with either listed NFCs or the connectivity session (i.e. all NFCs in the connectivity session).

- Update flow connection notification destination

  -This operation is used to set the default flow connection notification destination parameter for this connectivity session only.

6.4.1.5 Required Interfaces

The following interface are suggested for initialization of the FCCs:

i_FlowCtrl: Once the FCC is created, the CC can use this interface to initialize NFCs.

6.4.1.6 i_FlowCtrl

This interface provides operations for addition, removal, modification, activation, and deactivation of one or more branches (possibly all branches) of the NFC associated with the FCC object. It also provides a query operation for obtaining information on the associated NFC.

This interface provides the following operations:

- .Add flow connection branches

  -This operation is used for adding one or more branches to the NFC associated with the FCC object. NFC branches can be added in an active or inactive state.

- Delete flow connection branches

  -This operation is used for removing one or more branches (possibly all branches) from the flow connection associated with the FCC object. A request to delete all branches releases the NFC, and removes the related FCC. (The i_ConSCtrl interface can also be used to release NFCs).

- Activate flow connection branches

  -This operation is used for activating one or more branches (possibly all branches) of the flow connection associated with the FCC object.

- Deactivate flow connection branches

  -This operation is used for deactivating one or more branches (possibly all branches) of the flow connection associated with the FCC object.

- Modify flow connection branches

  -This operation is used for adding one or more branches to the flow connection associated with the FCC object.

- Get flow connection info

  -This operation is used for retrieving the flow connection information.

6.4.1.7 i_FCNotifyCtrl

This interface provides operations for controlling the emission of notifications regarding the network flow connection associated with the FCC object.

It provides the following operations:

- Enable flow connection notification

  -This operation is used for instructing the FCC to emit notifications regarding the NFC.

- Disable flow connection notification

  -This operation is used for instructing the FCC to suspend emission of notifications regarding the NFC.

- Set flow connection notification destination

  -This operation is used to communicate the interface reference to the FCC which its client will use to receive flow connection notifications.

6.4.1.8 Generic or Predefined Supported Interfaces

The following generic or predefined interfaces are supported by the FCC:

i_Notify: the FCC may support this interface to receive notifications associated with the trails it establishes in the layer network, see Section

6.4.1.9 Required interfaces

The following interfaces are required by the FCC:

i_FCNotify: The FCC sends notifications regarding its associated flow connection to this interface. See. Section

i_TrailSetup: The FCC uses this interface to establish a trail, see Section

i_TrailControl: The FCC uses this interface to control a trail it uses or has established, see Section

i_LNBSetup: The FCC uses this interface to request the LNBM (Layer Network Binding manager) to establish a Layer Network Binding

i_LNBControl:The FCC uses this interface on the LNBM to control a LNB (Layer Network Binding) the FCC uses or has established

i_NotifyCtrl: The FCC may require this interface to acquire notifications associated with trails it controls in the layer network, see Section

6.4.1.10 Components and Interfaces

6.4.1.10.1. Connection Coordinator Factory (CCF)

The CCF facilitates the working of the connectivity provider's domain through its ConSSetup interface by providing the capabilities to:

setup connectivity sessions by instantiation of connection controller objects;

manage all connectivity sessions in the domain by providing their names on request,

retrieve references to the ConSCtrl and ConSNotifyCtrl interfaces of a specific connectivity session.

Supported interfaces:

i_ConSSetup: This interface allows the establishment of connectivity sessions and their associated computation objects. It also allows some general connectivity level management operations. These operations are primarily included to aid management, such as recovery and transfer of connectivity session control.

Required interfaces:

i_ConSCtrl: to establish requested NFCs in a CC.

i_ConSNotifyCtrl:

6.4.1.10.2. Connection Coordinator(CC)

An instance of the object is created for each connectivity session. It offers the capabilities to:

control (setup,activate,deactivate,release) and manage (retrieve, session and flow control interface information) of flow connections within a connectivity session through its ConSCtrl interface

control (enable, disable) and management (update of recipient for the notifications) of notifications on the state and state changes of flow connections within a connection session through its ConSNotify interface

Supported interfaces:

i_ConSCtrl

i_ConSNotifyCtrl

Required interfaces: The following interface is suggested for initialization of the FCC.

i_FlowCtrl: Once the FCC is created, the CC can use this interface to initialize NFCs.

6.4.1.10.3. Flow Connection Controller (FCC)

The FCC object exists for each flow connection and offers the capabilities to:

control (addition, removal, modification, activation and deactivation) of one or more branches of a flow connection as well as management (retrieve information on the flow connection topology, traffic type, routing constraints,..) through its FlowConCtrl interface.

control (enable, disable, set) of the emission of notification through its FlowConNotifyCtrl interface,

generation of the notifications which are targeted towards a specific instance of the FlowConNotify interface in the connectivity user domain.

Supported interfaces:

i_FlowConnCtrl:

i_FCNotifyCtrl: to send notifications to the CSM regarding its associated
NFC;

i_NotifyCtrl:to acquire notifications associated to trails it controls in the layer network.

Required interfaces:

i_FlowConNotify: to send notifications to the CSM regarding its associated NFC;

i_TrailSetup: to request the LNC to establish a trail

i_TrailControl: on the LNC to control a trail the FCC uses or has established

i_LNBSetup: to request the LNBM to establish a Layer Network Binding

i_LNBControl: on the LNBM to control a LNB the FCC uses or has established

i_NotifyCtrl: acquire notifications associated to trails it controls in the layer network.

Operations on the i_ConSSetup interface:

```
/* Establish connectivity session. */
void setup_connectivity_session(
        in t_SecHandle handle,
        in t_SuccessCriterion criterion,
        in t_ConSessionName conSessionName,
        in t_FlowConnNotification notifIf,
        in t_ParameterListlist,
        in t_FlowConnDescSeq flowConnDescList,
        out i_ConnSessionNotificationControl notifCtrlIf,
        out t_FlowConnResponseSeq res)
        raises (NotAuthenticated, NotAuthorized,
        InvalidFlowConnInfo);
```

This operation allows to establish a connectivity session (create CC; if desired establish NFC(s) and return control interface for each NFC). It takes as input parameters: security info, success criterion, connectivity session name, connectivity session parameters (optional: if any overrides default values), NFC setup info (optional: if setup of NFC(s) is requested). The output parameters are success/failure, reference to i_conSCtrl, reference to i_ConSNotifyCtrl, list of tuples <NFC name, reference to i_FlowCtrl, reference to i_FCNotifyCtrl, FEPs that were bound, FEPs that could not be bound> for every setup NFC.

```
/* List connectivity session identifiers. */
void list_all_con_sessions(
        in t_SecHandle handle,
        out t_ConSessionNameList conSessionList)
        raises (NotAuthenticated, NotAuthorized);
```

This operation allows to list connectivity session identifiers and possible filtering. It takes as input parameters the security information and outputs the success/failure, list of names of connectivity sessions of this user.

```
/* Get a connectivity session. */
void get_con_session_control_interfaces(
```

```
in t_SecHandle handle,
in t_ConSessionName conSessionName,
out i_ConnSessionControl ctrlIf,
out i_ConnSessionNotificationControl notifCtrlIf)
raises (NotAuthenticated, NotAuthorized,
InvalidConnSessionName);
```

This operation allows to get a connectivity session (return reference to control interface given a connectivity session identifier). It takes in as input parameters the security information and Connectivity Session Name and outputs the success/failure,reference to i_conSCtrl,reference to i_ConSNotifyCtrl of the Connection Coordinator.

Operations on the i_ConSCtrl:

The i_ConnSessionControl provides operations to manipulate one or more NFCs in the connectivity session associated to the CC; for connectivity session query; and for acquisition
of a notification control interface associated with the connectivity session.

```
void setup_flow_connections(
        in t_SecHandle handle,
        in t_ParametersList list,
        in t_SuccessCriterion criterion,
        in t_FlowConnDescSeq flowConnDescList,
        out t_FlowConResponseSeq resp)
        raises (NotAuthenticated, NotAuthorized, InvalidFlowConnInfo);
```

This operation allows to set up one or more NFCs within the connectivity session (which includes creating the corresponding FCC component(s) for their manipulation) in active or inactive state; at least one branch of each requested NFC must be specified. It takes as input the security information, parameter list, success criterion, and NFC setup information. It then returns the success/failure,list of tuples <NFC name, reference to i_FlowCtrl, reference to FCNotifyCtrl, FEPs that were bound, FEPs that could not be bound>

```
/* Activate (one, more or all) NFCs in the connectivity session. */
        void activate_flow_connections(
        in t_SecHandle handle,
        in t_SuccessCriterion criterion,
            in boolean allFlag,
            in t_FlowConnNameSeq flowConnList,
        out t_ActivationResponseSeq resp)
        raises (NotAuthenticated, NotAuthorized, InvalidFlowConnName,
                ConnSessionActiveAlready);
```

This operation allows to activate one or more (or all) NFCs in the connectivity session; returns triplets <NFC, FEP, status> where FEP is an endpoint (root or leaf) of the NFC and status is “activated” or “unable to activate”. The input parameters are the security info,success criterion, boolean TRUE=all ConS must be activated, FALSE=only some specified NFCs, names of NFCs to be activated (only if previous parameter is FALSE).The return parameters are namely success/failure,list of triples <NFC name, FEP (root or leaf of NFC), status (activated/unable to).

```
/* Deactivate (one, more or all) NFCs of the connectivity session */
        void deactivate_flow_connections(
        in t_SecHandle handle,
           in t_SuccessCriterion criterion,
           in boolean allFlag,
           in t_FlowConnNameSeq flowConnList,
        out t_ActivationResponseSeq resp)
        raises (NotAuthenticated, NotAuthorized, InvalidFlowConnName,
        ConnSessionDeactiveAlready);
```

This operation allows to deactivate one or more (or all) NFCs of the connectivity session. It takes as input parameters the security information, the success criterion, boolean TRUE=all connectivity sessions must be deactivated, FALSE=only some specified NFCs, names of NFCs to be deactivated (only if previous parameter is FALSE). The return parameters are namely success/failure, list of triples <NFC name, FEP (root or leaf of NFC), status (deactivated/unable to).

```
/* Release (one, more or all) NFCs in the connectivity session. */
        void release_flow_connections(

        in t_SecHandle handle,
           in t_SuccessCriterion criterion,
           in boolean allFlag,
           in t_FlowConnNameSeq flowConnList)
        raises (NotAuthenticated, NotAuthorized, InvalidFlowConnName);
```

This operation allows to release one or more (or all) NFCs in the connectivity session (the corresponding FCCs delete themselves). If all then the connectivity session is released. It takes as input parameters the security info,success criterion,boolean TRUE=all connectivity sessions must be released, FALSE=only some specified NFCs,names of NFCs to be released (only if previous parameter is FALSE). The success or failure of the operation is then returned.The failure of the operation will result in an exception being raised either NotAuthenticated, NotAuthorized, InvalidFlowConnName)

```
/* Get information about the connectivity session. */
void get_conn_session_info(
        in t_SecHandle handle,
        out t_ConnSessionInfo info)
        raises(NotAuthenticated, NotAuthorized);
```

This operation allows to get information about the connectivity session: its state and a list of component NFC names.It takes the security information as the input parameters and returns the success/failure of the operation, connectivity session name, connectivity session's operational state, connectivity session's administrative state,connectivity session's profile
information, list of names of the NFCs that are components of the connectivity session. If the operation fails an exception is raised namely NotAuthenticated or NotAuthorized

```
/* Get references to interfaces i_FlowCtrl and i_FCNotifyCtrl for (one,
more or all) NFCs in the connectivity session. */
void get_conn_session_info(
        in SecHandle handle,
            in boolean allFlag,
            in t_FlowConnNameSeq flowConnList,
        out t_FlowConnInterfacesSeq connIfList)
        raises (NotAuthenticated, NotAuthorized, InvalidFlowConnName);}
```

This operation allows to get references to interfaces i_FlowCtrl and i_FCNotifyCtrl for one or more (or all) NFCs in the connectivity session*. It takes the security information, the flag Boolean TRUE=control interfaces for all NFCs are requested, FALSE=only for some specified NFCs,names of NFCs for which control interfaces are requested (only if previous parameter is FALSE).as the input parameter. The return parameters are namely the success/failure of the operation, the list of triples <NFC name, reference to i_FlowCtrl, reference to i_FCNotifyCtrl.

Operations on the i_ConSNotifyCtrl

The i_ConSNotifyCtrl provides operations for the control of notifications at the connectivity session level - joint or individual control of notifications associated to the connectivity session and its component NFCs, as desired.

```
* Enable notifications associated with one or more (or all) NFCs in
the connectivity session. */
void enable_Conn_Session_notification(
        in t_SecHandle handle,
            in boolean allFlag,
```

```
            in t_FlowConnNameSeq flowConnList)
        raises (NotAuthenticated, NotAuthorized,
        NotificationDestinationNotSet);
```

This operation enables notifications associated with one or more (or all) NFCs in the connectivity session. It takes as input parameters the security info, boolean TRUE=notifications for all NFCs are to be enabled, FALSE=only for some specified NFCs,names of NFCs for which notifications are to be enabled (only if previous parameter is FALSE).The success or failure of the operation is returned. The exception parameters raised in case of failure are either NotAuthenticated, NotAuthorized or NotificationDestinationNotSet.

```
/* Disable notifications associated with one or more (or all) NFCs in
the connectivity session. */
void disable_Conn_Session_notification(
        in t_SecHandle handle,
            in boolean allFlag,
            in t_FlowConnNameSeq flowConnList)
        raises (NotAuthenticated, NotAuthorized);
```

This operation allows to disable notifications associated with one or more (or all) NFCs in the connectivity session.It takes as input the security info,boolean TRUE=notifications for all NFCs are to be enabled, FALSE=only for some specified NFCs,names of NFCs for which notifications are to be enabled (only if previous parameter is FALSE). The success or failure of the operation is returned.

```
/* Set a CC interface as default destination of NFC notifications for
the connectivity session. */
void update_CS_flow_connection_notification_destination(
        in t_SecHandle handle,
            in i_FlowConnNotification destination)
        raises (NotAuthenticated, NotAuthorized);
```

This operation allows Set a CC interface as default destination of NFC notifications for the connectivity session. It takes as input parameters the security information, the reference to i_FCNotify and outputs the success or failure of the operation.

Operations on the i_FlowConnCtrl:

The i_FlowConnControl provides operations to manipulate one or more (or all) flow connection branches of the associated NFC, and a query operation to obtain information about it.

```
void add_flow_connection_branches(
        in t_SecHandle handle,
            in t_SuccessCriterion criterion,
            in t_FepDescSeq descList,
        out t_SuccFepList boundList,
            out t_FailedFepList unboundList)
        raises (NotAuthenticated, NotAuthorized,
        NonExistentFlowEndPoints, FlowEndPointsAlreadyBound,
        InvalidFlowConnBranchesInfo);
```

This operation allows to add one or more flow connection branches to the associated NFC, in an active or inactive state. It takes as input parameters the security info,success criterion, NFC branches setup info and returns the success/failure,list of leaf FEPs of branches successfully attached to the NFC, list of pairs <leaf FEP that could not be attached, failure code>. In case of failure the exception raised are either due to non existent flow end points, the flow end points are already bound or the flow connection branches information are invalid.

```
/* Delete one or more (or all) flow connection branches from the
associated NFC. */
void delete_flow_connection_branches(
        in t_SecHandle handle,
                in t_SuccessCriterion criterion,
            in boolean allFlag,
             in t_FepRefList list,
        out t_SuccFepList list1,
            out t_FailedFepList list2)
        raises (NotAuthenticated, NotAuthorized,
        NonExistentFlowEndPoints);
```

This operation allows to delete one or more (or all) flow connection branches from the associated NFC; if all then release the NFC and self-delete FCC. The input parameters are security info,success criterion,boolean TRUE=the entire NFC is to be deleted, FALSE=only some specified branches, names of leaf FEPs of branches to be deleted (only if previous parameter is FALSE). The success/failure, list of leaf FEPs of branches successfully deleted from the NFC,list of pairs <leaf FEP that could not be deleted, failure code> is then returned.If failure is returned the probable cause are either due to not authorized, not authenticated or the Network Flow End Points are non existent.

```
/* Activate one or more (or all) flow connection branches of the
associated NFC. */
void activate_flow_connection_branches(
            in t_SecHandle handle,
                   in t_SuccessCriterion criterion,
                 in boolean allFlag,
                  in t_FepRefList list,
            out t_SuccFepList activatedList,
                 out t_FailedFepList activationFailedList)
            raises (NotAuthenticated, NotAuthorized,
            NonExistentFlowEndPoints,
            FlowConnBranchesActiveAlready);
```

This operation allows to activate one or more (or all) flow connection branches of the associated NFC. It takes as input parameters the security info,success criterion,boolean TRUE=the entire NFC is to be activated, FALSE=only some specified branches,names of leaf FEPs of branches to be activated (only if previous parameter is FALSE).The success/ failure,list of leaf FEPs of branches successfully activated from the NFC,list of pairs <leaf FEP that could not be activated, failure code>. In case of failure the exception raised are either due to non existent flow end points, or the flow connection branches are already active.

```
/* Deactivate one or more (or all) flow connection branches of the
associated NFC. */
void deactivate_flow_connection_branches(
            in t_SecHandle handle,
                   in t_SuccessCriterion criterion,
                 in boolean allFlag,
                  in t_FepRefList list,
            out t_SuccFepList deactivatedList,
                 out t_FailedFepList deactivationFaliedList)
            raises (NotAuthenticated, NotAuthorized,
            NonExistentFlowEndPoints,
            FlowConnBranchesDeactiveAlready);
```

This operation allows the deactivation of one or more (or all) flow connection branches of the associated NFC. The input parameter are security info,success criterion,boolean TRUE=the entire NFC is to be deactivated, FALSE=only some specified branches,names of leaf FEPs of branches to be deactivated (only if previous parameter is FALSE). The output parameters are the success/failure,list of leaf FEPs of branches successfully deactivated from the NFC,list of pairs <leaf FEP that could not be deactivated, failure code>. In case of failure the exception raised are either due to non existent flow end points, or the flow connection branches are already deactived.

```
/* Modify one or more (or all) flow connection branches of the
associated NFC. */

void modify_flow_connection_branches(
        in t_SecHandle handle,
            in t_SuccessCriterion criterion,
            in t_FepDescSeq descList,
        out t_SuccFepList modifiedList,
            out t_FailedFepList unmodifiedList)
        raises(NotAuthenticated, NotAuthorized,
        NonExistentFlowEndPoints, InvalidFlowConnBranchesInfo);
```

This operation allows the modify one or more (or all) flow connection branches of the associated NFC. The input parameters are the security info,success criterion,NFC branches modification info.The success/failure,list of leaf FEPs of branches successfully modified, list of pairs <leaf FEP that could not be modified, failure code> is then returned.

.

```
/* Get information about the associated NFC. */
void get_flow_conn_info(
        in t_SecHandle handle,
        out t_FlowConnInfo connInfo)
        raises (NotAuthenticated, NotAuthorized);
        }
```

This operation allows to get information about the associated NFC. It takes as input the security information and outputs the success/failure, connection topology, traffic type, routing constraints, NFC notification destination, reliability class, for each FPE a tuple <FPE name, FPE type (root/leaf), peak bandwidth, average bandwidth, maximum delay, maximum de lay variation, administrative state, operational state>.

Operations on the i_FCNotifyCtrl

The i_FCNotifyCtrl provides operations for controlling the emission of notifications about the associated NFC.

```
/* Enable notifications regarding the associated NFC. */
void enable_flow_connection_notification(
        in t_SecHandle handle)
        raises (NotAuthenticated, NotAuthorized,
        NotificationDestinationNotSet);
```

```
/* Disable notifications regarding the associated NFC. */
void disable_flow_connection_notification(
        in t_SecHandle handle)
        raises (NotAuthenticated, NotAuthorized);

/* Set an FCC interface as default destination of notifications for the
associated NFC. */
void set_flow_connection_notification_destination(
        in t_SecHandle handle,
            in i_FlowConnNotification destination)
        raises (NotAuthenticated, NotAuthorized);
        }
```

Operations on the i_FCNotifyCtrl

```
/* Enable notifications regarding the associated NFC. */
void enable_flow_connection_notification(
        in t_SecHandle handle)
        raises (NotAuthenticated, NotAuthorized,
        NotificationDestinationNotSet);

/* Disable notifications regarding the associated NFC. */
void disable_flow_connection_notification(
        in t_SecHandle handle)
        raises (NotAuthenticated, NotAuthorized);

/* Set an FCC interface as default destination of notifications for the
associated NFC. */
void set_flow_connection_notification_destination(
        in t_SecHandle handle,
            in i_FlowConnNotification destination)
        raises (NotAuthenticated, NotAuthorized);

}
```

7. Layer Network Related Components

The concept of Layer Network (LNW) is used to denote a network that is based on a single technology and that transports information of a specific format, referred to as the characteristic information. Examples of layer networks are: ATM Virtual Path (VP) network, ATM Virtual Channel (VC) network, SDH VC4 Path network, Frame Relay (FR) network, Internet, Ethernet, and Token Ring Network. The functions provided at this level are the setup and management of Layer Network Bindings (LNBs). The LNB is the generalized concept of trail and represents an end-to-end connectivity across a layer network irrespective of its connection type (connection-oriented or connectionless layer network). Figure 8-1 shows the computational objects and interfaces associated with the layer network.

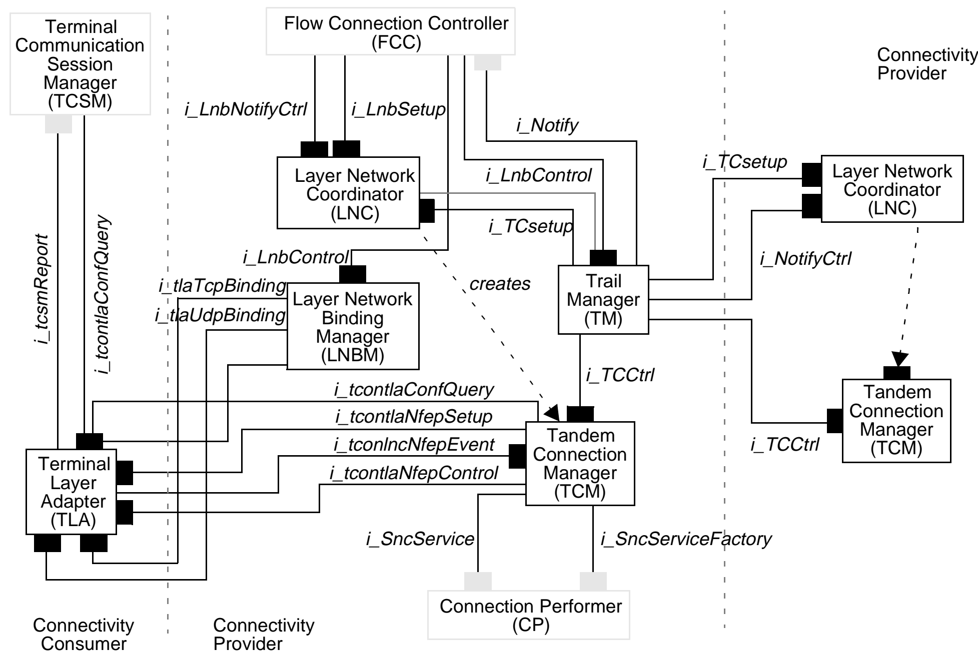


Figure 7-1. Layer Network Related Objects

The information objects that describe the network fragment in a layer network are the following:

Layer Network Domain (LND)

Represents the part of a layer network that is under the control of one administrative domain. A layer network domain consists of a top level subnetwork and a set of links.

Local Layer Network Domain (LLND)

Represents the part of a layer network that is under the control of the local administrative domain. LLND is the subtype of LND.

Foreign Layer Network Domain (FLND)

Represents from the perspective of a connectivity provider the part of a layer network that is under the control of a foreign network administration.

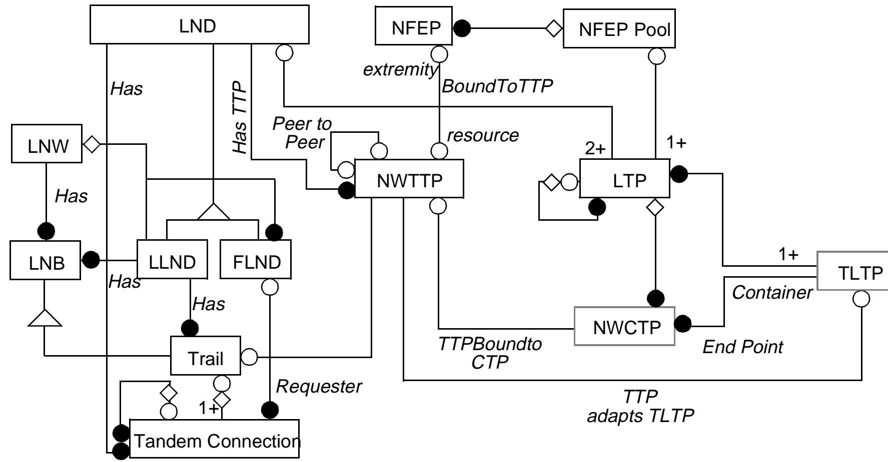


Figure 7-2. Layer network related information model fragment.

The information objects that describe the connectivity in a layer network are the following. These objects represents the connectivity view as perceived by one connectivity provider and does not represent a global view of a TINA network.

Trail (T) Represents the resource that transfers information between two or more end points of a layer network. The end points of a trail are called Network Trail Termination Points (NWTTPs).

Layer Network Binding (LNB)

Represents an association between two or more end points in a layer network and the generalized concept of trail.

Tandem Connection (TC)

Represents a portion of a trail that either exists in a local layer network domain or spans one or more foreign layer network domains. The end points of a tandem connection are NWTTPs or NWCTPs. From the perspective of a LND, a tandem connection consists of a subnetwork connection across the top level subnetwork of the local LND, zero or more tandem connections in one or more foreign LNDs.

Each layer network of a connectivity layer network represents a set of compatible inputs and outputs that may be characterized by the characteristic information. These end points represent access points to layer networks and modelled as end points but are specialized according to their types.

Network Trail Termination Point (NWTTP)

Represents an end point of a trail in a layer network. The NWTTP models the point where a layer network (typically an adaptation card in a CPE) receives information from a client layer in a format that it support or delivers information to the client layer. However, only technology independent aspects are represented in this object type. Technology specific trail terminations (e.g., ATM VP layer TTP) should be derived from this type to represent additional information.

Layer Network Access Point (LNAP)

Represents an end point of a LNB in a layer network. In the case of internet, a LNAP represents either a UDP port or a TCP port depending on the transport protocol used for the stream flow.

7.1 Layer Network Coordinator (LNC)

The Layer Network Coordinator (LNC) object acts on behalf of a Layer Network Domain (LND) and provides LNB setup, manipulation, and release operations to the FCC objects in the same connectivity provider domain. One instance of this object type exists in a connectivity provider domain for each LND contained in the connectivity provider domain. Furthermore, the LNC provides Tandem Connection (TC) setup, manipulation and release operations to the LNC of each neighbor foreign LND, and also requests them to perform tandem connection setup and release operations. Thus, an LNC has a peer-to-peer relationship (both client and server roles) with each neighbor LNC. However, LNCs in different layer networks do not communicate directly. The Connection Coordinator (CC) object is responsible for this interworking.

The LNC is a single access point to a layer network domain and supports queries on the status of the layer network and its associated LNBs. The LNC also supports queries about tandem connections established within a layer network domain.

When it receives a LNB setup request, the LNC creates a TM or a LNBM object for the management of the new LNB and passes the LNB setup request to the TM or LNBM. The selection between TM or LNBM is depend on the associated layer network capabilities. To set-up a LNB which spans across different management domains, an LNC federates with other domains and interacts in terms of tandem connections. When it receives a tandem connection setup request, the LNC creates a TCM object for the manipulation of the new tandem connection and passes the tandem connection management request to the TCM.

The LNC object provides i_LnbSetup and i_LnbNotifyControl interfaces, the client for these two interfaces is the FCC object in the same connectivity provider domain. The LNC provides also i_TCSetup interface for LNC objects in federated domains.

7.1.1 Interface Description

7.1.1.1 i_LnbSetup interface

i_LnbSetup interface allows the establishment of LNBs and their associated computation objects. It also allows query and LNB release operations. These operations are primarily included to aid management, such as recovery and transfer of LNB control.

- setup_LNBs()
  - This operation is used to set up one or more LNBs. If successful, this operation creates a Trail Manager object or a Layer Network Binding Manager object (if necessary) and returns a LNB control interface for each LNB established.
- release_LNBs()
  - This operation is used to release one or more specified LNBs within the LNC. It can only be used to release LNBs established in the domain associated with the LNC. This operation is included for management purposes.
- get_LNB_ctrl_interfaces()
  - This operation is used for obtaining the references to specified LNB control interfaces.
- get_LNB_notification_ctrl_interface()
  - Return the notification control interface reference associated with this LNC. This interface allows a client to modify notifications, including destinations and notification types of interest.

7.1.1.2 i_TCSetup interface

i_TCSetup interface allows the establishment of tandem connections and their associated computational objects. It also allows query and tandem connection release operations. These operations are primarily included to aid management, such as recovery and transfer of tandem connection control.

- setup_tc()
  - This operation is used to set up a tandem connection. If successful, this operation creates a Tandem Connection Manager object and returns a tandem connection control interface for each tandem connection established.
- release_tc()
  - This operation is used to release one or more specified tandem connections within the LNC. It can only be used to release tandem connections established in the domain associated with the LNC.
- get_tc_info()
  - This operation is used for retrieving lists of tandem connections in the layer network.
- get_tc_ctrl_interfaces()

    - This operation is used for obtaining the references to specified tandem connection control interfaces.

- get_tc_notification_ctrl_interface()
    - Return the notification control interface reference associated with this LNC. This interface allows a client to modify notifications, including destinations and notification types of interest.

7.2 Trail Manager (TM)

A Trail Manager (TM) is created by the Layer Network Coordinator of the layer network domain in which the trail is initially established. A trail is a specialization of LNB for connection-oriented networks. Therefore, one instance of this object type exists for each trail in the originating connectivity provider domain. The trail manager acts as a single point of contact for control of the trail, even though it extends over multiple domains. The FCC is the usual client of the trail manager.

When the LNC receives a LNB setup request, the LNC creates a TM or LNBM depending on whether it manages a connection-oriented or (exclusive) connectionless layer network. The TM created then manages the specific trail. To provide transparency to FCC, both LNBM and TM support an identical interface, called the LNB (i_LnbControl) Interface. After creates a TM, the LNC returns to the FCC a reference to the LNB interface of the TM.

When a TM is requested to setup a trail, it determines the layer network domains that are to be traversed by the trail. This determination is based on the layer network domains to which the LTPs of the root and leaf NFEPs belong, the topology of the layer network, and the route selection policy of the TM[1]. Based on this determination, the TM decomposes the trail into two components: a tandem connection in the local LND, and zero or more foreign tandem connections. More than one foreign tandem connection may need to be setup in the case of a point-to-multipoint trail.

To setup the tandem connection in the local LND, the TM requests the LNC of the local LND through i_TCSetup interface. To setup a foreign tandem connection, the TM requests the LNC of the neighbor foreign LND associated with the tandem connection. Therefore, the TM should keep the reference to the subnetwork connection associated with a trail in its domain, and the tandem connection with the neighbor domains. However, an LNC will return the interface to a single trail manager even if the trail requires federation between multiple domains of the layer network to be established.

The TM maps other trail management operations into one or more tandem connection management operations depending on the trail branches affected by the operation, and requests the corresponding TCMs to perform the management operation.

The TM object provides i_LnbControl interface, the client for this interface is the FCC object in the same connectivity provider domain. The LNC provides also i_TCSetup interface for LNC objects in federated domains.

1. Source Routing Acceptance is a possibility

7.2.1 Interface Description

7.2.1.1 i_LnbControl Interface

Through this interface the FCC object can manipulate a trail by adding, modifying, activating, deactivating or releasing LNB branches. An operation can act on the entire trail by specifying all branches. A trail and its branches may be specified in terms of Network Trail Termination Points or NWTTP Pools.

- add_LNB_branches()
  - This operation is used for adding one or more branches to the trail associated with the Trail Manager object. Branches can be added in an active or inactive state.
- delete_LNB_branches()
  - This operation is used for removing one or more branches (possibly all branches) from the trail associated with the Trail Manager object.
- activate_LNB_branches()
  - This operation is used for activating one or more branches (possibly all branches) of the trail associated with the Trail Manager object.
- deactivate_LNB_branches()
  - This operation is used for deactivating one or more branches (possibly all branches) of the trail associated with the Trail Manager object.
- modify_LNB_branches()
  - This operation is used for modifying the layer network binding information of one or more branches associated with the Trail Manager object.
- get_LNB_info()
  - This operation is used for retrieving the layer network binding information. It returns information on routing constraints, traffic type, QoS and termination points. Filtering attributes may be applied to manage which LNBs are listed.

7.3 Terminal Layer Adapter (TLA)

The Terminal Layer Adapter (TLA) is the counterpart in the connectivity consumer domain of the LNC in the connectivity provider domain. The TLA object provides functions for setting up (creating), manipulating and deleting network flow end points. TLA performs these operation upon requests from a TCM or LNBM in the associated LND. These operations involve bandwidth management, port allocation and channel assignment of access links that connect the CPE to adjoining NEs (i.e., switches or cross connectors).

One instance of this object type exists in a CPE for layer network to which the CPE is attached. TLA also serves as the linkage point between the technology independent aspects and technology specific aspects of a stream flow connection setup. After it sets up a NWTTP for a trail associated with a stream flow connection, the TLA notifies the TCSM that a

NFEP which is a generic view of the NWTTP has been setup. Upon receiving this notification, the TCSM sets up the terminal flow connection that binds the SFEP to the chosen NFEP.

7.3.1 Interface Description

7.3.1.1 i_tcontlaNfepSetup Interface

This interface is used for setting up a NFEP in the terminal. NFEPs may be set up dynamically during connection establishment or may be pre-provisioned. It provides the following operations:

- setup_nfep()
  - This operation is used to set-up the NFEP in the connectivity consumer domain. The connectivity consumer will request the operation and report success or failure. When the operation is performed successfully, the NFEP will exist, be bound to a NWTTP and its collocated NWCTP, and be in the *unlocked* state or the *locked* state, depending on the value of the parameter “Initial Administrative State”.
- release_nfep()
  - This operation is used to release a NFEP in the connectivity consumer domain. The connectivity consumer will execute the operation and report success or failure. This operation may be invoked in the *locked* state or in the *unlocked* state. After successful execution of the operation, the NFEP will not be bound to the supporting NWTTP anymore, and the supporting NWTTP will be deleted. If not pre-provisioned, the supporting NWCTP is deleted as well. Similar, the NFEP is deleted if it is not pre-provisioned.

7.3.1.2 i_tcontlaNfepControl Interface

- activate_nfep()
  - This operation is used to activate a NFEP in the connectivity consumer domain. The connectivity consumer will execute the operation and report success or failure.
- deactivate_nfep()
  - This operation is used to de-activate an active NFEP in the connectivity consumer domain. The connectivity consumer will execute the operation and report success or failure.
- modify_nfep_qos()
  - This operation is used to modify the characteristic of the NFEP (e.g. bandwidth, QoS, bearer service type, etc.). The connectivity consumer will execute the operation and report success of failure. This operation may be invoked in the *locked* state or the *unlocked* state. It will not result in a state change.

7.3.1.3 i_tcontlaConfQuery Interface

This interface is provided both to TCSM or TCSM in order to query the TLA about the available NFEP Pools and their status.

- get_nfep_pools()
  - This operation is used for retrieving the names of all flow endpoint pools that the connectivity consumer domain supports.

7.3.1.4 i_tcontlaEventCtrl Interface

It is provided from TLA to LNC in order to be notified (the TLA) about error and failures at the network that affect the terminal connec-tion.

- enable_tla_event()
  - This operation instructs the connectivity consumer (TLA) to emit notifications regarding the NFEPs.
- disable_tla_event()
  - This operation instructs the connectivity consumer (TLA) to suspend emission of notifications regarding the NFEPs.
- set_tla_event_destination()
  - This operation is used to instruct the TLA about the i_tconlncNfepEvent interface to which notifications should be sent.

7.3.1.5 i_tlaTcpBinding Interface

Even though TCP and UDP binding use the same network layer (IP), TCP binding provides a totally different service to the above layer than UDP binding does. Fundamentally, TCP binding offers a reliable, full duplex, sequenced, and unduplicated byte stream transport between two TCP endpoints. Through the TCP flow and traffic control mechanism, TCP binding provides the regulated stream flows between TCP endpoints. TCP binding operates very naturally in a client/server environment. A server TLA listens for incoming connection requests and a client TLA initiates TCP binding by invoking communication routines that establish a connection with a server. The TLA for the internet layer network offers an i_tlaTcpBinding interface that has the following operations and Figure 12-3 shows the interactions between the LNBM and the TLAs for the setup of a point-to-point LNB using TCP.

- create_Tcpep()
  - This operation takes an optional (IP) end point description as input. This description identifies the set of IP addresses (network interfaces) from which the TLA can select one for creating an end point for the LNB. If the IP end point description is omitted, the TLA selects one of the IP addresses associated with the CPE. After creating[2] the end-point it returns the fully resolved end point description[3] (e.g. adds selected port).
- accept_Tcp()

- This operation optionally a local and a remote end-point descriptions as input. It creates the local end point if it does not exist yet. The local end point will accept TCP connection requests only from the remote end point if it is specified. If no remote end point is specified it will accept connection requests from anywhere. It returns the fully resolved local end point description.

- connect_Tcp()
  - This operation takes an optional local and a remote end point description as input. If the local end point does not exist, it creates one. It connects to the remote end-point and returns the resolved local end-point.
- disconnect_Tcp()
  - This operation takes a local end point description as parameter and disconnects the connection if it exists.
- delete_Tcpep()
  - This operation takes a local end point description as parameter and deletes it and the associated connection, if any.

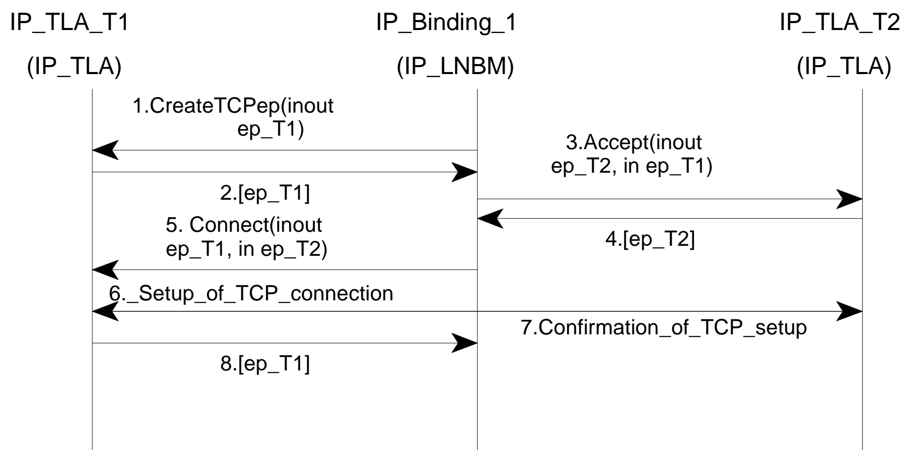


Figure 7-3. LNBM-TLA Interactions for a TCP/IP LNB

2. All operations that can create end-points must also have a correlation identifier as ‘in’ parameter. This is a reference to the terminal part of the stream flow connection.
3. The specification of the end-point description is out of the scope of this paper, but it must be able to contain full or partially resolved references to IP end-points. A URL like approach might be suitable (e.g. TCP://ip_address:port).

7.3.1.6 i_tlaUdpBinding Interface

This interface for the UDP transport supports both point-to-point and multipoint bindings. The TLA interface for UDP provides the following operations which can be applied consistently to both point-to-point and multipoint bindings. In spite of some limitations, i_tlaUdpBinding interface has the advantage over i_tlaUdpBinding interface in the following areas: speed, overhead, and simplicity. Because TCP binding, to provides reliable transport, exchanges huge number of packets over the network. UDP binding doesn't have this overhead, and is considerably faster than TCP binding. In consequence, in those situation where speed is important, or the number of packets sent over network must be kept to a minimum such as network management services, UDP binding is the solution. Figure 12-4 illustrates the LNBM-TLA interactions needed for the setup and release of a point-to-point LNB using UDP. A LNBM that manages a IP multicast address can easily use this for a point-to-multipoint UDP LNB as demonstrated by Figure 12-5. Figure 12-5 illustrates the LNBM-TLA interactions needed for the setup of a point-to-multipoint LNB using IP multicasting.

- create_Udpep()
  - This operation has the same semantics as CreateTCPep, the only difference is that the end point is a UDP end point.
- add_Udp_source()
  - This operation takes an optional local and a remote end point description as input. If the local end point does not exist, it creates one. Then, it adds the remote end-point to the list of accepted sources for the local end point, starts a receiving process if this is the first receiving end point and returns the fully resolved local end point description
- add_Udp_destination()
  - This operation takes an optional local and a remote end point description as input. If the local end point does not exist, it creates one. Then, it adds the remote end point to the list of destinations for the local end point and returns the fully resolved local end point description.
- remove_Udp_source()
  - This operation takes a local and a remote end point description as input. It removes the remote end point from the list of accepted sources for the local end point and stops the receiving process if this is the last receiving end point.
- remove_Udp_destination()
  - This operation takes a local and a remote end point description as input. It removes the remote end point from the list of destinations for the local end point.
- delete_Udpep()

- This operation takes an end point description as input. It stops the receiving process if it is running and deletes the end point.

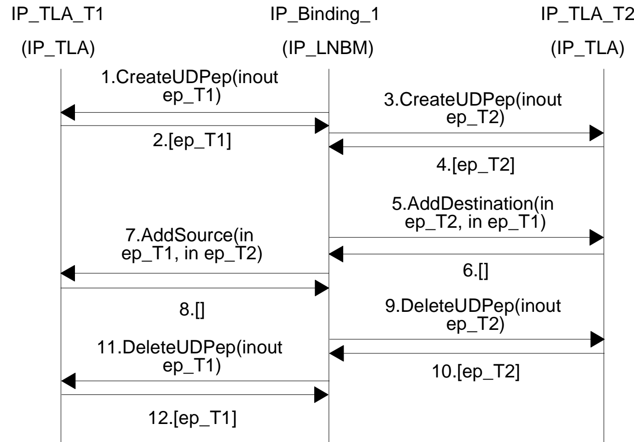


Figure 7-4. LNBM-TLA Interactions for a UDP LNB

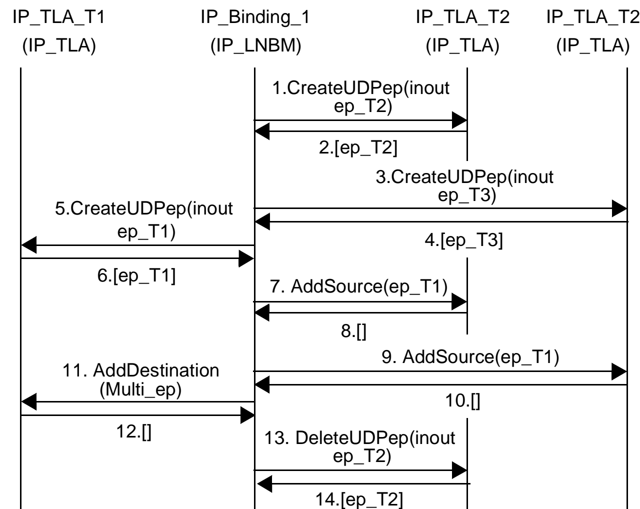


Figure 7-5. LNBM-TLA Interactions for a point-to-multipoint UDP LNB

7.4 Tandem Connection Manger (TCM)

A Tandem Connection Manager (TCM) is created by the Layer Network Coordinator of the same layer network domain. One instance of this object type exists for each tandem connection originating in the connectivity provider domain. A tandem connection manager object provides operations for manipulating the associated tandem connections, including the

tandem connection release operation. Tandem connection managers are responsible for the routing of the tandem connection through the layer network domain. When a tandem connection is released, the associated tandem connection manager object is deleted.

A tandem connection may be terminated by a connection termination point or a trail termination point. To support trail termination, a tandem connection manager can interact with a TLA to manipulate the endpoints of the network flow connection associated with the tandem connection. The tandem connection manager manipulates NFEPs through the TLA's i_tcontlaNfepControl interface.

When a TCM is requested to setup a tandem connection, it determines the layer network domains that are to be traversed by the tandem connection. This determination is based on the layer network domains to which the LTPs associated with the end points of the tandem connection belong, the topology of the layer network, and the route selection policy of the TCM. Based on this determination, the TCM decomposes the tandem connection into two components: a tandem connection in the local LND, and zero or more foreign tandem connections. More than one foreign tandem connection may need to be setup in the case of a multipoint trail. To setup a foreign tandem connection, the TCM requests the LNC of the neighbor foreign LND associated with the tandem connection. Before it sets up the tandem connection in the local LND, the TCM determines if the local tandem connection is a boundary tandem connection, i.e., one of the end points is on a CPE. if so, it requests the Terminal Layer Adapter object associated with the CPE to setup the NWCTP and NWTTP for the trail end point on the CPE. This request causes the TLA to assign bandwidth and channel number (such as VPI/VCI in the case of ATM) on the associated access link. Upon receiving a success response from the TLA, the TCM proceeds to setup the local tandem connection.

To setup the local tandem connection, the TCM selects the route in terms of (second level) subnetworks and links in the local LND. It assigns bandwidth and channel number on each link in the chosen route and requests the Network Management Layer Connection Performer (NML-CP) associated with each subnetwork in the chosen route to setup the associated subnetwork connection. Similarly, the TCM maps other tandem connection management operations into operations on foreign tandem connections, if necessary, and subnetwork connections in the local LND. Then, it requests the TCMs in the foreign domains and/or NML-CPs in the local domain to perform the corresponding management operation.

7.4.1 Interface Description

7.4.1.1 i_TCCtrl Interface

Through this interface, a client can manipulate a tandem connection by adding, modifying activating or releasing branches. An operation can act on the entire tandem connection by specifying all branches. A tandem connection and its branches may be specified in terms of NWTTPs, NWCTPs, NWTTP Pools or NWCTP Pools.

- activate_tc_branches()
  - This operation is used for activating one or more branches (possibly all branches) of the tandem connection associated with the Tandem Connection Manager object.

- deactivate_tc_branches()
  - This operation is used for deactivating one or more branches (possibly all branches) of the tandem connection associated with the Tandem Connection Manager object.
- add_tc_branches()
  - This operation is used for adding one or more branches to the tandem connection associated with the Tandem Connection Manager object. Branches can be added in an active or inactive state.
- delete_tc_branches()
  - This operation is used for removing one or more branches (possibly all branches) from the tandem connection associated with the Tandem Connection Manager object.
- modify_tc_branches()
  - This operation is used for modifying one or more branches to the trail associated with the Tandem Connection Manager object.
- get_tc_info()
  - This operation is used for retrieving the tandem connection information. It returns information on routing constraints, traffic type, QoS and termination points of the tandem connection.

7.4.1.2 i_tconIncNfepEvent Interface

Through this interface the TLA can notify the Tandem Connection Manager (or other layer network manager, e.g. Trail Manager) of changes in the operational state of a NFEP.

- nfep_status_change ()
  - This operation is used to notify the connectivity provider when the operational state of a NFEP changes.

7.5 Layer Network Binding Manager (LNBM)

A Layer Network Binding Manager (LNBM) is created by the Layer Network Coordinator of the layer network domain. The LNBM is the counterpart of the Trail Manager that is used for managing a trail in a connection oriented network. To provide transparency to FCC, both LNBM and TM support an identical interface. When a LNC receives a LNB setup request, the LNC creates a LNBM if it manages a connectionless layer network. In the case of Internet (connectionless LNW), a LNAP represents either a UDP port or a TCP port depending on the internet transport layer protocol used for the stream flow. After it creates a LNBM as a result of a SetupLNB, the LNC returns to the FCC a reference to the LNB interface of the LNBM.

From a resource modeling perspective, a LAB has the following attributes: characteristic information (such as ATM VP, ATM VC, Frame Relay, UDP, TCP, etc.); topology (point-to-point, point-to-multipoint, etc.), QoS parameters that can include bandwidth and timing. A trail is then a specialization of LNB for connection-oriented networks.

Federation: LNBMs may interact to establish a LNB across multiple layer network domains. The role of LNBMs in federation and interactions between them and LNC will be elaborated in future releases of the NCS.

7.5.0.1 i_LnbControl Interface

Through this interface the FCC object can manipulate a trail by adding, modifying, activating, deactivating or releasing LNB branches. An operation can act on the entire trail by specifying all branches. A layer network binding and its branches may be specified in terms of Network Trail Termination Points or NWTTP Pools.

- add_LNB_branches()
  - This operation is used for adding one or more branches to the LNB associated with the Layer Network Binding Manager object. Branches can be added in an active or inactive state.
- delete_LNB_branches()
  - This operation is used for removing one or more branches (possibly all branches) from the LNB associated with the Layer Network Binding Manager object.
- activate_LNB_branches()
  - This operation is used for activating one or more branches (possibly all branches) of the trail associated with the Layer Network Binding Manager object.
- deactivate_LNB_branches()
  - This operation is used for deactivating one or more branches (possibly all branches) of the trail associated with the Layer Network Binding Manager object.
- modify_LNB_branches()
  - This operation is used for modifying one or more branches of the LNB associated with the Layer Network Binding Manager object.
- get_LNB_info()
  - This operation is used for retrieving the layer network binding information. It returns information on routing constraints, traffic type, QoS and termination points. Filtering attributes may be applied to manage which LNBs are listed.

8. Subnetwork Related Components

8.1 Connection Performer

A Connection Performer (or CP for short) is a computational entity which manage a subnetwork and is responsible for the provision of interconnections between the termination points of subnetworks. In addition, it offers services to establish, modify and release subnetwork connections in the subnetwork. These services of the CP are provided by means of the operations for controlling NRIM objects[1] such as SubNetworkConnections and Edges, which represent the termination points of a SubNetworkConnection. The CP may manages one or more subnetworks but cannot manage the subnetworks in other administrative domains. In addition, the subnetworks will not span resources in more than one administrative domain.

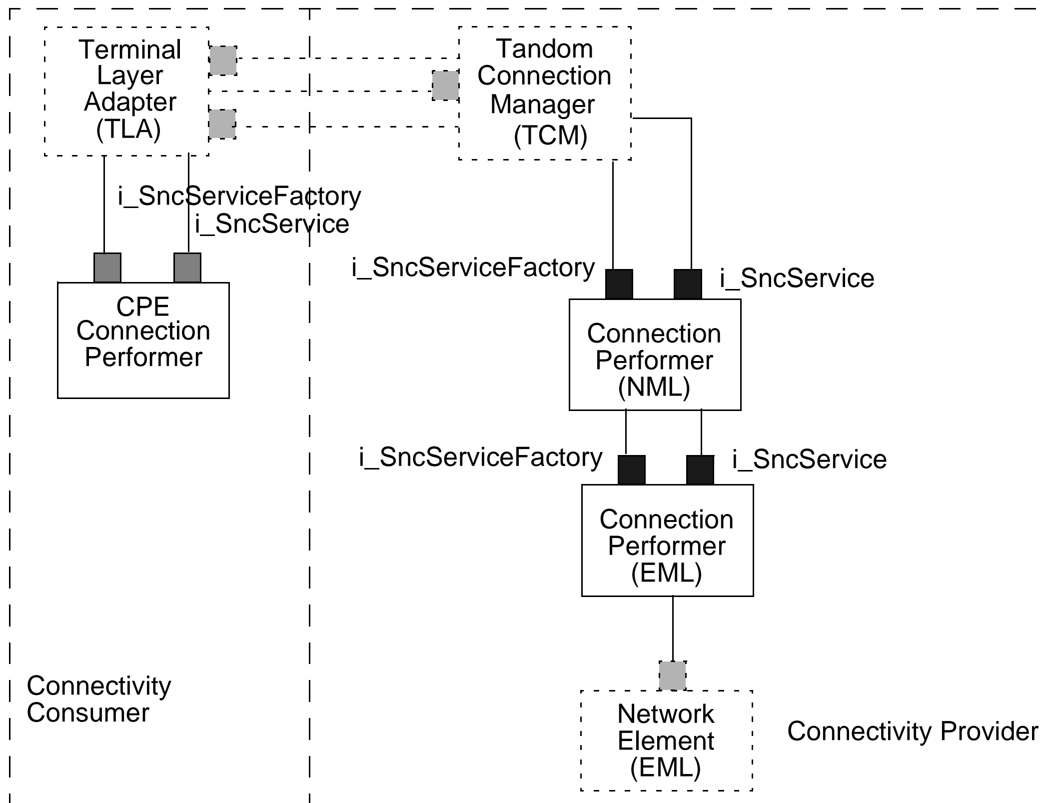


Figure 8-1. Subnetwork control related objects

1. NRIM object is an information object which defined in Network Resource Information Model.

The CPs may establish hierarchical relations with each other, reflecting the hierarchical relations between subnetworks. Like the partitioning and aggregation concept in the information model, a CP at lower-level provides the subnetwork view for higher level CP, hiding the unnecessary details under the subnetwork. In other words, the CP may manage the lower-level subnetworks without considering further operations to manage much lower-levels subnetworks.

Based on the types of TMN Layer Management, the CP can be classified as either Network Management Layer CP (or NML-CP for short) or Element Management Layer CP (or EML-CP for short).

- NML-CP manages connections in a partioned subnetwork of a layer network domain.
- EML-CP manages connections in the subnetwork which breaks down to the Network Element Layer, and have access to an agent in a Network Element.

Since logical (or technology independent) subnetworks in both CPs can be common, this document provides generic specificating of the CP. Further specialization of the CPs is beyond the scope of this document and will be done considering particular transport technologies, e.g., ISDN, ATM, and SDH.

8.2 Related Information Model Fragment

Since most of management related to the CP will be done by means of controlling NRIM objects, the CP should includes necessary NRIM objects and provide interfaces to control NRIM objects. This section provides brief description of NRIM objects contained in the CP. The further detail specification of each NRIM object is outside the scope of this document and can be found in NRIM.

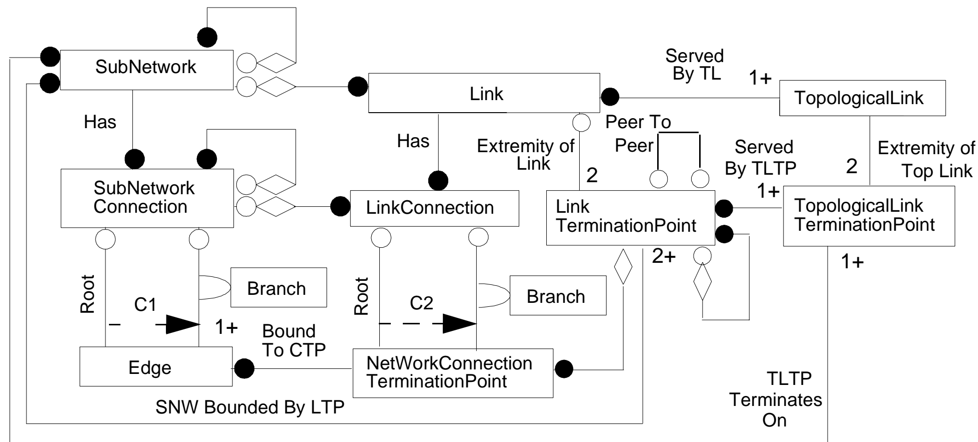


Figure 8-2. CP Related Information Model Fragment

The constraints C1 and C2 labelled in the above OMT diagram are described below:

- C1: An Edge object participates in exactly one of the two relationships
- C2: A NWCTP object participates in exactly one of the two relationships

With regard to the connection management, the manipulations of SubNetworkConnection and Edge are the most important aspect. The characteristics of these NRIM objects are described as follows:

- SubNetworkConnection(SNC): The resource that transfers information across a subnetwork. The end points of a subnetworks connection are called Edges. The connection topology may be point-to-point bidirectional, point-to-point unidirectional, or point-to-multipoint unidirectional.
- Edge: The resource that represents an end point of a subnetwork connection. Each Edge is bound to a NWCTP.

While the connection management, these NRIM objects are created, modified, and deleted to control communications in a subnetwork. For example, the SubNetworkConnection and is created in line with the following manner:

Pre condition: A Edge and SubNetwork have already been created and are ready for management. The Edge is bound to NWCTP and related to the SubNetwork. In addition, such information, e.g., existence and states of NRIM objects, have been known by a managing entity.

(1) CreateSNC: This action causes the creation of a new SubNetworkConnection in line with specified attributes and the establishment of Root relationship between the SubNetworkConnection with and the Edge.

Post condition: A SubNetworkConnection has been created, and two Edges have been created and attached to the SubNetworkConnection. Each Edge is bound to a NWCTP. A Edge is related to SubNetworkConnection with Root relationship; the other Edge, with Branch relationships.

Also, the SubNetworkConnection can be modified as follows:

Pre condition: A SubNetworkConnection has been created, and $N(N \geq 1)$ Edges have been created and attached to the SubNetworkConnection. Each Edge is bound to a NWCTP. A Edge is related to SubNetworkConnection with Root relationship; the other $N-1$ Edges, with Branch relationships. These information and the states of these NRIM objects are known by a managing entity.

(1) CreateEdge: The results of this action is the creation of a new Edge bound to a specified NWCTP.

(2) AttachEdge: This action causes attaching a created Edge to a specified SubnetworkConnection. The Edge is related to the SubNetworkConnection with Branch relationship.

Post condition: A Edge has been newly created, and $N+1$ Edges have been attached to the SubNetworkConnection. Each Edge is bound to a NWCTP. A Edge is related to SubNetworkConnection with Root relationship; the other N Edge, with Branch relationships.

For the configuration management, the other NRIM objects need to be considered. The details of these information aspects can be found in NRIM.

8.3 Mapping of Information Object to CP

The NRIM objects explained in Section 8.2 can be mapped into either computational interfaces or internal implementation[2]. Since the manipulation of SubNetworkConnection is a major aspect of the CP, direct mapping of a SubNetwork and SubNetworkConnection to interfaces is required for achieving the connection management smoothly. Moreover, the SubNetwork is the top level information entity in the CP, so it should be visible in a computational view to adapt information hierarchy to the structure constructed by CPs at various levels. The other NRIM objects are mapped into the internal implementation.

8.4 Computational structure

From the network management point of view, each management operation to network resources is done by means of the manipulation of NRIM objects. In the computational view, the access to NRIM object can be realized in different ways depending on the mapping style of NRIM objects. When the NRIM object maps to the interface directly, the manipulation of each NRIM object is done in the form of the access to the interface. On the other hands, a sort of general access interface provides NRIM object manipulations when the NRIM objects are implemented inside a computational entities. The following section provides the description for both direct and indirect mapping approaches.

8.4.1 General access interface to NRIM objects

Considering the definition of NRIM, the most of the manipulations of NRIM objects can be done by means of either attribute oriented operations, e.g., get and replace, or NRIM object life cycle operations, e.g., create. The number of these operations is not so many. Once these operations are defined in a general access interface for NRIM objects, hence, such interface can satisfy the large portion of the requirement for NRIM object manipulation.

The attribute oriented operations, which addressed in X.720, are as follows:

- get attribute value
- replace attribute value
- replace attribute value with default value
- add attribute member: add an new member (members) to an existence list
- remove attribute member: remove an member (or members) from an existences list

Through the computational manifestation, the operations, ‘replace’, ‘replace with default’, ‘add member’, and ‘remove member’ can be generalized as a ‘set’ type of operations.

2. ‘Internal implementation’ will be realized as a finer grain internal computational objects or technology dependent implementation, e.g., C++ objects.

Also, the following life-cycle operations are addressed in X.720:

- create
- delete

X.720 addresses another operation semantic called ‘action’. Since the ‘action’ implies any of operation, it is too much abstract to define such operation in the computational view. Thus, each action definition should be mapped to each computational operation directly rather than mapped to a generalized operation in the access interface.

Another major management addressed in NRIM is the relationship management between NRIM objects. Like NRIM object manipulation, the manipulation of the relationships can also be defined commonly.

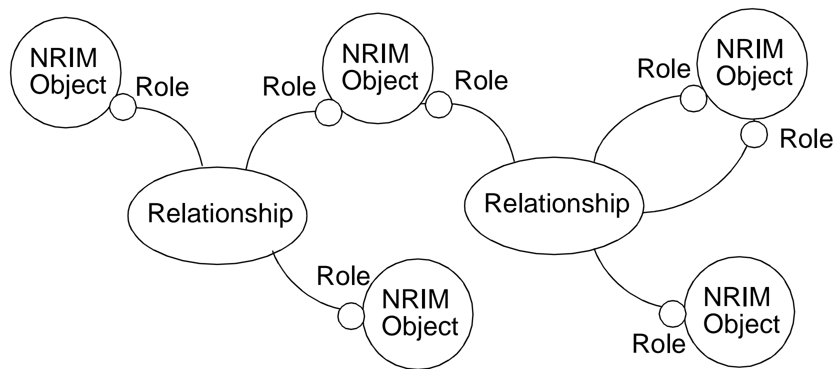


Figure 8-3. Relationship between NRIM objects

Figure 8-3 illustrates the notion of relationship between NRIM objects. A NRIM object may participate one or more relationships with particular roles, and may be bound to other NRIM objects or itself with relationships. The relationship can be created, deleted, and modified as the outcome of life cycle and attribute-oriented operations for NRIM objects. In addition, it can be manipulated by explicit operations. According to the definition in NRIM, the characteristics of the relationship is not so different from NRIM object, since the relationship is characterized by its attributes, behavior, and roles. The specific aspects of relationship management are the role handling and life cycle of relationship. To support life cycle of the relationship, the following operations need to be considered:

- create
- delete
- setup

The semantics of ‘create’ and ‘delete’ are mostly the same as the operations for NRIM objects. The ‘setup’ is the compound semantics of ‘create’ and a role attaching operation, which is a role handling operation. The role handling is the another specific aspect of the relationship management. With regard to the relationships, the NRIM object may be at-

tached to the relationship with the role, and detached under certain condition. Since the role is not an object, however, it cannot be created nor deleted individually and does not have any attributes. Therefore, the operations for role handling can be specified as follows:

- attach
- detach

The ‘attach’ operation causes the participation of the specified NRIM object in the relationship with the specified role, and the ‘detach’ operation lets the NRIM object leave from the relationship. Regarding the roles, some specialized query operations are also helpful for the relationship management. The following operations are defined in this specification:

- get roles
- get other related roles
- get role type
- get current relationship of NRIM objects

The attribute-oriented operations can satisfy the other query requirements for the relationship management.

8.4.2 Connection Performer Interface--Subnetwork Connection Management

Contrary to the approach in the Section 8.4.1, the SubNetwork and SubNetworkConnection are directly mapped to the interfaces. These interfaces also needs to provide the capabilities of controlling the edges, even if the edge itself is mapped to the implementation. In the case of the direct mapping, the attributes of the NRIM object are specified as the attributes of the interface. The operations other than attribute-oriented operations are designed as the operations in the interface. In this specification, a ‘SubNetwork’ is mapped to a ‘SncServiceFactory’ interface; a ‘SubNetworkConnection’, a ‘SncService’ interface. The ‘SncServiceFactory’ supports the following operations for the life cycle management of ‘SubNetworkConnection’, i.e., creation and deletion:

- create SubNetworkConnection(snc)
- create and setup SubNetworkConnection(snc)
- delete SubNetworkConnection(snc)

The ‘create’ operation causes the creation of the new ‘SncService’ interface instance in the computational view, and also cause the creation of the ‘SubNetworkConnection’ in the information view. The ‘create and setup’ is the compound operation of ‘create’ and initial configuration of the ‘SubNetwork’, i.e., attaching edges. This interface also supports the following operations for managing the edge life cycle:

- create edge
- delete edge

The ‘SncService’ interface provides the operations for the edge configuration, i.e., attaching to and detaching from the ‘SubNetworkConnection’, as follows:

- attach edge to SubNetworkConnection

- detach edge from SubNetworkConnection
- migrate edge from one SubNetworkConnection to another

Note that these interfaces are specified in line with logical definition in the information view. Once the NRIM objects are specialized for a particular technology, these interfaces also need to be re-defined according to the specialized definition, such as 'AtmVcSubNetwork' and 'AtmVpSubNetwork'.

8.4.3 Example scenarios

8.4.3.1 SubNetworkConnection creation

This scenario is the example procedure of the creation of the SubNetworkConnection from scratch.

Pre condition: A SubNetwork has been created: a 'SncServiceFactory' has been instantiated. The administrative state of SubNetwork is 'unlocked'; the operational state is either 'operational' or 'degraded'. A requesting entity, i.e., Layer Network Coordinator or certain Connection Performer, knows the reference of the 'SncServiceFactory'.

(1) Lock Sn: The requesting entity sets administrative state to 'locked'. The 'SncServiceFactory' should get 'Principal' to distinguish the requesting entity. After turing the administrative state to 'locked', the 'SncServiceFactory' only receives the request issued from the entity which locks the 'SncServiceFactory'. In other words, the 'SncServiceFactory' always examines 'Principal' matching the registered information at the time the 'SncServiceFactory' is locked.

(2) Create Edges: The requesting entity issues 'create edge' operations, e.g., one for a Root Edge and the other two operations for Leaf Edges. The outcome of this operation is the creation of Edge bound to the specified NWCTP. The 'administrative state' of the created Edge is 'locked', and 'operational state' is 'failed'. The created Edges are also protected from accesses other than the requests from the registered entity. If the operation successfully accomplished, the requesting entity receives the name of the Edge.

(3) Create Snc: The requesting entity issues 'create snc' operation. This operation causes the creation of 'SubNetworkConnection' and Instanciation of 'SncService' interface. The specified Root Edge is attached during this operation. The administrative state of the Edge turns to 'unlocked' before starting to attach the Edge. The administrative state of created SubNetworkConnection is 'locked', and operational state is 'failed'. For the continuity of the access control, the registered 'Principal' is copied to the created 'SncService'. If the operation is properly completed, the entity receives the reference of the created 'SncService' interface.

(4) Unlock Sn: The requesting entity sets administrative state to 'unlocked'. The 'SncServiceFactory' cleans up the registered 'Principal' and turn to be ready for accepting operations.

(4) Attach Edge: The requesting entity issues ‘attach edge’ operation to the created ‘SncService’ interface. The specified Leaf Edge is attached in this operation. The administrative state of the Edge turns to ‘unlocked’ before starting to attach the Edge.

(5) Unlock Snc: The requesting entity sets administrative state to ‘unlocked’. The ‘SncService’ cleans up the registered ‘Principal’, turns the operational state to ‘operational’ (or possibly degraded), and turn to be ready for accepting operations. While this operation, the operational states of attached Edges also turn to ‘operational’ or ‘degraded’

Post condition: The SubNetwork and ‘SncServiceFactory’ interface exist. The SubNetworkConnection is created and the ‘SncService’ is instantiated. The administrative states of both SubNetwork and SubNetworkConnection (or ‘SncServiceFactory’ and ‘SncService’) are ‘unlocked’. Those operational states are either ‘operational’ or ‘degraded’. The Edges are attached to the SubNetworkConnection. The administrative states of the edges are ‘unlocked’; the operational states, ‘operational’ or ‘degraded’.

9. Accounting Management Components

Accounting management in NRA, therefore this section of accounting management components in NCS deals with the following two issues;

- Accounting/billing management in TINA communication service
- Accounting management architecture

Although the first issue can be seen as an example of the second one, i. e. accounting/billing management of TINA communication service can be performed by giving generic accounting management interfaces to network resource components, we prioritize our focus on the first one, as it has primary importance in TINA service realizations. As such, we give our presentation in the order of practical importance in this NCS section, rather than generality of the components themselves. In the following parts of this section, we proceed from an scenario based on a typical usage of TINA communication service, from which we derive management and control interfaces. From that design, we proceed to present common architectural components, which would support the common accounting management architecture presented in NRA.

9.1 Overview of Accounting Management in TINA Service

Before we proceed to present accounting management components, we illustrate the usage of accounting management in a typical TINA service scenario and its relationship to service/network components.

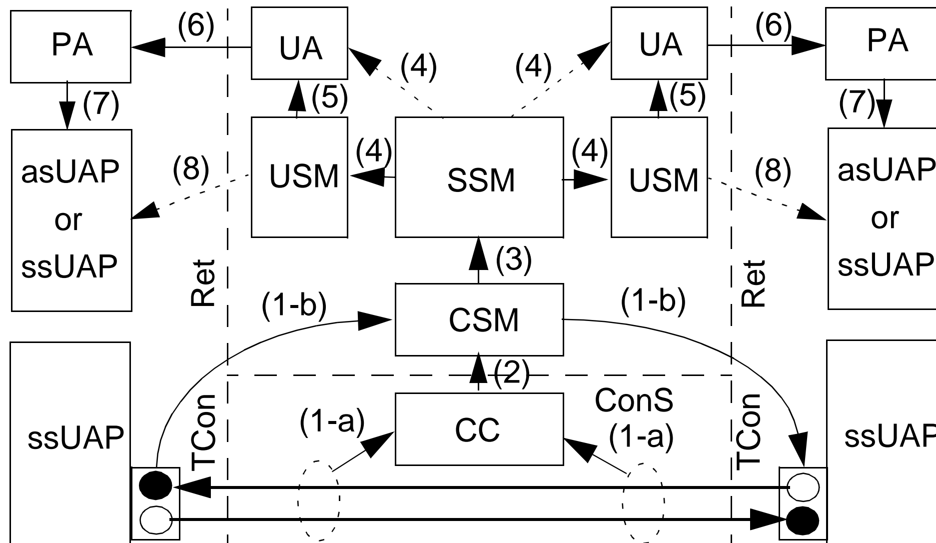


Figure 9-1. Overview of Accounting Management in TINA Service

Figure 9-1 shows an overview of accounting/billing management in a TINA service. In this scenario, two UAPs in user domains communicates each other through a bi-directional stream binding.

The preceding contexts of the above scenario are:

- *Service session creation:* service components such as UA, USM, SSM, etc. are already created and are in place.
- *Network resource components set-up:* network resource components such as CC, LNC, TCM, etc. are already created and are in place. For the sake of simplicity, the figure does not show all the network resource components.
- *Stream binding set-up:* necessary NFEPs and SFEPs are already provisioned are bound to the stream bindings.

Although all of these above steps are part of the accounting management, we do not delve into their details, as they do not directly correlate with usage accounting; they should rather be considered as provisions for usage accounting. For the same reason, accounting information after usage, e. g. deletions of components, are also out of the scope of this section. Those accounting information from pre- and post-usage provisions may be useful for fault management and system maintenance purposes, however.

We also assume that service transaction concept is in effect, i. e. billing information may be correlated with performance monitoring results during the transaction. In the current example, we assume the followings:

- The two users are on two separate service transactions with the retailer, whose contexts are passed through Ret.
- The retailer is on a service transaction with the communication provider, whose context is passed through ConS. This service transaction corresponds to client-server relation between stream flow connections (SFCs) and network flow connections (NFCs) on the stream binding.
- The retailer is acting as a billing agent for the communication service provider. Although it is possible that the retailer does not act as an agent, this is probably the most ‘typical’ usage of TINA service, implicitly assumed in the current TINA RfR specifications. A billing agent does not necessary imply, however, that the connectivity provider (CP) be made opaque from the users. It can be made transparent, i. e. a separate bill from the CP, or be made opaque, i. e. a combined, integrated bill from the retailer only, depending on the scope of the binding contexts.

The above assumptions imply the followings to the maintenance of service quality in TINA. In the transparent billing, both business entities, i. e. the retailer and the communication service provider, are visible from the users. Since visibility in the billing should be translated into responsibility on service quality maintenance, performance monitoring results should be correlated with billing information separately in case of transparent billing at the conclusion of service transactions.

9.1.1 Visibility of Billing Context

Before we proceed to explain the scenario depicted in Figure 9-1, it is necessary to illustrate the accounting relationship established by billing (accounting management) context, which essentially dictates who bills who in TINA services. In the current service architecture, we distinguish two different cases.

- *Visible Billing Context:* in the visible billing context, the retailer and the connectivity provider (CP) look as two independent separate entities to the eyes of the consumer. As such, two separate billing items are generated from the two entities.
- *Invisible Billing Context:* in the invisible billing context, the retailer appears as the integrator of all the necessary sub-services, which include communication service supported by the CP. The consumer does not directly deal with the CP, as such no contexts may be passed to the CP from the consumer. The bill from the CP may appear as one of billing items of the retailer, however.

We do not intend to make comparison of two billing models, nor do we assume that either of the models is to be used or more likely used in TINA services; our purpose is to explain that the TINA accounting management architecture supports both.

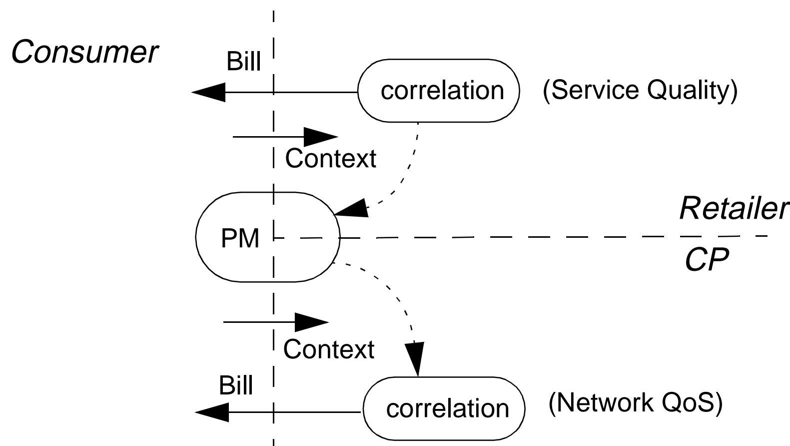


Figure 9-2. Visible Billing Context Example

Figure 9-2 illustrates a visible billing example, where the consumer sets up two separate billing contexts (accounting management contexts) to the retailer and to the CP. The results from performance monitoring (PM) are fed back to the respective providers separately, those on network QoS to the CP whereas those on service quality to the retailer, which are to be taken into account in the respective bills. In other words, in terms of the stream binding, the retailer is responsible for and is billing from SFEP to SFEP, whereas the CP is responsible for and is billing from NFEP to NFEP. This visible billing context case, however,

does not exclude the retailer to act as a billing agent for the CP, in which the consumer is still able to set-up a separate context with the CP, but he/she receives the bill indirectly via the retailer.

Service quality issues in TINA are discussed in more details in [12], and service transaction is explained in TINA service architecture [1].

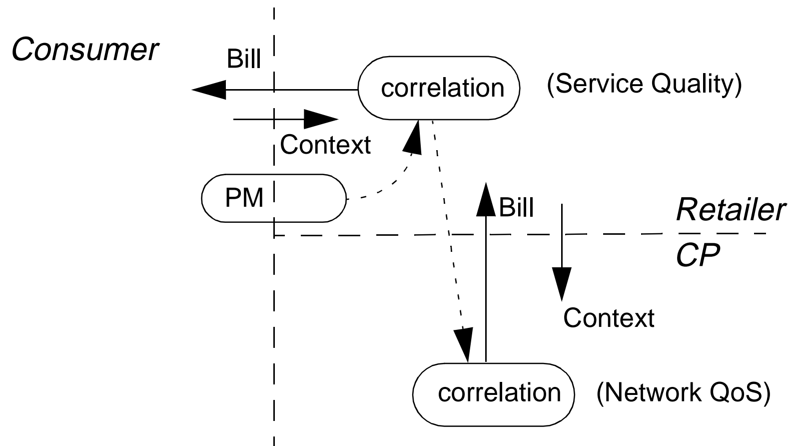


Figure 9-3. Invisible Billing Context Example

Figure 2-5 illustrates a invisible billing context example. The consumer does not see the CP directly, and its bill is included in the one from the retailer. The CP is responsible for network QoS of the stream binding, i. e. from NFEP to NFEP, which are billed to the retailer. The retailer is responsible for service quality of the stream binding, i. e. from SFEP to SFEP, which are billed to the consumer.

9.2 An Example Scenario of Accounting Management

We now proceed to explain the accounting management scenario of Figure 9-1. In this scenario, we assumed a transparent billing context model, where the retailer acts as a billing agent for the CSP. Although the flow of accounting events may differ, overall accounting management architecture and necessary component specifications are almost the same for other billing models.

1-a. Bare transport level traffic is measured, as they are specified in NRIM [5], which corresponds to the NFC under measurement. The accounting events are recorded, or collected using an event management ladder [4], such that usage information of the NFC is collected by CC. Although network resource components such as LNC, TM, etc. do not appear in the figure, they are assumed as they are described in NRA, forming an event management ladder as their instances are created.

1-b. In WYSWYP (What You See is What You Pay) [14] performance monitoring, performance and traffic on SFC are measured at SFEP, with assistance from TCSM (not shown in the figure). This provision is particularly useful for connection-less traffic on IP network, where internet service provider (CP) is not concerned with per connection QoS and its traffic measurement. The accounting management events, which may include both traffic measurement and performance monitoring results, are sent to and collected by CSM, via TCSM in the consumer domain.

2. Accounting events (records) are passed to CSM from CC. When on-line billing is used, filtered accounting events, which may cause a change in the retailer's billing status, are passed on-the-fly during the service transaction. When on-line billing is not used, only the calibrated billing information is passed from CC to CSM at the conclusion of the service transaction. The billing information is calibrated by taking both performance monitoring results and price compensation scheme into account, which are agreed at the beginning of the service transaction.

3. Filtered accounting events are passed to SSM, which in turn may pass the events to corresponding USMs or to UAs, depending on the availability of the components.

4. SSM passes accounting events or billing information to corresponding per user components. When on-line billing are used, and the bills are to be split among interested parties, the accounting events from CSM are stipulated and then passed to the corresponding USMs of the participating (paying) users. When on-line billing is not used, and only the billing information is obtained from CC at the conclusion of the service transaction, the stipulated billing information may be passed to UAs, not USMs, as the USMs may be non-existent at the time. This situation occurs because TINA service session is a multi-party entity, that is a user can leave a service session whereas other users are still on the session.

5. Accounting events sent to USM are turned into billing information, which is to be stored in UA. UAs continue to accumulate billing information of service sessions per user basis, which are made permanent to acquire fault tolerance. Temporary billing information of the on-going service sessions are also stored at UA, which can be used for on-line billing.

6. UA periodically displays billing information of on-going service sessions via the corresponding PA, when on-line billing is used. The user is able to query the current billing status using PA, which requests billing information from UA.

7. asUAP or ssUAP may receive periodic update or per request update of its accounting information from PA, if accounting information is cached in PA from UA.

8. There is an alternative path to obtain accounting information. In particular, when customized accounting/billing service is required, ssUAP or asUAP may obtain accounting/billing informaiton through a ancillary accounting USM, which may provide more sophisticated interface with on-line update than the one provided by generic PA.

In this section of NCS, we only specify the network resource components such as CC, LNC, TC, etc., whose instances appear in the CP domain. Accounting management of service components such as SSM, USM, etc., are specified in Service Component Specification (SCS) [10] document. In this section, however we took a service-centered view on accounting management, i. e. trying to explain accounting management from the billing point of view. As we noted earlier in Section 9.1, there are possibly many accounting events not immediately related to billing information, e. g. from creation/deletion of CC to NEL, EML level objects such as links, termination points, etc. Though those accounting events can give useful information on the status of the network, not all the information is used for the billing, and only a few important ones among those events are sufficient to generate a correct bill.

For example, upon the completion of connection management, an end-to-end trail of created, upon which an NFC can be bound, upon which an SFC can be bound. From the user's point of view, the billing starts only when the SFEP becomes usable by the preceding stream binding, therefore the preceding accounting events are irrelevant from the billing point of view, though they can be quite important from the connection management point of view.

In this section of NCS, we classify accounting events in two classes based on their relevance to the billing.

- *Essential accounting events:* this class of events are essential for generating a correct bill for TINA services using stream bindings. In other words, these events are necessary to calculate bills for multimedia services offered by TINA. Accounting events which may appear in Figure 9-1 belong to this class.
- *Non-essential accounting events:* this class of events are not essential for generating a correct bill. Although creation of CC is indispensable in TINA connection management, the event itself does not appear, nor correlate with the billing information.

We primarily focus on the essential accounting events, because of its practical importance. In particular, when on-line billing is used and essential accounting events need to be handled on the fly during the lifetime of a service session, these events and the corresponding interfaces need to be specified at respective components in SCS and NCS.

9.3 Accounting Event Management

Though we assume event management functions be provided by DPE, which enables to establish an event channel between concerned objects. The provision is mandatory when on-line billing is expected. In legacy systems and in some operational environment, however, on-line billing may be considered too difficult to realize, or it may be considered too computationally expensive to process those accounting events near real-time. In those

cases, accounting events, or common call records, would still prefer to be processed in a batch style, where all the accounting events are collected at log manager and are stored there, only to be processed periodically at the end of billing cycles.

Table 9-1. Accounting Event Class and Event Management

	On-line Billing	Batch Billing
Essential Accounting Events	I) DPE event management + event mgmt. ladder	III) Batch Processing + Log manager
Non-essential Accounting Events	II) Batch processing + Log manager	IV) Batch processing + Log manager

Table 9-1 illustrates relationship between accounting event classes and event management schemes. Quadrant I is the focus of this NCS section. Although other quadrants are also parts of TINA, they are almost the same, and necessary components and interfaces are well covered by generic part of accounting management architecture.

The underlying event management mechanisms is to be provided by DPE. DPE event management facilities will be based on the following specifications.

- CORBA COS Event Service
- X/Open Notification Service
- DPE Notification Service
- CORBA Notification Service

9.4 Essential Accounting Events in Network Resources

Essential accounting events in TINA network resources, which are covered in NCS, are the following:

1. *Event Name:* completion of stream binding
   Event Type: t_AccountingEvent
   Sender: TCSM
   Receiver: CSM::i_AccountNotify, CC::i_AccountNotify
   Contents: time stamp
   Description: upon arrival of this event, UAP in the consumer domain is allowed to start sending data, and the billing can be started. An event to CSM activates billing in the retailer, whereas another event to CC activates billing in the CSP.
2. *Event Name:* completion of stream un-binding
   Event Type: t_AccountingEvent
   Sender: TCSM
   Receiver: CSM::i_AccountNotify, CC::i_AccountNotify
   Contents: time stamp, measured traffic
   Description: upon arrival of this event, UAP in the consumer domain is no longer

allowed to sending data, and the billing must be stopped. An event to CSM stops billing in the retailer, whereas another event to CC stops billing in the CSP.

3. *Event Name:* traffic measurement on stream binding
Event Type: t_AccountingEvent
Sender: TCSM
Receiver: CSM::i_AccountNotify, CC::i_AccountNotify
Contents: time stamp, measured traffic
Description: when on-line billing is used, and when charging is based on amount of data being sent, not time-based, TCSM periodically reports measured traffic, which is to be converted to billing info. at CSM and CC, respectively.

Since essential accounting events originate in the consumer domain, it should be possible that they can be made non-reputable by attaching a certificate generated by TCSM, to prevent possible toll fraud.

9.5 Non-essential Accounting Events in Network Resources

NRIM [5] specifies following object types as accountable:

- Stream Flow Connection (SFC)
- Network Flow Connection (NFC)
- Terminal Flow Connection (TFC)
- Trail
- Subnetwork Connection (SNC)
- Link Connection (LC)
- Tandem Connection (TC)

Although they are all indispensable parts of TINA connection management, they are categorized as non-essential, only because they do not activate billing by themselves. These accounting events, however, can be very useful for monitoring usage and performance of the network. Accounting events of a connection can be measured at measurement points within the flow, most typically at its respective endpoints. For example, a trail is terminated at two Network Trail Termination Points (NWTTPs). When these termination points are made Accountable Objects (Section 9.6.3), they report accounting events to a corresponding Log manager (Section 9.6.4) in the respective management domain.

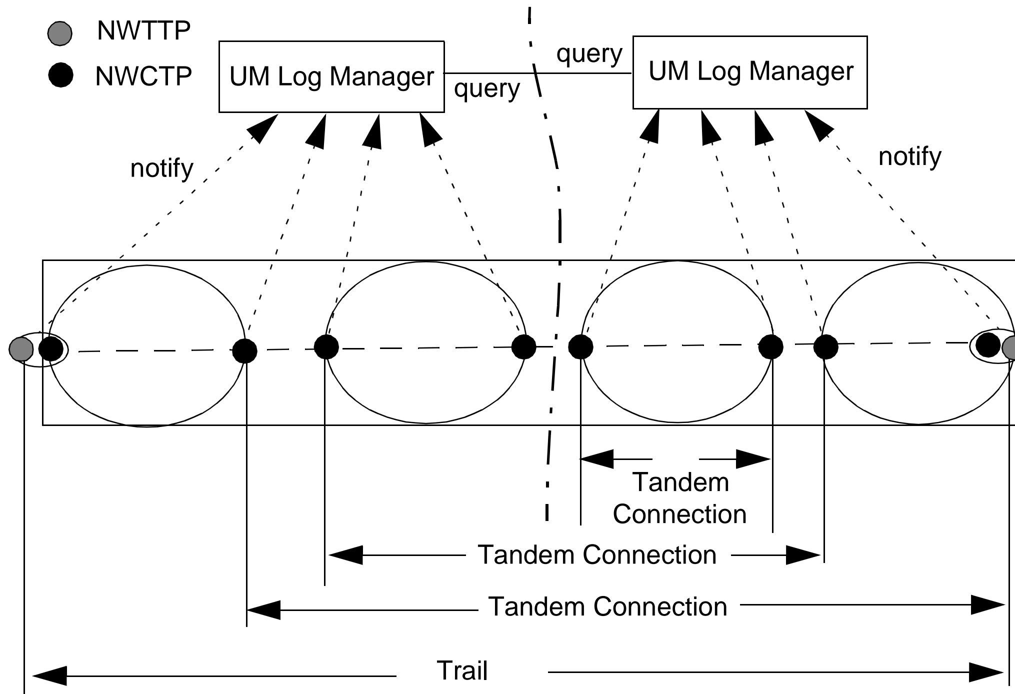


Figure 9-4. Trail and Tandem Connections

Figure 9-4 illustrates an accounting example for a trail and associated tandem connections. The trail extends over two management domains, each of which has an associated log manager in the domain. Non-essential accounting events are notified and then collected at respective UM Log managers, which provides persistent storage to keep the records. Two Log managers provides a query interface to the other, such that one manager can refer to the records kept by the other manager.

Termination points of respective connection object types can generate the following accounting events:

1. *Event Name:* completion of connection set-up
   Event Type: t_AccountingEvent
   Sender: connection TP
   Receiver: LogManager::i_EventNotify
   Contents: time stamp
   Description: this event notifies a connection (e. g. SFC, NFC, etc.) set-up at the sender termination point is completed.
2. *Event Name:* completion of connection release
   Event Type: t_AccountingEvent

Sender: connection TP
Receiver: LogManager::i_EventNotify
Contents: time stamp, measured traffic
Description: this event notifies a connection release at the sender termination point is completed.

3. *Event Name:* traffic measurement on connection
   Event Type: t_AccountingEvent
   Sender: connection TP
   Receiver: LogManager::i_EventNotify
   Contents: time stamp, measured traffic
   Description: traffic on a connection may be periodically measured, which is kept as a record at Log Manager.

9.6 Generic Accounting Management Components

9.6.1 AmcLadder

Object AmcLadder is a generic object for accounting management, from which other components such as SSM and USM can be derived. In other words, AmcLadder object does not exist as a stand alone object, it exists only as a base object for other service components.

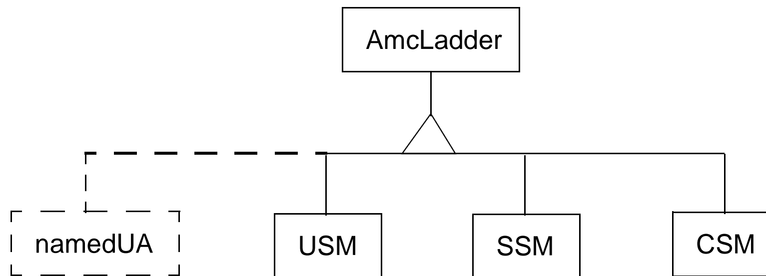


Figure 9-5. Object Inheritance of AmcLadder to Service Components

Figure 9-5 illustrates object inheritance of AmcLadder to service components. Although AmcLadder does not show as a service component itself, we are able to treat actual service components (SSM, USM, etc.) as if they are all amcLadder, as far as accounting management is concerned.

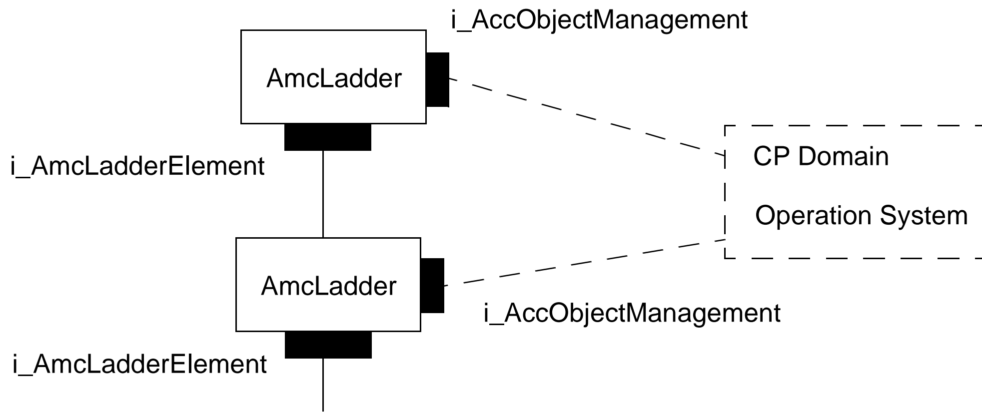


Figure 9-6. Formation of Accounting Management Ladder

Figure 9-6 illustrates the formation of an accounting management ladder. When an AmcLadder is created (actually SSM, CSM etc.), its notification destination is set to another AmcLadder, which is to be positioned above in the ladder. For example, when a CSM is created, its notification destination is set to the corresponding SSM, such that an accounting event path is formed among the session components. Interface *i_AmcLadderElement* of the upper element becomes the notification destination of the lower element.

Object AmcLadder is also an X.742 compliant accountable object. As such, its accounting activities are controllable from outside, using a management interface *i_AccObjectManagement*. Although not being specified in this SCS document, potential clients of this interface are management applications, e.g. an operation system in the retailer domain. Service Factory may also use the interface, when it creates service session and its components, thus forming a part of the accounting management ladder.

Table 9-2. Supported Interfaces of AmcLadder

Interface	Client(s)	Event traces	IDL
i_AmcLadderElement	AmcLadder	Scenario in Accounting Management (sect 5)	sect 8
i_AccObjectManagement	Operation System [AmcLadder] [SF]		sect 8

Table 9-3. Required Interfaces of AmcLadder

Server	Interfaces
AmcLadder	i_AmcLadderElement

The accounting management ladder was originally proposed to handle on-line accounting. As such, the concept is widely used in service level accounting described in SCS [10]. In NCS, however, most of the accounting will be performed using Usage Metering Log Manager in off-line style, therefore the ladder concept may not be used as much. In NCS accounting, the generic accountable object management (i_AccObjectManagement) and policy-driven management domain control will be more dominant. The ladder concept, however, will be useful when on-line management of events is necessary, in particular fault and alarm management.

9.6.2 i_AmcLadderElement

This interface provides destination of events from the lower elements of the ladder. It inherits from COS Event Management (CosEventComm::PushConsumer). No other operations are provided to this interface.

9.6.3 Accountable Object

- Description: the accountable object is a TINA resource object with an accounting management interface. In other words, the resource object must support *i_AccObjectManagement* interface in its ODL specification, to become an accountable object.
- Behavior: the metering of the accountable object can be controlled through the management interface, *i_AccObjectManagement*.
- Operations: the following operations should be supported.
  - Control operations:
    start
    stop
    suspend
    resume
    set_state
    set_accounting_cycle
  - Notification control operations:
    suspend_notification
    resume_notification
    set_verbosity_level
    set_Notification_Destination
    reset_Notification_Destination

The reader should substitute “connection TP” for respective termination point object type of connection objects such as SFC, NFC, and so on.

9.6.4 Usage Metering Log Manager (UMLog)

The log manager is essentially the same as the *log* managed object of X.721 [12]. UM Log supports an interface (*i_UMLogOperation*), which provides operations for management and maintenance of log records.

- Description: the log manager performs logging.

- Behavior: the same as X.721 log managed object.
- Operations (*i_X721LogOperation*):
  - Control operation:
    start
    stop
    suspend
    resume
    set_log_attributes
    reset_log_attributes
  - Notification operation:
    event_notify
- Operations (*i_UMLogOperation*):
  - Log Management operation:
    store
    getUserLogEntries
    removeUserLogEntries
    getSessionLogEntries

9.7 Management Domain Related Components

9.7.1 Accounting Policy Manager

- Description: the accounting policy manager maintains the accounting policy. For example, when a new resource object is created in the domain, the object obtains the accounting policy from the accounting policy manager. The accounting policy manager provides *i_AccountingPolicyManager* interface.
- Behavior: the accounting policy manager serves as a policy server, from which the objects in the domain can obtain the policy. When a new policy is installed or an old policy is updated, the policy needs to be notified to all the accountable objects in the domain, with the help of the network resource management of the domain.
- Operations (*i_AccountingPloicyManager*):
  start
  stop
  suspend
  resume
  update_policy
  delete_policy
  propagate_policy

9.8 Relationship to Other Documents

Service Architecture [9] describes FCAPS in TINA services, and overview of accounting in TINA service architecture. Service Component Specification [10] describes service components and interfaces provided for accounting management. Network Resource Information

Model [5] includes information model of accountable object and accounting management. Network Resource Architecture [4] describes overall structure of resource level accounting, accounting management domain, and FCAPS concept.

10.Fault Management Components

10.1 Introduction

Fault Management is concerned with detection, localization and correction of abnormal behaviour of the telecommunication network and its environment.

All CM Managed Objects (COs) shall allow the emission of alarms associated with CM. Fault Management (FM) shall create alarm records which can be used for activities such as fault localization, fault correction, and alarm summary reporting.

How can an abnormal behaviour of an element be detected? There are two methods:

- Use a test engine, i.e. a element that tests periodically all the COs. If the test engine detects a fault, it will send a notification to the Notification Server (NS) and the NS will send alarms to the FM components [1].
- The client of the element, when the invocation of an operation return an exception or no response is received, then a fault can be detected. When the client detects a fault, it will send a notification to NS and the NS will send alarms to the FM components.

The first method is better than the second because using the second method nobody detects a fault until the client invokes a operation. However the first method may be inefficient with a reduced test interval. So the best option is use both methods, i.e. to an element that test periodically all the COs (the test interval shouldn't be very short) and allow the client to send notifications when a fault is detected, in order to not overhead the messages in the system.

10.2 Computational Viewpoint

The network resource fault management services are provided by the interaction of the Computational Objects (COs) inside and outside of the fault management area. Figure 8-1 shows the fault management functions and the fault management interfaces services and activities.

COs identified for network resource fault management are as follows:

- Alarm Manager (AM): receives fault-related alarm from Managed Objects (COs) and performs relevant procedures for alarm correlation, alarm filtering, forwarding the alarm to fault coordinator or fault management service user and for alarm record management. Each AM has its own discriminating criteria through which incoming alarms are logged and forwarded to relevant computational objects in the system.
- Fault Coordinator (FC): includes capabilities to internally analyze alarms re--ceived from multiple COs to determine next possible step for fault localization/ correction. For this purpose, the FC correlates all available information to refine information concerning the root cause of event in question. During the analysis, the TDS can be invoked to run a test as appropriate.

- Testing/Diagnostic Server (TDS) is concerned with testing of COs for the purpose of service and function verification of COs. This CO must support an operation to allow realize a testing of all COs periodically, in this way the TDS may invoke a test operation periodically, and detect if a COs it's crashed.

Figure 10-1. Fault Management Computational Viewpoint

10.3 Functions

This section describes the steps to be realized by COs and FM computational objects when a fault is detected.

10.3.1 FM functions

The TDS performs the following operations:

- Perform a test on the elements to be supervised to detect faults.
- While it's realizing the testing detects that a element doesn't work correctly (Let elem1 that element)
- Send an alarm to the AM.
- Continues performing the test.

The AM performs the following operations:

- When it receives an alarm from the TDS or NS, it realizes relevant procedures for alarm correlation, alarm filtering and alarm record management.
- Forward the alarm to the FC.

The FC realizes the following operations:

- When it receives an alarm from the AM, the alarm is analyzed to determine next possible step for fault localization/correction.
- Invokes an operation to the CMC or NRCM for restore elem1.

10.3.2 COs functions

All the CM COs must have the following interfaces:

Figure 10-2. Figure 8-2.Notifications interfaces

i_NotifyCtrl This interface provides operations for controlling the emission of notifications from the server to the client. It provides the following operations:

- Enable notifications: this operation is used by the client for instructing the server to emit notifications regarding the server

- Disable notifications: this operation is used by the client to suspend emission of notifications from the server
- Set notification destination: this operation is used to communicate the interface notification destination reference to the server.

i_Notify This interface is used by the client to receive notifications associated which the server.

The CM COs configurators (CMC & NRCM in CM) perform the following operations:

- Configures elem1.
- To update the children list of the higher element in the hierarchy of elem1 (elem1_parent). For example: update snwCpChildren operation if elem1_parent is a NML-CP.

The elem1 performs the following operations:

- Notifies that it is configured. This notification will be received by the AM. The AM will use it to delete the respective alarm (if it exists)
- Reset all the elements that it is on the lower level of the hierarchy.

The elem1_parent realizes the following operations:

- It will send a notification to the NS (if an operation was invoked on and a failure was detected elem1).
- Indicates to its parent (elem1_grandparent) that it can't realize the operation temporarily because elem1 is not available, the indication avoids the expiration of the ORBIX operation timeout (if an operation was invoked on and a failure was detected elem1).
- When receives a operation to update the children list will release all the resources that it was using with respect to elem1 (i.e. the children).

The `elem1_grandparent` when receives a notification from the `elem1_parent` (using the i_Notify interface) should act consistently, probably sending a notification to the higher element in the hierarchy and releasing all the resources that it had reserved for such connection or group of connections.

The `elem1`, `elem1_parent` and `elem1_grandparent` can be any element of the hierarchy, for example:

- `elem1_grandparent`=LNC, `elem1_parent`= Top NML-CP and `elem1`=EML-CP
- `elem1_grandparent`=Top NML-CP `elem1_parent`=NML-CP and `elem1` =RA
- `elem1_grandparent`=CC, `elem1_parent` =LNC and `elem1`=Top NML-CP
- `elem1_grandparent`=LNC, `elem1_parent`=Top NML-CP and `elem1`=NML-CP (if there are more than two levels of Cps)
- etc.

10.3.3 Future extensions

This chapter is a first draft that should be extended according to the following aspects:

- Characterize all types of emitters.
- Characterize all the notifications that each element can send.
- Define all the operations that an element has to perform when a notification is received.
- Fault Management using the Notification Server.

11.Document History

A document schedule according to the plan for this activity is the following:

- End of March: Edition of the first version of the document (this version).
- End of April: Edition of final version of the document.
- End of June: Update of the document based on problems or experience gained in CM implementation.
- End of July: with considrable inputs from VITAL 2.0, many of the idl files, including main parts of connection management, were supplied from VITAL.
- Ver. 2.1 was released at August 13, 1997, of which Jarno Rajahalme and Frank Steegmans as editors.
- Ver. 2.2 was scheduled to be released in October, which was delayed till November 30. Due to scheduling problems and diminishing number of core-teamers, the document was not finished.
- Ver. 2.2 is released “as is”, as the final draft from the core-team. It has not been reviewed, or by the core-team. It is hoped, however, that the current version serves as the starting point of the fututure activities within TINA-C, of the network resource architecture and ConS reference point (Dec. 20, 1997).

References

TINA-C Documents

[1] *Computational Modelling Concepts*, Version 3.2, TINA document no. TP_HC.012_3.2_96, TINA-C, May1996; TINA-C valid baseline.
Authors: T. Handegård, and many TINA-C Core Team members
http://tinac.com:4070/96/dpe/docs/computational_model/v3.2/cmc.ps

[2] *Connection Management Specifications*, Draft, TINA document no. TP_NAD.001_1.2_95, TINA-C, Feb. 1995; TINA-C internal report.
Authors: F. Ruano, C. Aurrecoechea, A. Hopson, H. Oshigiri, J. Pavón
http://tinac.com:4070/95/resources/viewable/cmspecs.ps

[3] *Information Modelling Concepts*, Draft, TINA document label TB_EAC.001_1.2_94, TINA-C, Mar. 1995; TINA-C valid baseline.
Authors: E. Colban, H. Christensen
file:/u/tinac/94p2/base/info/info.ps

[4] *Network Resource Architecture*, Version 3.0, TINA document no. NRA_v3.0_97_02_10, TINA-C, Feb. 1997; TINA-C valid baseline.
Authors: F. Steegmans, C. Abarca, J. Forslöw, T. Hamada, S. Hogg, H.B. Jeon, D.S. Kim, H.Y. Lee, N. Natarajan
http://www.tinac.com/deliverable/nra_v3.0_public.ps

[5] *Network Resource Information Model Specification*, Draft, TINA document no. XXX, TINA-C, May 1997; TINA-C internal review release.
Authors: N. Natarajan, H. Flinck, R.M. Rosli
file:/u/tinac/97/resources/network/docs/nrim/v2.0.2/NRIM.ps

[6] *The ConS Reference Point*, Version 1.0, TINA-C Cons-RP review panel, Feb. 1997; TINA-C draft.
Authors: N. Natarajan, L. Demounem
http://tinac.com:4070/97/integration/rfrs/RFR-96-02/feb1297/consRP.ps

[7] *TINA Business Model and Reference Points*, Version 4.0, TINA-C May 1997; TINA-C valid baseline.
Authors: H. Mulder, M. Yates, W. Takita, L Demoudem, R. Jansson
http://tinac.com:4070/97/integration/docs/business/viewable/final_v4.0.ps

[8] *TINA Object Definition Language MANUAL*, Version 2.3, TINA document no. TR_NM.002_2.2_96, TINA-C, July 1996; TINA-C draft.
Authors: A. Parhar
http://www.tinac.com/deliverable/odl96_public.ps

[9] *Service Architecture*, Version 5.0, TINA-C June 1997; TINA-C valid baseline.
Authors: L. Kristiansen, C. Abarca, P. Farley, J. Forslöw, J.C. García, T. Hamada, P.F.

Hansen, S. Hogg, H. Kamata, C.A. Licciardi, H. Mulder, E. Utsunomiya, M. Yates http://tinac.com:4070/97/services/docs/sa/sa5.0/final/main.ps and annex.ps

[10] *Service Component Specification*, Version 1.0, TINA-C December 1997; TINA-C valid baseline.

Auxiliary Project Documents

[11] VITAL, *Connection Management Specifications for VITAL 2nd phase*, CEC Deliverable no. AV003/TID/CM/DS/I/002/A2, Jun. 1997; VITAL internal specification.

ITU-T Recommendation

[12] X.721

[13] H.245

Other Documents

[14] T. Hamada, S. Hogg, J. Rajahalme, C. Licciardi, L. Kristiansen, P. Hansen, “Service Quality in TINA”, proc. in EDOC’97, Oct. 1997, Gold Coast, Australia.

Misc.

[15] Tcon-RP RFR/S Response, TINA document no. ???, TINA-C Core Team, Nov. 1996

[16] Ret-RP RFR/S Response, TINA document no. ???, TINA-C Core Team, Nov. 1996

12.Acronyms

ATM	Asynchronous Transmission Mode
CA	Resource Configuration Management Application
CC	Connection Coordinator
CG	Connection Graph
CM	Connection Management
CMA	Connection Management Architecture
CMC	Connection Management Configurator
CSM	Communication Session Management
CO	Computational Object
CORBA	Common Object Request Broker Architecture
CP	Connection Performer
DPE	Distributed Processing Environment
EML	Element Management Layer
EML-CP	Element Management Layer-Connection Performer
FC	Flow Connection
IDL	Interface Description Language
KTN	Kernel Transport Network
LCG	Logical Connection Graph
LNC	Layer Network Coordinator
LNW	Layer NetWork
LTP	Link Termination Point
MO	Managed Object
MSC	Message Sequence Charts
MRCM	Management Resource Configuration Manager
NCG	Nodal Connection Graph
NE	Network Element
NFC	Network Flow Connection
NFCBranch	Network Flow Connection Graph
NFEP	Network Flow End Point
NML	Network Management Layer
NML-CP	Network Element Layer-Connection Performer
NRCM	Network Resource Configuration Management

NRIM	Network Resource Information Model
NRM	Network Resource Map
NWTP	Network Termination Point
NWCTP	Network Connection Termination Point
NWTTP	Network Connection Termination Point
NWTpPool	Network Termination Point Pool
OMT	Object Modelling Technique
PCG	Physical Connection Graph
QoS	Quality of Service
RA	Resource Adapter
RC	Resource Configuration
SFC	Stream Flow Connection
SFEP	Stream Flow End Point
SFCBranch	Stream Flow Connection Branch
SG	Session Graph
SI	Stream Interface
SML	Service Management Layer
SNC	Subnetwork Connection
SNW	Subnetwork
TC	Tandem Connection
TCG	Terminal Connection Graph
TCSM	Terminal Communication Session Manager
TFC	Terminal Flow Connection
TINA	Telecommunication Information Networking Architecture
TLA	Terminal Layer Adapter
TP	Network Termination Point
TPPool	TP pool
VC	Virtual Channel
VCC	Virtual Channel Connection
VCI	Virtual Channel Identifier
VITAL	Validation of Integrated Telecommunication Architecture for the Long term
VP	Virtual Path
VPC	Virtual Path Connection

VPI Virtual Path Identifier

13.Glossary

Action
: An operation on a managed object, the semantics of which are defined as part of the managed object class definition.

Attribute
: Information of a particular type concerning an object.

Behavior
: (Of a managed object:) The way in which managed objects, name bindings, attributes, notifications and actions interact with the actual resources they model and with each other.

Communication Session Manager
: A computational object in the connection management functional area. It provides clients with the service of interconnection of computational stream interfaces.

Computational Interface
: An abstraction that provides access to a subset of capabilities provided by a computational object.

Computational Object
: An abstraction that encapsulates data and processing; provides a set of capabilities that can be used by other objects.

Connection
: A basic abstract concept in most communication models. A connection is an association between two or more end points which is used to convey information between these end points. The term connection can be used recursively, so connections are usually composed of sub-connections, each of which can be managed independently. Examples of end points are sinks/sources in stream interfaces, sink/source ports in a connection graph and termination points of a subnetwork. The term "connection" should be used with other clarifying adjectives in order to have a specific meaning.

Connection Coordinator
: A computational object in the connection management functional area. It provides clients with the service of interconnection of addressable termination points, multipoint-to-multipoint bidirectional. It hides from clients the concepts of layering and partitioning of transmission networks. The interface specification is based on the connection graph concept.

Connection Graph
: An object type used in the computational interface specification of a Communication Session Manager and a Connection Coordinator. It is used as a container object for other object classes to model transport abstractions.

Connection Management
: One of the six TINA-C network management functional areas. Functions in this category are responsible for establishing, modifying and releasing connections in response to client requests.

Connection Management Configurator
: A computational object in the connection management functional area. The Connection Management Configurator provides an interface to Resource Configuration functions that must configure Connection Management functions (such as a Communication Session Manager, a Connection Coordinator, a Connection Performer) that are co-located in one building block.

Connection Performer
: A computational object in the connection management functional area. It provides clients with the service of interconnecting termination points of a subnetwork. Every subnetwork is managed by one Connection Performer.

CorrelationId
: Identifies the mapping between a SFC and a NFC.

Distributed Processing Environment
: The Distributed Processing Environment (DPE) provides the infrastructure for computationally specified applications on top of Native Computation and Communication Environments (NCCEs). It enables the interworking of these applications residing on different, possibly heterogeneous, NCCEs in a distribution transparent way. The DPE consists of a DPE runtime and a DPE development system.

Edge
: A Managed Object that represents association between a subnetworkConnection and a NWCTP or a NWTTP.

Element Management Layer
: A sublayer of resource management functions defined in TMN standards that consists of functions that manage individual network elements or subsets of network elements (which may be viewed by network management layer functions as subnetworks).

Federation
: An organizational structure involving two or more autonomous administrations in which the member administrations have an agreement on how they will interwork with each other including the extent to which the resources of one member can be shared by other members.

Flow Connection
: Abstract class that models transport between flow end points.

Flow End Point
: Abstract class that models the termination of a flow connection.

Functional Area
: A task-specific grouping of required network management functions. The OSI defines five management functional areas. The TINA-C architecture defines six TINA functional areas by dividing the OSI Configuration Management functional area into Resource Configuration and Connection Management. The six TINA functional areas are: Account-

ing Management, Connection Management, Fault Management, Performance Management, Resource Configuration Management, and Security Management.

Layer Network
: A set of transport functions which support the transfer of information of a characteristic type. Generally, a layer network is closely tied to a specific type of network transmission and/or switching technology, e.g., SDH/SONET VC-4, ATM virtual channel (ATM VC) or ATM virtual path (ATM VP).

Layer Network Coordinator
: A computational object responsible for providing trails in a layer network. It is associated with one domain in the layer network and federates with other Layer Network Coordinators to provide a trail across domain boundaries.

Link Connection
: An connectivity which runs between a pair of Subnetwork.

Link Termination Point
: A termination point of a Link.

Managed Object
: An abstract representation of a resource that can be supervised and controlled by other objects.

Network Element Layer
: The category of functions defined in TMN standards that are linked to the technology or architecture of the network resources that provide the basic telecommunications services. These functions may be accessed by the element management layer functions using standard or open information specifications that may hide vendor-specific functions within network resources.

Network Flow Connection
: A point-to-point unidirectional, point-to-point bidirectional or a point-to-multipoint unidirectional flow connection between network flow end points.

Network Flow End Point
: A network level termination point of a network flow connection, that is always related to a network termination point.

Network Flow End Point pool
: An aggregation of resource flow end points.

Network Management Layer
: A sublayer of network resource management functions defined in TMN standards that have the responsibility for the management of all the network elements, as presented by the element management layer. It is not concerned with how a particular network element provides service internally. Complete visibility of the whole network is typical, and a

vendor independent view will need to be maintained. The functions in this layer interact with the service management layer on end-to-end connections, performance, faults, etc. across the network.

Notification
: A management operation initiated by a managed object for the purpose of communicating the occurrence of some significant event within the managed object.

Resource Flow End Point
: Abstract class from which network flow end points and network flow end point pools are derived.

Server
: Defined relative to an operational interface; the object that provides an operational interface is the server of the interface.

Stream Flow Connection
: A uni-directional point-to-point or point-to-multi-point flow connection between stream flow end points.

Stream Flow End Point
: A uni-directional (source or sink) application level end point of a stream flow connection, which is always aggregated in a stream interface

Stream Interface
: An abstraction that represents a communication endpoint that may be a source for some stream flows and a sink for some stream flows.

Subclass (1)
: One class is a subclass of another class precisely when it is a subset of the other class.

Subclass (2)
: A object class which inherits the template of another class.

Subnetwork
: A subset of the network resources such that the resources, having common operations properties (e.g., manufacturer, common function, or common geographical location) cooperate to support some aspect or portion of one or more telecommunications services. It may contain resources of different suppliers, and may consist of several nodes that are operated as a cohesive entity. In the context of Connection Management the subnetwork is used as a topological component to effect routing and management. It can be partitioned into interconnected subnetworks and connections.

Subnetwork Connection
: A transport entity formed by a connection across a subnetwork between termination points.

Subordinate
: A Managed Object instance which is placed below its Superior Managed Object in the Containment relationship tree.

Superclass (1)
: One class is a superclass of another class precisely when the other class is a subset of it.

Superclass (2)
: A object class whose template is inherited by another class.

Superior
: A Managed Object instance which is placed above its Subordinate Managed Object.

Terminal Flow Connection
: A point-to-point uni-directional flow connection between a stream flow end point and another stream flow end point or a network flow endpoint.

Topological Link
: Collection of Link Connections which are served by a trail of a server layer network.

TP Pool
: A collection of termination points that is used for some management purpose such as routing.

Trail
: A transport entity which spans across a Layer Network.

Trail Termination Point
: A termination point of a Trail.

14.Annex: ODL-specs

This section gives the current ODL specifications for the interfaces and operations described in previous sections.

14.1 ConnectionCoordinator.odl

```
/** ConnectionCoordinator.odl           */
/**                                     */
/**                                     */
/**                                     */
/** Author: Frank Steegmans (Alcatel)   */
/** Creation date: Dec. 4th, 1997       */
/** Revised: 12-04-1997 by Takeo Hamada */
/** Reviewed:                           */

// ODL of object template ConnectionCoordinator

object ConnectionCoordinator {

    behavior
        "   One such object exists for each connectivity
            session that has been setup. This object is
            created by the Connection CoordinatorFactory
            object. This object offers two interfaces:
            an interface, i_ConnSessionControl, that provides
            operations          for manipulating the connectivity session;
            an interface, i_ConnSessionNotificationControl, that
            provides flow connection notification control at
            connectivity session level.

            This object serves as the factory object for
            flow connections that are components of the
            associated connectivity session.
            The capabilities offered by this object are
            parts of the Connectivity Control Service.
            This object is deleted when the associated
            connectivity session is released using an
            operation defined in the i_ConnSessionControl
            interface.";


    supports
        i_ConnSessionControl, i_ConnSessionNotificationControl;

    initial
        i_ConnSessionControl;

};
```

14.2 ConnectionCoordinatorFactory.odl

```
/** ConnectionCoordinatorFactor.odl     */
/**                                     */
```

```
/**                              */
/**                              */
/** Author: Frank Steegmans (Alcatel)   */
/** Creation date: Dec. 4th, 1997       */
/** Revised: 12-04-1997 by Takeo Hamada */
/** Reviewed:                    */

// ODL of object template ConnectionCoordinatorFactory

object ConnectionCoordinatorFactory {

      behavior
            "     This object is the factory object for connectivity sessions.
                  It offers an interface, i_ConnSessionSetup, that provide an
                  operation for connectivity session setup,an operation for
                  listing all connectivity sessions belonging to a CU, and
                  an operation for obtaining references to i_ConnSessionControl
                  and i_ConnSessionNotificationControl interfaces for
                  controlling a specific connectivity session. The capabilities
                  offered by this object are                 parts of the Connectivity Control
                  Service.
            ";


      supports
            i_ConnSessionSetup;

      initial
            i_ConnSessionSetup;

};
```

14.3 ConsUserAgent.odl

```
/** ConsUserAgent.odl                 */
/**                              */
/**                              */
/**                              */
/** Author: Frank Steegmans (Alcatel)   */
/** Creation date: Dec. 4th, 1997       */
/** Revised: 12-04-1997 by Takeo Hamada */
/** Reviewed:                    */

// ODL of object template ConsUserAgent

object ConsUserAgent {

      behavior
            "     This object represents a connectivity user (CU)
                  that has setup a business relationship with the
                  connectivity provider (CP). This object is created
                  when the business relationship is setup and exists
                  as long as the business relationship exists. It
                  provides an interface called Cons_UA_Access that
                  provides operations for establishing and controlling
                  service sessions in this reference point, i.e.,
```

```
                Connectivity service sessions.";


      supports
            i_ConsUAAccess;

      initial
            i_ConsUAAccess;

};
```

14.4 ContractProfileManager.odl

```
/** ContractProfileManager.odl          */
/**                                     */
/**                                     */
/**                                     */
/** Author: Frank Steegmans (Alcatel)   */
/** Creation date: Dec. 4th, 1997       */
/** Revised: 12-04-1997 by Takeo Hamada */
/** Reviewed:                           */

// ODL of object template ContractProfileManager

object ContractProfileManager {

      behavior
            "     This object manages the contractProfile information
                  associated with a Connectivity User. It offers an
                  interface i_ContractProfileMgmnt that provides
                  operations for the retrieval and modification of
                  contract profile information.";


      supports
            i_ContractProfileMgmnt;

      initial
            i_ContractProfileMgmnt;

};
```

14.5 FlowConnectionController.odl

```
/** FlowConnectionController.odl        */
/**                                     */
/**                                     */
/**                                     */
/** Author: Frank Steegmans (Alcatel)   */
/** Creation date: Dec. 4th, 1997       */
/** Revised: 12-04-1997 by Takeo Hamada */
/** Reviewed:                           */

// ODL of object template FlowConnectionController
```

```
object FlowConnectionController {

    behavior
        "    One such object exists for each flow connection
             that has been setup. This object is created by a
             ConnectionCoordinator object. This object offers
             an interface, Flow_Conn_Control, that provides
             operations for manipulating the associated flow
             connection. The capabilities offered by this
             object are parts of the Connectivity Control
             Service.

             This object is deleted when the associated flow
             connection is released using an operation defined
             in the i_FlowConnControl interface. If the
             operational state of the associated flow connection
             changes, this object uses an interface,
             i_FlowConnNotification, offered by a CU, and
             reports the operational state change to the CU.
             This object also offers another interface,
             i_FlowConnNotificationControl, that can be used by
             CUs to control the emission of operational state
             change notifications.";


    supports
        i_FlowConnControl, i_FlowConnNotificationControl;

    initial
        i_FlowConnNotificationControl;

};
```

14.6 InitialAgent.odl

```
/** InitialAgent.odl                */
/**                          */
/**                          */
/**                          */
/** Author: Frank Steegmans (Alcatel)   */
/** Creation date: Dec. 4th, 1997       */
/** Revised: 12-04-1997 by Takeo Hamada */
/** Reviewed:                       */

// ODL of object template InitialAgent

object InitialAgent {

    behavior
        "    This object is the initial access point to a
             Connectivity Provider's domain. It provides
             the function of authenticating a Connectivity
             User. It offers an interface i_ConsInitialAccess
             that provides the authentication function.";
```

```
    supports
        i_ConsInitialAccess;

    initial
        i_ConsInitialAccess;

};
```

15.Annex: IDL-specs

This section gives the current IDL specifications for the interfaces and operations described in previous sections.

15.1 CLNCommonDefs.idl

```
/*
 *    File:         CLNCommonDefs.idl
 *
 *    Description:    This file contains all the idl definitions common to all layers
 *                within a connectivity layer network
 *
 *    Comments:      -
 *
 *    History:
 *
 *                97/08/14: Initial Contribution
 *                      by Frank Steegmans
 *                97/12/04: Revised by Takeo Hamada
 *
 */



#ifndef i_CLNCommonDefs_IDL
#define i_CLNCommonDefs_IDL


#include "NRACommonDefs.idl"
#include "naming.idl"
#include "States.idl"
#include "nfep.idl"

enum t_ActivationStatus {Activated, UnableToActivate, Deactivated,
                    UnableToDeactivate};

interface i_CLNCommonDefs : i_NRACommonDefs
{

  /*
   * Flow endpoint related definitions
   */

  typedef      m_NFEP::t_NfepName      t_NfepRef;
  typedef      m_NFEP::t_NfepNameList  t_NfepRefList;

  enum  t_NfepUse              {Root, Leaf};
  enum  t_NfepResolveState      {Partial, Full};


  /*
   * Flow endpoint information
   */

  struct t_NfepInfo
```

```
{
  m_NFEP::t_ANfep            nfepDescription;
  t_NfepUse                  nfepUsedAs;
  m_STATE::t_ManagementState  state;
};

struct t_NfepStatusInfo
{
  t_NfepRef                  ref;
  t_NfepResolveState         resolveState;
  m_NFEP::t_ANfep            nfep;
  m_STATE::t_OperationalState operationalState;
  string                     additionalInfo;
};

typedef sequence <t_NfepStatusInfo>   t_nfepStatusInfoSeq;


/*
 * Operation request-response related definitions
 */

enum t_ParametersTag { AdminState, RelClass };

union t_ParameterValue
  switch(t_ParametersTag)
    {
    case AdminState:  m_STATE::t_AdministrativeState  state;
    case RelClass:    t_ReliabilityClass              rclass;
    };

typedef sequence <t_ParameterValue> t_ParametersList;


typedef sequence <t_NfepRef> t_SuccNfepList;


enum  t_FailureCode {InsufficientBandwidth,
                  InsufficientResources,
                  QoSCannotBeMet,
                  NoPathFound,
                  NetworkFailure,
                  KtnFailure};
/*
 * Note:
 * It is not clear how 'Insufficient bandwidth' and 'QoS can not be met'
 * should be treated based on the new LNW TP internal specifications of
 * these parameters.
 */

struct t_FailedNfep
{
  t_NfepRef ref;
  t_FailureCode code;
};
typedef sequence <t_FailedNfep> t_FailedNfepList;
```

```
 struct t_NfepDesc
 {
   t_NfepRef            ref;
   m_NFEP::t_ANfep      nfepDescription;
   t_NfepUse            use;
   t_ParametersList     list;   // Used for specifying only the admin state
   t_TinaName           tfcBranchRef; // relates resolved nfep with tfc
 };
 typedef sequence <t_NfepDesc> t_NfepDescSeq;

 /*
  * Commonly used exceptions (in alphabetical order)
  */

 exception BranchesActiveAlready {
   t_NfepRefList list;
 };

 exception BranchesDeactiveAlready {
   t_NfepRefList list;
 };

 exception NetworkFlowEndPointsAlreadyBound {
   t_NfepRefList list;
 };

 exception InvalidDefaultValues { };

 exception InvalidBranchesInfo {
   t_NfepRefList       list;
   string              info;
 };

 exception NonExistentFlowEndPoints {
   t_NfepRefList list;
 };

 exception NotificationDestinationNotSet { };


 struct t_NfepActivationResponse {
   t_NfepRef ref;
   t_ActivationStatus status;
 };

 typedef sequence <t_NfepActivationResponse> t_NfepActivationResponseSeq;

};

#endif /* i_LayerNetworkCommonDefs_IDL */
```

15.2 ComSCommonDefs.idl

```
/*
 *    File:        ComSCommonDefs.idl
 *
 *    Description:    This file contains all the idl definitions common to the
```

```
 *          Communication Session.
 *
 *    Comments:    -
 *
 *    History:
 *
 *          97/08/14: Initial Contribution
 *                by Frank Steegmans
 *          97/12/04: Revised by Takeo Hamada
 *
 */


#ifndef i_ComSCommonDefs_IDL
#define i_ComSCommonDefs_IDL


#include "NRACommonDefs.idl"
#include "naming.idl"
#include "States.idl"
#include "nfep.idl"


interface i_ComSCommonDefs : i_NRACommonDefs
{
  typedef string t_SFCName;
  typedef sequence<t_SFCName> t_SFCNameList;

};

#endif /* i_ComSCommonDefs_IDL */
```

15.3 Common.idl

```
#ifndef _COMMON_IDL_
#define _COMMON_IDL_

// Generic Types


typedef string t_IntRef; // "stringified" Interface Reference.
typedef sequence <t_IntRef> t_IntRefList;

typedef string t_ApplicationInfo;

typedef string t_TermId;
// Terminal Identifier. This is used for e.g. personal mobility.
// This could contain the hostname of the terminal
typedef sequence <t_TermId> t_TermIdList;

enum t_Topology { PointToPoint, PointToMultiPoint, Broadcast };
// in VitalV2  topol will be always ptmp

enum t_SuccessCriterion {BestEffort, AllOrNone};

typedef string t_SFep_CId ;
```

```
typedef sequence <t_SFep_CId> t_SFep_CIdList ;

struct t_SFepRef
{
 t_IntRef      tcsmRef;
 t_SFep_CId    SFepId;
};
typedef sequence<t_SFepRef> t_SFepRefList;

enum t_SFepDirection {SFepSource, SFepSink};

exception Error {t_ApplicationInfo            applicationInfo;};

#endif
```

15.4 ConnectionPerf.idl

```
//
// Connection Performer Specific Interfaces
// OMG IDL ConnectionPerf Module
//
// Written by Wataru Takita, 1997
//
// History:
// 97/08/09 modification by Frank Steegmans
//      Minor syntactical corrections for compilation
// 97/08/17 modification by Wataru Takita
//      Correction of improper modification on 97/08/09
//

#include <NrimObjectConf.idl>

module ConnectionPerf {

typedef any NrimInstanceName;
typedef sequence<NrimInstanceName> NrimInstanceNames;
typedef string  NrimAttributeName;
typedef sequence<NrimAttributeName> NrimAttributeNames; // Added by fste for compilation
typedef unsigned short ExceptionReason;

struct NrimAttribute {
      NrimAttributeName       nrim_attribute_name;
      any                     nrim_attribute_value;
};
typedef sequence<NrimAttribute> NrimAttributes;
struct NrimInstanceException {
      ExceptionReason         reason;
      NrimInstanceName        nrim_instance_name;
};

struct NrimAttributeException {
      ExceptionReason         reason;
      NrimAttributeName       nrim_attribute_name;
};

typedef sequence<NrimInstanceException> NrimInstanceExceptions;
typedef sequence<NrimAttributeException> NrimAttributeExceptions;
```

```
exception MultipleNrimInstanceExceptions {
      NrimInstanceExceptions          exceptions;
};

exception MultipleNrimAttributeExceptions {
      NrimAttributeExceptions         exceptions;
};
//
// Virtual Interface Definitions
//

interface SubNetworkVirtual {

//
// Data Types
//

/* The 'AdministrativeState_t' definition in NRIM is different from
X.721:1992. Perhaps, some people might prefer the following commented
definition.
      enum AdministrativeState_t {
            locked,
            unlocked,
            shutingDown
      };
*/

      enum AdministrativeState_t {
            locked,
            unlocked
      };

/* The 'OperationalState_t' definition in NRIM is different from
X.721:1992. Perhaps, some people might prefer the following commented
definition.
      enum OperationalState_t {
            disabled,
            enabled
      };

*/

      enum OperationalState_t {
            failed,
            operational,
            degraded
      };

/* The 'ConnectionTopology_t' definition in NRIM is different from the
ways of G.853-01 and G.854-01. Perphaps, the following definition is
better for the people who really love ITU-T standards.
      enum Directionality {
            unidirectional,
            bidirectional
      };
*/
```

```
    enum ConnectionTopology_t {
        pt_pt_unidirectional, // '-' --> '_' by fste
        pt_pt_bidirectional,
        pt_multipt_unidirectional
    };

/* 'UserLabel' is defined in both ATMF M4 and ITU-T G.85x. It is used
for put a nick name to SubNetworkConnection. It is not a global unique
name, but should be unique in a SubNetwork. */

    typedef string  UserLabel;

//
// Exceptions
//

/* 'Topology Error' is raised at the time conflicting specified SNC
topology with Edge directionality.*/

    exception InvalidNwctpName {};
    exception NwctpNotFound {};
    exception NwctpIsInUse {};
    exception InvalidTopology {};
    exception TopologyError {};
    exception InvalidSncIdentifier {};
    exception SncNotFound {};
    exception SncIsInUse {};
    exception InvalidEdgeName {};
    exception EdgeNotFound {};
    exception EdgeIsInUse {};
    exception InvalidUserLabel {};
    exception UserLabelIsInUse {};
};

//
// Interface Definitions
//

interface SncServiceFactory:
    NrimObjectConf::NrimInstanceVirtual, SubNetworkVirtual {
//
// Data Types
//
    enum SncIdentifierType {
        label,
        if_ref
    };

    union SncIdentifier switch (SncIdentifierType) {
        case label :    UserLabel       user_label;
        case if_ref :   Object          snc_service;
    };
//
// Exceptions
//
```

```
        const ExceptionReason   invalid_edge_name = 401;
        const ExceptionReason   edge_not_found = 402;
        const ExceptionReason   edge_is_in_use = 403;
        const ExceptionReason   topology_error = 403;

        struct  EdgeNamedException {
              ExceptionReason         reason;
              NrimInstanceName        edge_name;
        };

        typedef sequence<EdgeNamedException>    EdgeNamedExceptions;

        exception MultipleEdgeExceptions {
              EdgeNamedExceptions     exceptions;
        };
//
// Attributes
//
        readonly attribute      NrimInstanceName        subnetwork;
        attribute               AdministrativeState_t   administrative_state;
        readonly attribute      OperationalState_t      operational_state;
//
// Operations
//
        void    create_snc(
              in     NrimInstanceName        root_edge_name,
              in     ConnectionTopology_t    connection_topology,
              in     UserLabel               supplied_user_label,
              in     NrimAttributes          nrim_attributes,
              out    Object                  snc_service,
              out    UserLabel               agreed_user_label
        )raises(
              InvalidEdgeName,
              EdgeNotFound,
              EdgeIsInUse,
              InvalidTopology,
              TopologyError,
              InvalidUserLabel,
              UserLabelIsInUse,
              MultipleNrimAttributeExceptions
        );

/* The exceptions ('InvalidEdgeName', 'EdgeNotFound', 'EdgeInUse', '
TopologyError' are raised for the errors related to the specified root
edge name. The errors for leaf edges is raised with '
MultipleEdgeExceptions'. */

        void    create_setup_snc(
              in     NrimInstanceName        root_edge_name,
              in     NrimInstanceNames       leaf_edge_names,
              in     ConnectionTopology_t    connection_topology,
              in     UserLabel               supplied_user_label,
              in     NrimAttributes          nrim_attributes,
              out    Object                  snc_service,
              out    UserLabel               agreed_user_label
        )raises(
              InvalidEdgeName,
```

```
            EdgeNotFound,
            EdgeIsInUse,
            TopologyError,
            InvalidTopology,
            MultipleEdgeExceptions,
            InvalidUserLabel,
            UserLabelIsInUse,
            MultipleNrimAttributeExceptions
        );

        void    delete_snc(
            in      SncIdentifier   snc_identifier
        )raises(
            InvalidSncIdentifier,
            SncNotFound,
            SncIsInUse
        );

/* The return of the following operation is a edge name. */

        NrimInstanceName        create_edge(
            in      NrimInstanceName        nwctp_name,
            in      NrimAttributes          nrim_attributes
        )raises(
            InvalidNwctpName,
            NwctpNotFound,
            NwctpIsInUse,
            MultipleNrimAttributeExceptions
        );

        void    delete_edge(
            in      NrimInstanceName        edge_name
        )raises(
            InvalidEdgeName,
            EdgeNotFound,
            EdgeIsInUse
        );

};

interface SncService:
        NrimObjectConf::NrimInstanceVirtual, SubNetworkVirtual {
//
// Attributes
//
        readonly attribute      NrimInstanceName        snc_name;
        attribute               AdministrativeState_t   administrative_state;
        readonly attribute      OperationalState_t      operational_state;
        readonly attribute      ConnectionTopology_t    connection_topology;
        readonly attribute      UserLabel               user_label;

/* NRA 3.0 specifies the following attributes.  Are they really
meaningful for logical description?
        attribute       any     traffic_description;
        attribute       any     qos;
*/
```

```
/* Only leaf edges can be attached to SNC or detached from SNC. A root
edge cannot be attached or detached with the following operations */

    void    attach_edge(
        in      NrimInstanceName        edge_name
    )raises(
        InvalidEdgeName,
        EdgeNotFound,
        EdgeIsInUse,
        TopologyError
    );

    void    detach_edge(
        in      NrimInstanceName        edge_name
    )raises(
        InvalidEdgeName,
        EdgeNotFound,
        EdgeIsInUse
    );

/* The return of the following operation is a migrated edge name. */

    NrimInstanceName        migrate_edge(
        in      NrimInstanceName        edge_name,
        in      NrimInstanceName        nwctp_name
    )raises(
        InvalidNwctpName,
        NwctpNotFound,
        NwctpIsInUse,
        InvalidEdgeName,
        EdgeNotFound,
        EdgeIsInUse,
        TopologyError,
        MultipleNrimAttributeExceptions
    );
};

}; // End of Module
```

15.5 NRACommonDefs.idl

```
/*
 *    File:         NRACommonDefs.idl
 *
 *    Description:    This file contains all the idl definitions common to the
 *                whole Network Resource Architecture
 *
 *    Comments:      -
 *
 *    History:
 *
 *                97/08/14: Initial Contribution
 *                      by Frank Steegmans
 *
 */
```

```
#ifndef i_NRACommonDefs_IDL
#define i_NRACommonDefs_IDL

#include "naming.idl"

interface i_NRACommonDefs
{

  /*
   * Flow Connection related information
   */

  enum  t_ConnTopology         {PointToPoint, PointToMultipoint};
  enum  t_ReliabilityClass     {ReleaseOnFailure, HoldOnFailure};
  enum  t_SuccessCriterion     {AllOrNone, BestEffort};

  //enum       t_RoutingOption        {SameRoute, DifferentRoute};
  // Hyun
  enum  t_RoutingOption        {SameRoute, DifferentRoute, SourceRoute};
//  string useSpecificDomain;
// string dontUseSpecificDomain;
//  unsigned long maxNumberOfHops;
//  string useSpecificNwTp;
//  string dontUseSpecificNwTp;

  typedef t_TinaName t_NetworkFlowConnectionName;
  typedef sequence <t_NetworkFlowConnectionName> t_NetworkFlowConnectionNameSeq;

};

#endif /* i_NRACommonDefs_IDL */
```

15.6 NrimObjectConf.idl

```
#ifndef _NrimObjectConf_idl_
#define _NrimObjectConf_idl_

//
// NRIM Object Configuration
// OMG IDL NrimObjectConf Module
//
// Written by Wataru Takita, 1997
//
// History:
// 97/08/09 modifications by Frank Steegmans
//     Correction of syntactical errors for compilation
// 97/08/17 modification by Wataru Takita
//     Correction of improper modification on 97/08/09
// 97/12/09 couldn't be compiled by hidl, due to virtual
//     interface inheritance?
// 97/12/10 hashes hidl are used to compile it - virtual
//     interface inheritance is got around.

module NrimObjectConf {
```

```
//
// Virtual Interface Definitions
//

interface NrimTypeVirtual {

//
// DataTypes
//

/* In line with TINA Naming, the use of 'sequence<unsigned short>' or
'Naming::Name' is recommended for the actual type of 'NrimTypeName'. If
you like, however, 'sequence<string>' is also possible. */

    typedef any NrimTypeName;
    typedef sequence<NrimTypeName> NrimTypeNames;
//
// Exceptions
//
    exception InvalidTypeName {};
    exception UnsupportedType {};

};

interface NrimInstanceVirtual {

//
// Data Types
//

/* According to TINA Naming, the use of 'string' is recommended for
the actual type of 'NrimInstanceName', stringfied RDN/DN. */

    typedef any NrimTypeName;
    typedef any NrimInstanceName;
    typedef string  NrimAttributeName;
    typedef sequence<NrimAttributeName> NrimAttributeNames; // Added by fste for compilation

    struct NrimAttribute {
        NrimAttributeName       nrim_attribute_name;
        any                     nrim_attribute_value;
    };

/* Alternative type definition of 'NrimAttribute' might be "string
NrimAttribute". This definition seems nice for interoperability but
imposes additional efforts for implementation, e.g., parser. If you
love 'yacc' or 'bison', perhaps this thing is not a problem */

    typedef sequence<NrimInstanceName> NrimInstanceNames;
    typedef sequence<NrimAttribute> NrimAttributes;
//
// Exceptions
//
    exception InvalidInstanceName {};
    exception InstanceNotFound {};
    exception InstanceIsInUse {};
```

```
        exception InvalidAttributeName {};
        exception InvalidAttributeVaule {};
        exception UnsupportedAttribute {};
        exception ReadOnlyAttribute {};
        exception FixedAttribute {};
        exception AttributeValueConfliction {};
//
// For Multiple Exceptions
//

/* To avoid difficulties in debugging with a poor debugger, 'const'
declaration is used instead of 'enum'. It seems easy for human beings
to distinguish the difference between '101' and '201' rather than '0'
and '0'. */

        typedef unsigned short  ExceptionReason;

        const ExceptionReason   invaild_instance_name = 101;
        const ExceptionReason   instance_not_found = 102;
        const ExceptionReason   instance_is_in_use = 103;

        const ExceptionReason   invalid_attribute_name = 201;
        const ExceptionReason   invalid_attribute_value = 202;
        const ExceptionReason   unsupported_attribute = 203;
        const ExceptionReason   readonly_attribute = 204;
        const ExceptionReason   fixed_attribute = 205;
        const ExceptionReason   attribute_value_confliction = 206;

        struct NrimInstanceException {
                ExceptionReason         reason;
                NrimInstanceName        nrim_instance_name;
        };

        struct NrimAttributeException {
                ExceptionReason         reason;
                NrimAttributeName       nrim_attribute_name;
        };

        typedef sequence<NrimInstanceException> NrimInstanceExceptions;
        typedef sequence<NrimAttributeException> NrimAttributeExceptions;

        exception MultipleNrimInstanceExceptions {
                NrimInstanceExceptions          exceptions;
        };

        exception MultipleNrimAttributeExceptions {
                NrimAttributeExceptions         exceptions;
        };

};

//
// Interface Definitions
//

interface NrimInstanceFactory: NrimTypeVirtual, NrimInstanceVirtual {
```

```
#ifdef hidl
    typedef any NrimTypeName;
    exception InvalidTypeName {};
    exception UnsupportedType {};
#endif

    boolean       is_supported (
        in     NrimTypeName    nrim_type_name
    );

    NrimInstanceName       instantiate (
        in     NrimTypeName    nrim_type_name,
        in     NrimAttributes  nrim_attributes
    ) raises (
        InvalidTypeName,
        UnsupportedType,
        MultipleNrimAttributeExceptions,
        AttributeValueConfliction
    );
};

interface NrimInstanceConfigurator: NrimInstanceVirtual {

    void    set_attribute(
        in     NrimInstanceName       nrim_instance_name,
        in     NrimAttribute          nrim_attribute
    )raises(
        InvalidInstanceName,
        InstanceNotFound,
        InstanceIsInUse,
        InvalidAttributeName,
        InvalidAttributeVaule,
        UnsupportedAttribute,
        ReadOnlyAttribute,
        FixedAttribute,
        AttributeValueConfliction
    );

    void    set_attributes(
        in     NrimInstanceName       nrim_instance_name,
        in     NrimAttributes         nrim_attributes
    )raises(
        InvalidInstanceName,
        InstanceNotFound,
        InstanceIsInUse,
        MultipleNrimAttributeExceptions
    );

    void    delete_instance(
        in     NrimInstanceName       nrim_instance_name
    )raises(
        InvalidInstanceName,
        InstanceNotFound,
        InstanceIsInUse
    );
};
```

```
interface NrimInstanceQuery: NrimInstanceVirtual {

      any  get_attribute(
            in     NrimInstanceName       nrim_instance_name,
            in     NrimAttributeName      nrim_attribute_name
      )raises(
            InvalidInstanceName,
            InstanceNotFound,
            InvalidAttributeName,
            UnsupportedAttribute
      );

      boolean get_attributes(
            in     NrimInstanceName       nrim_instance_name,
            in     NrimAttributeNames     nrim_attribute_names,
            out    NrimAttributes         nrim_attributes
      )raises(
            InvalidInstanceName,
            InstanceNotFound,
            MultipleNrimAttributeExceptions
      );

 };


interface NrimTypeAdmin: NrimTypeVirtual, NrimInstanceVirtual {

//
// Data Types
//
      enum NrimAttributeMode {
            read_write,
            read_only,
            fixed
      };

/* The following 'nrim_attribute_value_type is defined as
"string". The real content of this argument is beyond the scope of
this specification. Perhaps, string form of "short" or stringfied
'CORBA::TypeCode' will likely be used. To define this argument as 'any'
instead of 'string' is also not so bad idea. */

      struct NrimAttributeDef {
            NrimAttributeName       nrim_attribute_name;
            string                  nrim_attribute_value_type;
            NrimAttributeMode       nrim_attribute_mode;
      };

      typedef sequence<NrimAttributeDef> NrimAttributeDefs;

/* 'NrimTypePolicy' will be used to specify the parameters related to
administrative policies, e.g., maximum number of instances. */

      struct NrimTypePolicy {
            string  policy_name;
            any     policy_value;
      };
```

```
    typedef sequence<NrimTypePolicy> NrimTypePolicies;
//
// Exceptions
//
    exception InvalidPolicyName {};
    exception UnsupportedPolicy {};
    exception InvalidPolicyValue {};
//
// Operations
//

#ifdef hidl
    typedef any NrimTypeName;
    typedef sequence<NrimTypeName> NrimTypeNames;
    exception InvalidTypeName {};
    exception UnsupportedType {};
#endif

    NrimInstanceNames    get_instances(
        in    NrimTypeName    nrim_type_name
    )raises(
        InvalidTypeName,
        UnsupportedType
    );

    NrimInstanceNames    list_all_intances();

    void    describe_type(
        in    NrimTypeName          nrim_type_name,
        out   NrimAttributeDefs     nrim_attribute_defs,
        out   NrimTypePolicies      nrim_type_policies
    )raises(
        InvalidTypeName,
        UnsupportedType
    );

    NrimTypeNames   list_supported_types();

    void    set_type_policy(
        in    NrimTypeName    nrim_type_name,
        in    NrimTypePolicy  nrim_type_policy
    )raises(
        InvalidTypeName,
        UnsupportedType,
        InvalidPolicyName,
        UnsupportedPolicy,
        InvalidPolicyValue
    );
};

}; //End of Module

#endif
```

15.7 NrimRelConf.idl

```
#ifndef _NrimRelConf_idl_
#define _NrimRelConf_idl_

//
// NRIM Relationship Configuration
// OMG IDL NrimRelConf Module
//
// Written by Wataru Takita, 1997
//

#include <NrimObjectConf.idl>

module NrimRelConf {

//
// Vitual Interface Definitions
//

interface NrimRelationshipVirtual {

//
// DataTypes
//
    typedef string          NrimRoleTypeName;
    typedef unsigned long   NrimRoleIdentifier;

    typedef sequence<NrimRoleTypeName>      NrimRoleTypeNames;
    typedef sequence<NrimRoleIdentifier>    NrimRoleIdentifiers;
//
// Exceptions
//
    exception InvalidRelationshipName {};
    exception RelationshipNotFound {};
    exception RelationshipIsInUse {};
    exception InvalidRoleTypeName {};
    exception UnsupportedRoleType {};
    exception InvalidRoleIdentifier {};
    exception RoleNotFound {};
    exception RelatedTypeError {};
    exception CardinalityError {};
    exception UnsupportedObjectType {};
    exception RoleNotAttached {};
    exception RoleIsAttached {};

};

//
// Interface Definition
//

interface NrimCompoundFactory:
    NrimObjectConf::NrimInstanceFactory, NrimRelationshipVirtual {

#ifdef hidl
/* In line with TINA Naming, the use of 'sequence<unsigned short>' or
```

```
'Naming::Name' is recommended for the actual type of 'NrimTypeName'. If
you like, however, 'sequence<string>' is also possible. */

      typedef any NrimTypeName;
      typedef sequence<NrimTypeName> NrimTypeNames;
//
// Exceptions
//
      exception InvalidTypeName {};
      exception UnsupportedType {};

/* According to TINA Naming, the use of 'string' is recommended for
the actual type of 'NrimInstanceName', stringfied RDN/DN. */

//    typedef any NrimTypeName;
      typedef any NrimInstanceName;
      typedef string  NrimAttributeName;
      typedef sequence<NrimAttributeName> NrimAttributeNames; // Added by fste for compilation

      struct NrimAttribute {
            NrimAttributeName      nrim_attribute_name;
            any                    nrim_attribute_value;
      };

/* Alternative type definition of 'NrimAttribute' might be "string
NrimAttribute". This definition seems nice for interoperability but
imposes additional efforts for implementation, e.g., parser. If you
love 'yacc' or 'bison', perhaps this thing is not a problem */

      typedef sequence<NrimInstanceName> NrimInstanceNames;
      typedef sequence<NrimAttribute> NrimAttributes;
//
// Exceptions
//
      exception InvalidInstanceName {};
      exception InstanceNotFound {};
      exception InstanceIsInUse {};

      exception InvalidAttributeName {};
      exception InvalidAttributeVaule {};
      exception UnsupportedAttribute {};
      exception ReadOnlyAttribute {};
      exception FixedAttribute {};
      exception AttributeValueConfliction {};
//
// For Multiple Exceptions
//

/* To avoid difficulties in debugging with a poor debugger, 'const'
declaration is used instead of 'enum'. It seems easy for human beings
to distinguish the difference between '101' and '201' rather than '0'
and '0'. */

      typedef unsigned short  ExceptionReason;

      const ExceptionReason   invaild_instance_name = 101;
      const ExceptionReason   instance_not_found = 102;
```

```
        const ExceptionReason   instance_is_in_use = 103;

        const ExceptionReason   invalid_attribute_name = 201;
        const ExceptionReason   invalid_attribute_value = 202;
        const ExceptionReason   unsupported_attribute = 203;
        const ExceptionReason   readonly_attribute = 204;
        const ExceptionReason   fixed_attribute = 205;
        const ExceptionReason   attribute_value_confliction = 206;

        struct NrimInstanceException {
                ExceptionReason         reason;
                NrimInstanceName        nrim_instance_name;
        };

        struct NrimAttributeException {
                ExceptionReason         reason;
                NrimAttributeName       nrim_attribute_name;
        };

        typedef sequence<NrimInstanceException> NrimInstanceExceptions;
        typedef sequence<NrimAttributeException> NrimAttributeExceptions;

        exception MultipleNrimInstanceExceptions {
                NrimInstanceExceptions          exceptions;
        };

        exception MultipleNrimAttributeExceptions {
                NrimAttributeExceptions         exceptions;
        };

#endif

//
// Exceptions
//
        const ExceptionReason   invalid_role_type_name = 301;
        const ExceptionReason   unsupported_role_type = 302;
        const ExceptionReason   related_type_confliction = 303;
        const ExceptionReason   cardinality_error = 304;
        const ExceptionReason   unsupported_object_type = 305;

        struct NrimRoleException {
                ExceptionReason         reason;
                NrimRoleTypeName        nrim_role_type_name;
                NrimInstanceName        nrim_instance_name;
        };

        typedef sequence<NrimRoleException>     NrimRoleExceptions;

        exception MultipleNrimRoleExceptions {
                NrimRoleExceptions      exceptions;
        };
//
// Operations
//

/* The return of the following operation is the instance name of
```

```
a created relationship. */

    NrimInstanceName        instantiate_relationship(
        in      NrimTypeName            nrim_relationship_type,
        in      NrimAttributes          nrim_relationship_attributes
    )raises(
        InvalidTypeName,
        UnsupportedType,
        MultipleNrimAttributeExceptions
    );

    void    setup_relationship(
        in      NrimTypeName            nrim_relationship_type,
        in      NrimAttributes          nrim_relationship_attributes,
        in      NrimRoleTypeNames       nrim_role_types,
        in      NrimInstanceNames       nrim_instance_names,
        out     NrimInstanceName        nrim_relationship_name,
        out     NrimRoleIdentifiers     nrim_role_identifiers
    )raises(
        InvalidTypeName,
        UnsupportedType,
        MultipleNrimAttributeExceptions,
        MultipleNrimRoleExceptions
    );
};

interface NrimCompoundConfigurator:
    NrimObjectConf::NrimInstanceConfigurator, NrimRelationshipVirtual {

#ifdef hidl
/* In line with TINA Naming, the use of 'sequence<unsigned short>' or
'Naming::Name' is recommended for the actual type of 'NrimTypeName'. If
you like, however, 'sequence<string>' is also possible. */

    typedef any NrimTypeName;
    typedef sequence<NrimTypeName> NrimTypeNames;
//
// Exceptions
//
    exception InvalidTypeName {};
    exception UnsupportedType {};

/* According to TINA Naming, the use of 'string' is recommended for
the actual type of 'NrimInstanceName', stringfied RDN/DN. */

//      typedef any NrimTypeName;
    typedef any NrimInstanceName;
    typedef string  NrimAttributeName;
    typedef sequence<NrimAttributeName> NrimAttributeNames; // Added by fste for compilation

    struct NrimAttribute {
        NrimAttributeName       nrim_attribute_name;
        any                     nrim_attribute_value;
    };

/* Alternative type definition of 'NrimAttribute' might be "string
NrimAttribute". This definition seems nice for interoperability but
```

```
imposes additional efforts for implementation, e.g., parser. If you
love 'yacc' or 'bison', perhaps this thing is not a problem */

	typedef sequence<NrimInstanceName> NrimInstanceNames;
	typedef sequence<NrimAttribute> NrimAttributes;
//
// Exceptions
//
	exception InvalidInstanceName {};
	exception InstanceNotFound {};
	exception InstanceIsInUse {};

	exception InvalidAttributeName {};
	exception InvalidAttributeVaule {};
	exception UnsupportedAttribute {};
	exception ReadOnlyAttribute {};
	exception FixedAttribute {};
	exception AttributeValueConfliction {};
//
// For Multiple Exceptions
//

/* To avoid difficulties in debugging with a poor debugger, 'const'
declaration is used instead of 'enum'. It seems easy for human beings
to distinguish the difference between '101' and '201' rather than '0'
and '0'. */

	typedef unsigned short  ExceptionReason;

	const ExceptionReason   invaild_instance_name = 101;
	const ExceptionReason   instance_not_found = 102;
	const ExceptionReason   instance_is_in_use = 103;

	const ExceptionReason   invalid_attribute_name = 201;
	const ExceptionReason   invalid_attribute_value = 202;
	const ExceptionReason   unsupported_attribute = 203;
	const ExceptionReason   readonly_attribute = 204;
	const ExceptionReason   fixed_attribute = 205;
	const ExceptionReason   attribute_value_confliction = 206;

	struct NrimInstanceException {
		ExceptionReason        reason;
		NrimInstanceName       nrim_instance_name;
	};

	struct NrimAttributeException {
		ExceptionReason        reason;
		NrimAttributeName      nrim_attribute_name;
	};

	typedef sequence<NrimInstanceException> NrimInstanceExceptions;
	typedef sequence<NrimAttributeException> NrimAttributeExceptions;

	exception MultipleNrimInstanceExceptions {
		NrimInstanceExceptions         exceptions;
	};
```

```
    exception MultipleNrimAttributeExceptions {
        NrimAttributeExceptions        exceptions;
    };

#endif

#ifdef hidl
//
// DataTypes
//
    typedef string          NrimRoleTypeName;
    typedef unsigned long   NrimRoleIdentifier;

    typedef sequence<NrimRoleTypeName>      NrimRoleTypeNames;
    typedef sequence<NrimRoleIdentifier>    NrimRoleIdentifiers;
//
// Exceptions
//
    exception InvalidRelationshipName {};
    exception RelationshipNotFound {};
    exception RelationshipIsInUse {};
    exception InvalidRoleTypeName {};
    exception UnsupportedRoleType {};
    exception InvalidRoleIdentifier {};
    exception RoleNotFound {};
    exception RelatedTypeError {};
    exception CardinalityError {};
    exception UnsupportedObjectType {};
    exception RoleNotAttached {};
    exception RoleIsAttached {};
#endif

//
// Operations
//
    NrimRoleIdentifier      attach_role(
        in      NrimInstanceName        nrim_relationship_name,
        in      NrimRoleTypeName        nrim_role_type_name,
        in      NrimInstanceName        nrim_instance_name
    )raises(
        InvalidRelationshipName,
        RelationshipNotFound,
        RelationshipIsInUse,
        InvalidRoleTypeName,
        UnsupportedRoleType,
        UnsupportedObjectType,
        InvalidInstanceName,
        InstanceNotFound,
        RelatedTypeError,
        CardinalityError,
        InstanceIsInUse
    );

    void    detach_role(
        in      NrimInstanceName        nrim_relationship_name,
        in      NrimRoleIdentifier      nrim_role_identifier
    )raises(
```

```
            InvalidRelationshipName,
            RelationshipNotFound,
            RelationshipIsInUse,
            InvalidRoleIdentifier,
            RoleNotFound,
            RoleNotAttached,
            CardinalityError,
            InstanceIsInUse
      );

      void    delete_relationship(
            in      NrimInstanceName        nrim_relationship_name
      )raises(
            InvalidRelationshipName,
            RelationshipNotFound,
            RelationshipIsInUse
      );

};

interface NrimCompoundQuery:
      NrimObjectConf::NrimInstanceQuery, NrimRelationshipVirtual {

#ifdef hidl
/* In line with TINA Naming, the use of 'sequence<unsigned short>' or
'Naming::Name' is recommended for the actual type of 'NrimTypeName'. If
you like, however, 'sequence<string>' is also possible. */

      typedef any NrimTypeName;
      typedef sequence<NrimTypeName> NrimTypeNames;
//
// Exceptions
//
      exception InvalidTypeName {};
      exception UnsupportedType {};

/* According to TINA Naming, the use of 'string' is recommended for
the actual type of 'NrimInstanceName', stringfied RDN/DN. */

//    typedef any NrimTypeName;
      typedef any NrimInstanceName;
      typedef string  NrimAttributeName;
      typedef sequence<NrimAttributeName> NrimAttributeNames; // Added by fste for compilation

      struct NrimAttribute {
            NrimAttributeName       nrim_attribute_name;
            any                     nrim_attribute_value;
      };

/* Alternative type definition of 'NrimAttribute' might be "string
NrimAttribute". This definition seems nice for interoperability but
imposes additional efforts for implementation, e.g., parser. If you
love 'yacc' or 'bison', perhaps this thing is not a problem */

      typedef sequence<NrimInstanceName> NrimInstanceNames;
      typedef sequence<NrimAttribute> NrimAttributes;
//
```

```
// Exceptions
//
	exception InvalidInstanceName {};
	exception InstanceNotFound {};
	exception InstanceIsInUse {};

	exception InvalidAttributeName {};
	exception InvalidAttributeVaule {};
	exception UnsupportedAttribute {};
	exception ReadOnlyAttribute {};
	exception FixedAttribute {};
	exception AttributeValueConfliction {};
//
// For Multiple Exceptions
//

/* To avoid difficulties in debugging with a poor debugger, 'const'
declaration is used instead of 'enum'. It seems easy for human beings
to distinguish the difference between '101' and '201' rather than '0'
and '0'. */

	typedef unsigned short  ExceptionReason;

	const ExceptionReason   invaild_instance_name = 101;
	const ExceptionReason   instance_not_found = 102;
	const ExceptionReason   instance_is_in_use = 103;

	const ExceptionReason   invalid_attribute_name = 201;
	const ExceptionReason   invalid_attribute_value = 202;
	const ExceptionReason   unsupported_attribute = 203;
	const ExceptionReason   readonly_attribute = 204;
	const ExceptionReason   fixed_attribute = 205;
	const ExceptionReason   attribute_value_confliction = 206;

	struct NrimInstanceException {
		ExceptionReason		reason;
		NrimInstanceName	nrim_instance_name;
	};

	struct NrimAttributeException {
		ExceptionReason		reason;
		NrimAttributeName	nrim_attribute_name;
	};

	typedef sequence<NrimInstanceException> NrimInstanceExceptions;
	typedef sequence<NrimAttributeException> NrimAttributeExceptions;

	exception MultipleNrimInstanceExceptions {
		NrimInstanceExceptions		exceptions;
	};

	exception MultipleNrimAttributeExceptions {
		NrimAttributeExceptions		exceptions;
	};

#endif
```

```
//
// Operations
//

/* The return of the following operation is a list of role identifiers
with the specified role type and bound to the specified
relationship. */

    NrimRoleIdentifiers    get_roles(
        in    NrimInstanceName     nrim_relationship_name,
        in    NrimRoleTypeName     nrim_role_type_name
    )raises(
        InvalidRelationshipName,
        RelationshipNotFound,
        InvalidRoleTypeName,
        UnsupportedRoleType
    );

/* The return of the following operation is a list of the other roles
in the specified relationship. */

    NrimRoleIdentifiers    get_other_related_roles(
        in    NrimInstanceName     nrim_relationship,
        in    NrimRoleIdentifier   nrim_role_identifier
    )raises(
        InvalidRelationshipName,
        RelationshipNotFound,
        InvalidRoleIdentifier,
        RoleNotFound
    );

/* The return of the following operation is a role type name of the
specified role. */

    NrimRoleTypeName       get_role_type (
        in    NrimInstanceName     nrim_relationship,
        in    NrimRoleIdentifier   nrim_role_identifier
    )raises(
        InvalidRelationshipName,
        RelationshipNotFound,
        InvalidRoleIdentifier,
        RoleNotFound
    );

/* The return of the following operation is a list of the relationship
names relavant to the specified NRIM object. */

    NrimInstanceNames      get_relationship(
        in    NrimInstanceName     nrim_instance_name
    )raises(
        InvalidInstanceName,
        InstanceNotFound
    );

};

}; // End of Module
```

```
#endif
```

15.8 PLATyToolsFix.idl

```
#ifndef PLATyToolsFix
#define PLATyToolsFix

//
// to get around problem of nested modules in
// RP0.7/TINACommonTypes.idl
//

module CosTrading {
	typedef string Istring;
	typedef Istring PropertyName;
	typedef sequence<PropertyName> PropertyNameSeq;
	typedef any PropertyValue;
	struct Property {
		PropertyName	name;
		PropertyValue	value;
	};
	typedef sequence<Property> PropertySeq;

	enum HowManyProps {none, some, all};
	union SpecifiedProps switch (HowManyProps) {
		case some: PropertyNameSeq prop_names;
	};

};		// module CosTrading

#endif
```

15.9 RACommon.idl

```
#ifndef _RACOMMON_IDL_
#define _RACOMMON_IDL_

//
// RACommon.idl
//
// History:
// 97/08/?? Initial contribution from VITAL
// 97/11/?? Modified by Frank Steegmans?
// 97/12/05 Revised by Takeo Hamada
//

#include <States.idl>
#include <naming.idl>
#include <Common.idl>

// ---------------
// VITALCommon.idl
// ---------------
```

```
enum Directionality {Unidirectional, Bidirectional};

enum Direction {Source, Sink, SourceSink};

// ----------------------
// End of VITALCommon.idl
// ----------------------

// -----------------
// NetworkCommon.ild
// -----------------


typedef string CharacteristicInfo;
// VITAL Definition for LNType. This enumerated will be extended as new LNs are incorporated.
// enum LNType {ATM_VC, ATM_VP};

// -----------------
// Added by fste
// -----------------

#include <capability.idl>

///////////////////////////////////////////////////////////////////////////
// Data Structure: t_NFEPDescription
// Members: t_NFEPReference, t_CapabilitySet
// The reference part is used to identify the NFEP concerning in NFEP in
// the context of the different COs. (e.g. crossreferences composed of
// an "identifier=value" pair)
// The capability set is used to identify a complete NFEP pool or just one
// NFEP.
// E.g. NFEPDescriptions are passed down as an inout parameters from the
// CSM to the CC and further (possible as a subset) to the LNBM to
// identify the several end-points of a NFC and respective a LNB.
// In the return of the operation, the NFEPDescription will identify the
// particular capability that has been choosen.
///////////////////////////////////////////////////////////////////////////



typedef string          t_IP_ReferenceType;
const   t_IP_ReferenceType      DNS_IP_ReferenceType="DNS";
const   t_IP_ReferenceType      Plain_IP_ReferenceType="IP";

typedef string          t_IP_ReferenceVersion;
const   t_IP_ReferenceVersion   IP_DNSVersion="DNS"; // :)
const   t_IP_ReferenceVersion   IP_AddressVersion_4="IPv4";
const   t_IP_ReferenceVersion   IP_AddressVersion_6="IPv6";

typedef string t_IP_ActualReference;    // 192.4.160.1 or redbank.tinac.com

struct  t_IP_HostReference {
        t_IP_ReferenceType      type;
        t_IP_ReferenceVersion   version;
        t_IP_ActualReference    reference;
        };
```

```
struct  t_IP_NWInterfaceReference {
        t_IP_ReferenceType      type;
        t_IP_ReferenceVersion   version;
        t_IP_ActualReference    reference;
        };

typedef string                  t_IP_ProtocolType;
const   t_IP_ProtocolType       TCP_IP_ProtocolType="TCP";
const   t_IP_ProtocolType       UDP_IP_ProtocolType="UDP";
const   t_IP_ProtocolType       RTP_TCP_IP_ProtocolType="RTP_TCP";      // ???

typedef unsigned short  t_TCPPortNumber;
typedef unsigned short  t_UDPPortNumber;


// -----------------
// End added by fste
// -----------------



// New VITAL definition:
enum ConnType {PointToPointUni, PointToPointBi, PointToMultipointUni};
// New VITAL definition. Included to simplify the parameters in CP operations.
enum TrafficDirection {Reception, Transmission};

// typedef Direction TPDirection;
// typedef any StreamIFref;             // still to be defined

// Traffic definitions
struct PacketStreamQos {
  unsigned long errorRatio;             // magnitude order (10exp-)
  unsigned long lossRatio;              // magnitude order (10exp-)
  unsigned long misinsertionRate;       // mag order (10exp-)
  unsigned long delay;                  // end-end delay (msec)
  unsigned long jitter;                 // delay variation (msec)
};

struct PacketStreamTrafficDescription {
  unsigned long maxLength;              // bits
  unsigned long averageRate;            // bits/sec
  unsigned long peakRate;               // bits/sec
  unsigned long burstiness;             // no. of packets
};

struct ConnectionDescription {
  ConnType  type;
  PacketStreamTrafficDescription recBw;
  PacketStreamTrafficDescription transBw;
  PacketStreamQos recQos;
  PacketStreamQos transQos;
};

struct NatureOfSyncGroup {
  boolean sameRoute;
  boolean lipSynchronization;
```

```
  boolean sameLife;
  boolean sameDeath;
  boolean differentRoute;
  boolean sameNwTp;
};

struct FaultPolicies {
  boolean sendNotification;          // default true
  boolean selfRecovery;              // default false
};

struct AccountingPolicies {
  string currency;
  unsigned long maxCost;
};

struct RoutingPolicies {
  string useSpecificDomain;
  string dontUseSpecificDomain;
  unsigned long maxNumberOfHops;
  string useSpecificNwTp;
  string dontUseSpecificNwTp;
};

struct ConfigPolicies {
  RoutingPolicies routingPolicies;
  boolean aliveWoLeafEdges;
};

// Changes with v1 because imcompatibility with t_SuccessCriterion
//enum QosCommit {BestEffort, Deterministic, Statistical};
enum QosCommit {QoSBestEffort, QoSDeterministic, QoSStatistical};

struct QosCommitClass {
  QosCommit commitType;
  unsigned short percentage;
};

struct PerformancePolicies {
  QosCommitClass error;
  QosCommitClass loss;
  QosCommitClass misinsertion;
  QosCommitClass avgDelay;
  QosCommitClass varDelay;
};

typedef any SecurityPolicies;          // still to be defined

// New VITAL definitions. Added to allow for the definition of default configurations
// for CM Policies.
// Types of Policies:
enum PolicyType { Fault, Configuration, Accounting, Performance, Security };
typedef sequence<PolicyType> PolicyTypeList;

// VITAL Definition. Modification of previous TINA definition.
struct CMpolicies {
  // New field added. It indicates the policies which configuration is indicated in
```

```
  // the structure. If not included the policy will adopt a default value.
  PolicyTypeList policiesUsed;
  FaultPolicies  fault;
  ConfigPolicies configuration;
  AccountingPolicies accounting;
  PerformancePolicies performance;
  SecurityPolicies security;
};


// End-point in a SNW (Ep):
// VITAL Definition: A new type is added.
// The Tp specified by the user will be UserTp.
// The Tp from the network point of view will be NwTp.

enum EndPointType {NwTp, NwTpPool, Ltp, NwAddress, UserTp};

typedef t_TinaName TpName;
typedef t_TinaName TpPoolName;
typedef t_TinaName LinkTermPointName;
typedef t_TinaName UserTpName;
typedef string t_NetworkAddress;
typedef string ServiceState;

typedef sequence<t_NetworkAddress> t_NFEPpools;

union EpRef switch (EndPointType) {
case NwTp: TpName tp;
case NwTpPool: TpPoolName tpPool;
case Ltp: LinkTermPointName linktp;
case NwAddress: t_NetworkAddress naddr;
case UserTp: UserTpName usertp;
};
typedef sequence <EpRef> EpRefList;

// Edge: used by SNCs, tandem connections and trails
typedef long EdgeId;
typedef sequence <EdgeId> EdgeIdList;

// Maps a logical id "id" with the physical end-point "physicalId".
struct EdgeIdentity {
  EdgeId id;
  t_TinaName physicalId;
};
typedef sequence <EdgeIdentity> EdgeIdentityList;

enum EdgeType {Root, Leaf};

struct EdgeInfo {
  EpRef  ep;
  EdgeType type;
  ServiceState state;
};
typedef sequence <EdgeInfo> EdgeInfoList;

// To BE DEFINED
typedef string BandwidthType;
```

```
typedef string EndPointGroup;
typedef sequence <EndPointGroup> EndPointGroupList;

/* Domains */
typedef string AdministrationDomain;
typedef string CMdomain;

typedef boolean WaitInd;

// --------------------------
// -- END OF NetworkCommon.idl
// --------------------------

// ----------------
// TCSM definitions
// ----------------

typedef string LNType;

typedef sequence<LNType> LayerNetworkList;

enum t_CNSBoolean {True, False};

typedef string t_NProviderId;

enum t_MFStatusType { MFOn_Line, OSFailure, HWFailure, UnspecifiedFailure };

enum t_NFEPStatusType { NFEPOn_Line, NormalTermination, Failure };

enum t_NotifyPolicy { Warning, Fatal };

enum t_NotifyObjectType { NfepNotify, MfNotify};

union t_NotifyData switch(t_NotifyObjectType)
{
case NfepNotify: t_NFEPStatusType newNfepStatus;
case MfNotify:   t_MFStatusType   newMfStatus;
};

struct t_CnsNotifyRecord
{
  t_NotifyPolicy policy;
  t_NotifyData   data;
};



struct t_UserTpDescr
{
  UserTpName usertp;
  ConnectionDescription connDescr;
};
typedef sequence<t_UserTpDescr> t_UserTpList;


// ----------------------
// End of TCSM definitions
```

```
// -----------------------

// ----------------------
// CSM and CC definitions
// ----------------------

typedef string t_CorrelationId;
typedef t_IntRef t_TLARef;


// -----------------------------
// End of CSM and CC definitions
// -----------------------------

#endif
```

15.10Security.idl

```
#ifndef _security_idl_
#define _security_idl_

//
//      File Name: Security.idl
//      Decription: definitions for security components.
//      Revision Histroy:
//          9-17-97 v0.1 by Takeo Hamada
//          9-18-97 v0.2 by Takeo Hamada
//              module structure revised, passed hidl
//              compiler.
//

#ifdef debug
module Security {
   typedef sequence <octet> Opaque;
       // same as Security::Opaque, CORBA security 15-76.
};
#endif

#endif
```

15.11States.idl

```
#ifndef _STATES_IDL_
#define _STATES_IDL_

//   DESCRIPTION:
//      State Management Definitions.
//
//   COMMENTS: Based in TINA Defs file (deviations are commented)

/* 1. Generic Attributes --------------------------------------- */

/* 1.1. State Attributes --------------------------------------- */

module m_STATE {
```

```
enum t_OperationalState {      /* single-valued and read-only */
  Disabled,          /* resource totally inoperable and unable to
                    provide service to the users */
  Enabled            /* resource partially or fully operable and
                    available for use */
};

enum t_UsageState {     /* single-valued and read-write */
  /* Not all values are applicable to every class of managed object */
  Idle,             /* resource not currently in use */
  Active,            /* resource in use,
                    and with sufficient spare operating capacity */
  Busy,              /* resource in use, but no spare operating capacity */
  Reserved           /* resource reserved. This is NOT an ISO state */
};

enum t_AdministrativeState {    /* single-valued and read-write */
  /* Not all values are applicable to every class of managed object */
  Locked,             /* prohibited from performing services for its users */
  ShuttingDown,        /* permitted to existing instances of use only */
  Unlocked            /* permitted to perform services for its users.
                    This is independent of its inherent operability */
};

struct t_ManagementState {
  t_OperationalState operational;
  t_UsageState usage;
  t_AdministrativeState administrative;
};

/* 1.2. Status Attributes (qualify the state attribute) -------- */

enum AlarmStatus {      /* set-valued and read-write */
  UnderRepair,         /* resource currently being repaired */
  Critical,           /* some critical alarms have not yet been cleared */
  Major,              /* some major alarms have not yet been cleared */
  Minor,              /* some minor alarms have not yet been cleared */
  AlarmOutstanding     /* see additional attributes */
};

enum ProceduralStatus { /* set-valued and read-write */
  InitializationRequired, /* the resource requires
                     initialization to be invoked by the manager */
  NotInitialized,      /* the resource initializes ifself autonomously */
  Initializing,        /* initialization procedure not yet completed */
  Reporting,          /* the resource is notifying the results of an
                    operation. Its operational state is Enabled */
  Terminating          /* the resource is in termination phase */
};

enum AvailabilityStatus {      /* set-valued and read-only */
  InTest,             /* the resource is undergoing a test procedure */
  Failed,             /* the resource has an internal fault.
                    Its operational state is Disabled */
  PowerOff,           /* the resource is not powered on.
                    Its operational state is Disabled */
  OffLine,            /* the resource requires to be placed on-line.
```

```
                 Its operational state is Disabled */
 OffDutty,         /* the resource has been made inactive internally.
                 Its operational state is Disabled or Enabled */
 Dependency,       /* the resource cannot operate because some other
                 resource on which it depends is unavailable.
                 Its operational state is Disabled */
 Degraded,         /* but its operational state is Enabled */
 NotInstalled,     /* the resource represented by the managed
                 object is not present or is incomplete.
                 Its operational state is Disabled */
 LogFull           /* log full condition (see Rec. X.735) */
};

enum ControlStatus {    /* set-valued and read-write */
 SubjectToTest,       /* available for users, but tests may be
                 conducted on it */
 PartOfServicesLocked, /* administrative state = Unlocked */
 ReservedFortest,     /* administrative state = Locked */
 Suspended            /* administrative state = Unlocked */
};

enum StandByStatus {    /* set-valued and read-only */
 /* its value is only meaningful when the back-up relationship
    role exists (see Rec. X.732) */
 HotStandBy,          /* Not providing service, but operating in synchronism
                 with the resource that is to be backed-up */
 ColdStandBy,         /* Not providing service. Take-over
                 requires some initialization activity */
 ProvidingService     /* */
};

/* From ETSI/NA4, Network Level View: ServiceState ------------- */
/* ServiceState values are defined as a combination of OperationalState,
   UsageState, AdministrativeState, AvailabilityStatus and ControlStatus */

enum ServiceState {
 Planned,
 InServiceAssignedBusy,
 InServiceAssignedActive,
 InServiceReserved,
 InServiceSpare,
 UnavailableFaultyAssigned,
 UnavailableFaultyReserved,
 UnavailableFaultySpare,
 UnavailableLockedAssigned,
 UnavailableLockedReserved,
 UnavailableLockedSpare,
 UnderTestAssigned,
 UnderTestReserved,
 UnderTestSpare,
 CeasingShuttingDown,
 CeasingShutDown,
 Decommissioned
};

}; // module m_STATE
```

```
#endif
```

15.12UMLogManager.idl

```
#ifndef _UmLogManager_idl_
#define _UmLogManager_idl_

//
//    File Name: UMLogManager.idl
//    Decription: usage metering log manager in NCS
//    Comments:
//        Original VITAL inputs were provided by
//        George Pavlou, UCL, UK.
//    Revision Histroy:
//      7-31-97 v0.1 by Takeo Hamada
//      9-17-97 v0.2 by Takeo Hamada
//        module structure revised, passed hidl
//        compiler.

//
//    NVList definition
//

#ifdef debug
module CORBA {
  // bogus definition just for compiler.  See 4-10 of
  // CORBA v2.0.
  interface NVList {
  };
};
#endif

#include "TINAScsAmcCommon.idl"
#include "Security.idl"

module UMLogManager {

  exception e_X721operation {
    enum t_X721error {
      cannotStart,
      cannotStop,
      cannotSuspend,
      cannotResume
    } error;
    string reason;
  };
  exception e_unsupportedAttributeName {
    CORBA::NVList errorList;
  };
  exception e_unsupportedAttributeValue {
    CORBA::NVList errorList;
  };

  interface i_X721LogOperation {
    boolean start() raises (e_X721operation);
    boolean stop() raises (e_X721operation);
    boolean suspend() raises (e_X721operation);
```

```
    boolean resume() raises (e_X721operation);
    boolean set_log_attributes(
        in CORBA::NVList arg_list
    ) raises (
        e_unsupportedAttributeName,
        e_unsupportedAttributeValue
    );
};

exception e_UMLogOperation {
    enum t_UMLogOperation {
        cannotStore,
        cannotGetUserLogEntries,
        cannotRemoveUserLogEntries,
        cannotGetSessionLogEntries,
        // the following error codes are inherited from
        // original VITAL spec. from George.
        unknownUserId,
        invalidLoggingPeriod,
        logBusy,
        unknownSessionId
    } error;
    string reason;
};

interface i_UMLogOperation {
    boolean storeEvent (
        in TINAScsAmcCommon::t_AccountingEvent event
    ) raises (e_UMLogOperation);

    boolean storeEventList (
        in TINAScsAmcCommon::t_AccountingEventList events
    ) raises (e_UMLogOperation);

    boolean removeEvent (
        in TINAScsAmcCommon::t_AccountingEvent event
    ) raises (e_UMLogOperation);

    boolean removeEventList (
        in TINAScsAmcCommon::t_AccountingEventList events
    ) raises (e_UMLogOperation);

    boolean getUserLogEntries (
        in TINACommonTypes::t_UserId userId,
        in TINAScsAmcCommon::t_DateTime from,
        in TINAScsAmcCommon::t_DateTime to,
        out TINAScsAmcCommon::t_AccountingEventList events
    ) raises (e_UMLogOperation);

    boolean removeUserLogEntries (
        in TINACommonTypes::t_UserId userId,
        in TINAScsAmcCommon::t_DateTime from,
        in TINAScsAmcCommon::t_DateTime to
    ) raises (e_UMLogOperation);

    boolean getSessionLogEntries (
        in TINACommonTypes::t_SessionId sessionId,
```

```
        out TINAScsAmcCommon::t_AccountingEventList events
      ) raises (e_UMLogOperation);
   };
};

#endif
```

15.13accPolicyManager.idl

```
#ifndef _accPolicyManager_idl_
#define _accPolicyManager_idl__

#ifdef debug
#include "PLATyToolsFix.idl"
#endif

//
//      File Name: accPolicyManager.idl
//      Decription: policy manager of
//              accounting management domain
//      Revision Histroy:
//              8-1-97 v0.1 by Takeo Hamada
//              12-10-97 v0.2 by Takeo Hamada
//                      successfully compiled by hidl
//

module AccPolicyManager {

   enum t_AccPolicyManagerCode {
      cannotStart,
      cannotStop,
      cannotSuspend,
      cannotResume,
      cannotUpdatePolicy,
      cannotDeletePolicy,
      cannotPropagatePolicy,
      noAuthorization
   };
   exception e_AccPolicyManager {
      t_AccPolicyManagerCode error;
      string reason;
   };

   // it is assumed that policy rules can be
   // represented by an NVList, name-value pairs.

   interface i_AccountingPolicyManager {
      // control operations
      boolean start() raises (e_AccPolicyManager);
      boolean stop() raises (e_AccPolicyManager);
      boolean suspend() raises (e_AccPolicyManager);
      boolean resume() raises (e_AccPolicyManager);
      boolean update_policy(
         in CosTrading::PropertySeq rules
      ) raises (e_AccPolicyManager);
      boolean delete_policy(
         in CosTrading::PropertySeq rules
```

```
        ) raises (e_AccPolicyManager);
        boolean propagate_policy() raises (e_AccPolicyManager);
    };
};

#endif
```

15.14attribute.idl

```
#ifndef _ATTRIBUTE_IDL_
#define _ATTRIBUTE_IDL_

//
// Created by Jarno Rajahalme
// Updates:
// 97/08/06 by Frank Steegmans
//      Replace 'TinaAttrib' with 'Attrib'
//

#include "naming.idl"

//
// Attribute Data Types
//
// An attribute consists of a pair of an identifier and a
// value. The identifier identifies the semantics of the attribute,
// and in particular, the actual data type of the value.
//
// The "tag" is defined as t_TinaName to follow the TINA naming
// conventions. The value is defined as any to allow carriage of
// IDL structured values without explicit parsing etc. (which would
// be required, if e.g. string would be used).
//

struct t_Attrib {
        t_TinaName  id;
        any         value;
};

//
// Attribute list
//
typedef sequence <t_Attrib> t_AttribList;

#endif // _ATTRIBUTE_IDL_
```

15.15capability.idl

```
#ifndef _CAPABILITY_IDL_
#define _CAPABILITY_IDL_

//
// Created by Jarno Rajahalme
// Updates:
// 97/08/06 by Frank Steegmans
//      Added ';' to solve syntax error
```

```
//


//
// IDL definition for terminal capability information.  This is
// based on ASN.1 code given in ITU-T H.245, with following
// extensions:
//
// 1. Capability set can also have session and transport protocol
// capabilities.
//
// 2. Each codec, protocol, etc. capability is explicitly
// identified with ASN.1 OBJECT IDENTIFIER. This will identify the
// semantics of the capability and the semantics of the associated
// attributes. This is like the non-standard capability in H.245.
//
// 3. A capability can require other capabilities. For example, a
// codec may only be supported over RTP session protocol over UDP
// transport. Another instance of the same codec could be
// available over ATM interface only, etc.
//
// See H.245 section 7.2.1 for overview on terminal capabilities
//
// Problems:
//
// 1. Usage of 'any' (via t_AttribList)
//
// TODO:
//
// 1. Full example of capability information
//
// HISTORY
//
// 07/24/97     Jarno Rajahalme         Initial Contribution
//

#include "naming.idl"          // t_TinaName
#include "attribute.idl"       // t_AttribList

//
// Type for capability unique semantic identifier.  This should
// from a subtree of ASN.1 OBJECT IDENTIFIER name space
//
typedef t_TinaName t_CapabilityId;

//
// Type for terminal unique capability key
//
typedef unsigned long t_CapabilityKey;
typedef sequence<t_CapabilityKey> t_CapabilityKeyList;

//
// Capability dependencies
//
// t_AlternativeDependencies is a sequence of alternative
// t_CapabilityKeyLists.  If the contained list has more than one
// t_CapabilityKey, then all those are required at the same
```

```
// time. This is useful to express dependency of a multi-channel
// ISDN link, for example.
//
typedef sequence<t_CapabilityKeyList> t_AlternativeDependencies;

//
// High-level type of the capability.
//
// This information should be contained in the capability semantic
// identifier (a certain subtree of OBJECT IDENTIFIERS would be
// reserved for codec capabilities, subrees of which would be
// reserved for audio, video, data, etc.)
//
// The problem with this enumeration is that it is most probably
// not exhaustive, and would require updating every now and
// then. By containing this information into the capability
// semantic name, neither this type, nor the corresponding field
// below, would be needed anymore.
//
enum t_CapabilityType {
  audioCapability, videoCapability, dataCapabilty,
  sessionProtocolCapability, transportCapabilty
};

//
// Directionality of the capability.
//
// NOTE: This is not related to the directionality of an SFEP! Two
// SFEPs would be needed for e.g. video receive and transmit, even
// if only one capability with directionality
// 'receiveAndTransmitCapability' is indicated.
//
enum t_CapabilityDir {
  receiveCapability,
  transmitCapability,
  receiveAndTransmitCapabilty
};

//
// Capability
//
struct t_Capability {
  t_CapabilityKey       key;    // Terminal unique id
  t_CapabilityId        name;   // unique semantic identifier
  t_CapabilityType      type;   // High-level Capability Type
  t_CapabilityDir       dir;    // Capability directionality
  t_AttribList          attributes; // Capability attributes
  t_AlternativeDependencies dependencies; // Required capabilities
};
// list of capabilities:
typedef sequence<t_Capability> t_CapabilityList;

//
// Capability Combinations
//

//
```

```
// Terminal unique id for a CapabilityDescriptor.
//
// The number indicates preference of mode of operation by the
// terminal: descriptors with lower numbers are preferred (i.e. 0
// == most preferred)
//
typedef unsigned long t_CapabilityDescriptorNumber;

//
// List of alternative capabilities. Any ONE of these can be done
// at one time - list items are mutually exclusive (within this
// list only)
//
typedef t_CapabilityKeyList t_AlternativeCapabilities;

//
// Capability Descriptor. This contains a terminal unique
// identifier for this set of capabilities and a set of
// capabilities supported simultaneously
//
struct t_CapabilityDescriptor {
  t_CapabilityDescriptorNumber       capabilityDescriptorNumber;
  sequence<t_AlternativeCapabilities> simultaneousCapabilities;
};

//
// Capability Set
//

struct t_CapabilitySet {
  sequence<t_CapabilityDescriptor>     capabilityDescriptors;
  t_CapabilityList                     capabilityTable;
};

#endif // _CAPABILITY_IDL_
```

15.16csm.idl

```
#ifndef _CSM_IDL_
#define _CSM_IDL_

//
//  Still needs to be sorted out!!!!!!!!!
//



//   NAME:  CSM.idl
//   DESCRIPTION:
//      Communication Session Manager Interfaces.
//
//      IDL INTERFACES
//            Supported:
//                  Name: CSM::i_ComSSetup
//                  Name: CSM::i_ComSCtrl
//            Requested:
//                  Name: TCSM::i_TerminalFlowControl
```

```
//
//
//    SPECIFICATION REF:
//          Document: Network Resource Architecture Version 3.0
//
//    SCENARIOS REF:
//          Document: Network Resource Architecture Version 3.0
//
//----------------------END DESCRIPTION HEADER----------------------------

/*
 * REMARK:
 * 97/08/20     Modified by Frank Steegmans
 *      VITAL specification has been used as source
 *      Sfep description has been changed to take mediadescription
 *      into account. Furthermore an operation is added to change
 *      this description for a nubmer of SFEPs.
 *      Minor modifications have been done to make it compilable.
 * 97/12/05     Revised by Takeo Hamada
 */

// #include "Common.idl"
#include "sfepcoms.idl"
#include "nfep.idl"
#include "States.idl"
#include "exceptions.idl"
#include "ComSCommonDefs.idl"


module RET_CSM
{

  interface i_CSMCommonDefs : i_ComSCommonDefs
    {

  typedef string t_ComSessionName;
  typedef sequence<t_ComSessionName> t_ComSessionNameList;


//enum t_SFepDirection {SFepSource, SFepSink}; - Common.idl
//enum t_Topology {PointToPoint, PointToMultipoint, Broadcast}; - Common.idl
//enum AdministrativeState {Locked, ShuttingDown, Unlocked} - States.idl

//struct t_SFep
//{
//    t_IntRef       tcsmRef;
//    t_SFep_CId     SFepId;
//};
//typedef sequence<t_SFEPName> t_SFEPNameList; - Common.idl


  /*
  struct t_SFepDesc
  {
    t_SFEPName              sfepRef;
    m_STATE::t_AdministrativeState          sfepState;
    t_SFepDirection         sfepDir;
```

```
};
typedef sequence<t_SFepDesc>  t_SFepDescList;
*/

typedef      t_SFEPComDesc          t_SFepDesc;
typedef      t_SFEPComDescList      t_SFepDescList;


struct t_SFCDesc
{
  t_SuccessCriterion         sfcSuccCrit;
  t_SFCName                  sfcName;
  t_ConnTopology             sfcTopology;
  t_SFepDescList             SFeps;
};
typedef  sequence<t_SFCDesc> t_SFCDescList;

struct t_SFCResponse
{
  t_SFCName                  sfcName;
  t_SFEPNameList             boundList;
};
typedef  sequence<t_SFCResponse> t_SFCRespList;

  }; // i_CSMCommonDefs

interface i_ComSCtrl : i_CSMCommonDefs {

  //...Flow Connection related Operations

  void setup_flow_connections
    (in  t_SuccessCriterion   criterion,
     in  t_SFCDescList               sfcDescList,
     out t_SFCRespList               sfcSetList)
    raises (e_Error) ;

  void release_flow_connections
    (in  t_SuccessCriterion   criterion,
     in t_SFCNameList                sfcList,
     out t_SFCRespList               sfcRelList)
    raises (e_Error);

  void activate_flow_connections
    (in  t_SuccessCriterion   criterion,
     in  t_SFCNameList        sfcList,
     out t_SFCRespList               sfcActList)
    raises (e_Error);

  void deactivate_flow_connections
    (in  t_SuccessCriterion   criterion,
     in  t_SFCNameList               sfcList,
     out t_SFCRespList               sfcDeactList)
    raises (e_Error);



  //...Branch related Operations
```

```
    void add_branches
      (in  t_SuccessCriterion   criterion,
       in  t_SFCName            sfcName,
       in  t_SFepDescList       sfepDescList,
       out t_SFEPNameList       boundList)
      raises (e_Error);

    void delete_branches
      (in  t_SuccessCriterion   criterion,
       in  t_SFCName            sfcName,
       in  t_SFEPNameList       list,
       out t_SFEPNameList       dellist)
      raises (e_Error);

    void activate_branches
      (in  t_SuccessCriterion   criterion,
       in  t_SFCName            sfcName,
       in  t_SFEPNameList       list,
       out t_SFEPNameList       actList)
      raises (e_Error);

    void deactivate_branches
      (in  t_SuccessCriterion   criterion,
       in  t_SFCName            sfcName,
       in  t_SFEPNameList       list,
       out t_SFEPNameList       deactList)
      raises (e_Error);


    struct t_SFEPMediaDesc {
      t_SFEPName       name;
      t_MediaDesc      media;
    };
    typedef sequence<t_SFEPMediaDesc> t_SFEPMediaDescList;


    void modify_branches_media_desc
      (in  t_SuccessCriterion   criterion,
       in  t_SFCName            sfcName,
       in  t_SFEPMediaDescList  list,
       out t_SFEPNameList       succList)
      raises (e_Error);


    // ....Get operations

    void list_all_SFCs
      (out  t_SFCNameList      sfcList)
      raises (e_Error);

    void get_flow_conn_info
      (in  t_SFCName            sfcName,
       out t_SFCDesc            sfcDesc)
      raises (e_Error);

  }; // i_ComSCtrl
```

```
  interface i_ComSSetup : i_CSMCommonDefs {
    void  setup_communication_session
      (in  t_SuccessCriterion   criterion,
       in  t_SFCDescList              sfcDescList,
       out t_ComSessionName     comSessionName,
       out i_ComSCtrl           comSCtrl,
       out t_SFCRespList              sfcRespList)
      raises (e_Error);

    void release_com_session
      (in t_ComSessionName      comSessionName)
      raises (e_Error);

    void  activate_communication_session
      (in t_ComSessionName             comSessionName)
      raises (e_Error);

    void  deactivate_communication_session
      (in t_ComSessionName             comSessionName)
      raises (e_Error);

    void list_all_communication_sessions
      (in t_ComSessionNameList  comSessionNameList)
      raises (e_Error);

  };

}; // end of module RET_CSM

#endif // i_ComSSetup
```

15.17 exceptions.idl

```
#ifndef _EXCEPTIONS_IDL_
#define _EXCEPTIONS_IDL_

//
// Created by Jarno Rajahalme
// Updates:
// 97/08/06 by Frank Steegmans
//      type identifier 't_error' added to solve syntax problem
//


//
// Generic exceptions to be used by all Tina code
//
// NOTE: I'm not 100% sure whether enum usage below is actually
// allowed...
//

//
// Generic non-trancient error, typically indicates a bug in the
// client code
```

```
//

exception e_Error {
  enum t_error {
    Unknown,          // Reason unknown
    InvalidArgs,      // Invalid arguments for the operation
    InvalidState      // Incorrect operation sequencing
  } error;
  string      message;
};

//
// Generic error on the server side
//

exception e_ServerError {
  enum t_ServerError {
    Unknown,          // Reason unknown
    NoResources,      // Computing resource shortage
    ObjectNotFound,   // Addressed object not found
    ObjectStateInvalid, // Addressed object not in suitable state
    ObjectInUse       // Object name etc. is already in use
  } error;
  string      message;
};

#endif
```

15.18lnc.idl

```
/*
 *    File:       LN.idl
 *
 *    Description:   This file contains all the IDL files related to the
 *                Layer Network
 *
 *    Comments:      UNDER CONSTRUCTION
 *
 *    History:
 *
 *                97/08/11: Initial Contribution
 *                      by Frank Steegmans
 *                97/12/01: Second Contribution
 *                      by Hyun
 *                97/12/10: Revised by Takeo Hamada
 *
 *
 *    TO DO:       - Take out common definitions
 *                - Take out 'allFlag' and replace by success criterium were necessary
 *                - Review exceptions and add lists in were appropriate.
 *
 */


#ifndef i_LayerNetworkCommonDefs_IDL
#define i_LayerNetworkCommonDefs_IDL
```

```
#include "CLNCommonDefs.idl"

// Forward declarations
interface i_LnbNotification;
interface i_LnbControl;
interface i_LnbNotificationControl;

// added by Takeo Hamada - 12/10/97
interface i_LayerNetworkBindingNotification;
interface i_LayerNetworkBindingControl;
interface i_LayerNetworkBindingNotificationControl;
typedef string t_LayerNetworkBindingName;

interface i_LayerNetworkCommonDefs : i_CLNCommonDefs
{

  typedef t_TinaName                          t_LayerNetworkBindingName;
  typedef sequence <t_LayerNetworkBindingName>  t_LayerNetworkBindingNameSeq;

  struct t_RoutingConstraint
  {
    t_LayerNetworkBindingName   refConn;
    t_RoutingOption             routingoption;
  };


  exception InvalidLayerNetworkBindingDesc {
    t_LayerNetworkBindingName   lnbName;
    string                      info;
  };

  exception InvalidLayerNetworkBindingName {
    t_LayerNetworkBindingNameSeq        lnbNameList;
  };


  struct t_LayerNetworkBindingInfo
  {
    t_LayerNetworkBindingName           lnbName;
    t_ConnTopology                      topology;
    m_STATE::t_AdministrativeState      adminState;
    m_STATE::t_OperationalState         opState;
    t_ReliabilityClass                  relClass;
    t_RoutingConstraint                 routeConstraint;
    i_LayerNetworkBindingNotification   notifIf;
    sequence <t_NfepInfo>               nfepInfoList;
  };


  struct t_LayerNetworkBindingDesc
  {
    t_LayerNetworkBindingName   lnbName;
    t_ConnTopology              topology;
    t_ParametersList            parameterList;
    t_RoutingConstraint         routeConstraint;
```

```
    t_NfepDescSeq              nfepDescList;
  };


  typedef sequence <t_LayerNetworkBindingDesc> t_LayerNetworkBindingDescSeq;

  struct t_LayerNetworkBindingResponse
  {
    t_LayerNetworkBindingName                  lnbName;
    i_LayerNetworkBindingControl               ctrlIf;
    i_LayerNetworkBindingNotificationControl   notifCtrlIf;
    t_SuccNfepList                             boundLlist;
    t_FailedNfepList                           unboundList;
  };

  typedef sequence <t_LayerNetworkBindingResponse> t_LayerNetworkBindingResponseSeq;


  struct t_ActivationResponse {
    t_LayerNetworkBindingName   lnbName;
    t_ActivationStatus          lnbStatus;
    t_NfepActivationResponseSeq nfepActivationResponseList;
  };

  typedef sequence <t_ActivationResponse> t_ActivationResponseSeq;


  struct t_LayerNetworkBindingInterfaces {
    t_LayerNetworkBindingName                  lnbName;
    i_LayerNetworkBindingControl               ctrlIf;
    i_LayerNetworkBindingNotificationControl   notifCtrlIf;
  };

  typedef sequence <t_LayerNetworkBindingInterfaces> t_LayerNetworkBindingInterfacesSeq;

};

#endif /* i_LayerNetworkCommonDefs_IDL */


/* IDL of interface i_LayerNetworkBindingNotification */

#ifndef i_LayerNetworkBindingNotification_IDL
#define i_LayerNetworkBindingNotification_IDL

// #include "i_LayerNetworkCommonDefs.idl"


/*
 * This interface is used by a LayerNetworkBinding object
 * as a client for sending notifications to the connectivity layer
 * regarding changes in the operational state of the associated
 * layer network binding.
 */

interface i_LnbNotification :  i_LayerNetworkCommonDefs
{
```

```
  void lnb_StatusChange
    (in t_LayerNetworkBindingName       lnbName,
     in t_nfepStatusInfoSeq             nfepStatusList);

};

#endif /* i_LayerNetworkBindingNotification_IDL */



/* IDL of interface i_LayerNetworkBindingNotificationControl */

#ifndef i_LayerNetworkBindingNotificationControl_IDL
#define i_LayerNetworkBindingNotificationControl_IDL

// #include "i_LayerNetworkBindingNotification.idl"


/*
 * This interface is used by the Connectivity layer to control the emission
 * of notifications by a LayerNetworkBinding object regarding changes in the
 * operational state of the associated binding.
 */

interface i_LnbNotificationCtrl : i_LayerNetworkCommonDefs
{

  void enable_LNB_Notification ()
    raises(NotificationDestinationNotSet);

  void disable_LNB_Notification();

  void set_LNB_Destination
    (in i_LayerNetworkBindingNotification destination);

};

#endif /* i_LayerNetworkBindingNotificationControl_IDL */


/* IDL of interface i_LayerNetworkBindingControl */

#ifndef i_LayerNetworkBindingControl_IDL
#define i_LayerNetworkBindingControl_IDL

// #include "i_LayerNetworkBindingNotificationControl.idl"


/*
 * This interface provides operations for manipulating the associated
 * layer network binding
 */

interface i_LnbControl : i_LayerNetworkCommonDefs
{
```

```
void add_LNB_Branches
 (in    t_SuccessCriterion   criterion,
  in    t_NfepDescSeq        descList,
  out   t_SuccNfepList       boundList,
  out   t_FailedNfepList     unboundList)
 raises(NonExistentFlowEndPoints,
        NetworkFlowEndPointsAlreadyBound,
        InvalidBranchesInfo);


void delete_LNB_Branches
 (in    t_SuccessCriterion   criterion,
  in    boolean              allFlag, // set to true if all branched are to be deleted
  in    t_NfepRefList        list,
  out   t_SuccNfepList       succList,
  out   t_FailedNfepList     failList)
 raises(NonExistentFlowEndPoints);


void activate_LNB_Branches
 (in    t_SuccessCriterion   criterion,
  in    boolean              allFlag, // set to true if all branched are to be activated
  in    t_NfepRefList        list,
  out   t_SuccNfepList       succList,
  out   t_FailedNfepList     failedList)
 raises(NonExistentFlowEndPoints,
        BranchesActiveAlready);


void deactivate_LNB_Branches
 (in    t_SuccessCriterion   criterion,
  in    boolean              allFlag, // set to true if all branched are to be deactivated
  in    t_NfepRefList        list,
  out   t_SuccNfepList       succList,
  out   t_FailedNfepList     failedList)
 raises(NonExistentFlowEndPoints,
        BranchesDeactiveAlready);


void modify_LNB_Branches
 (in    t_SuccessCriterion   criterion,
  in    t_NfepDescSeq        descList,
  out   t_SuccNfepList       succList,
  out   t_FailedNfepList     failedList)
 raises(NonExistentFlowEndPoints,
        InvalidBranchesInfo);


void get_LNB_Info
 (out t_LayerNetworkBindingInfo    lnbInfo);


};

#endif /* i_LayerNetworkBindingControl_IDL */
```

```
#ifndef i_LayerNetworkCoordinator_IDL
#define i_LayerNetworkCoordinator_IDL

// #include "i_LayerNetworkCommonDefs.idl"


/*
 * This interface provides operations for manipulating a connectivity
 * session. One such interface is associated with each connectivity
 * session
 *
 */

/*
 * interface i_LayerNetworkCoordinator :  i_LayerNetworkCommonDefs
 */
interface i_LnbSetup :  i_LayerNetworkCommonDefs
{

  /*
   * This operation is used for creating a layer network binding
   * The LNC is typically associated with large number of LNBs.
   */
  void setup_LNBs
    (in t_ParametersList list,
     in t_SuccessCriterion criterion,
     in t_LayerNetworkBindingDescSeq LayerNetworkBindingDescList,
     out t_LayerNetworkBindingResponseSeq resp)
    raises(InvalidLayerNetworkBindingDesc);

  /*
   * This operation is used for activating a number of bindings
   * Note:
   * The returned boolean indicates the succes of the operation
   * with respect to the succes criterion. The ActivationResponse
   * has to be checked for further details in all cases except
   * when the exception is thrown.
   */

  boolean activate_LNBs
    (in t_SuccessCriterion criterion,
     in t_LayerNetworkBindingNameSeq LayerNetworkBindingList,
     out t_ActivationResponseSeq resp)
    raises(InvalidLayerNetworkBindingName);

  /*
   * This operation is used for deactivating a number of bindings
   */

  boolean deactivate_LNBs
    (in t_SuccessCriterion criterion,
     in t_LayerNetworkBindingNameSeq LayerNetworkBindingList,
     out t_ActivationResponseSeq resp)
    raises(InvalidLayerNetworkBindingName);

  /*
   * This operation is used for releasing a number of bindings
```

```
  */

 void release_LNBs
   (in boolean allFlag, // set to true if all bindings have to be released
    in t_LayerNetworkBindingNameSeq LayerNetworkBindingList)
   raises(InvalidLayerNetworkBindingName);

 /*
  * This operation is used for retrieving the references to the
  * i_LayerNetworkBindingControl and
  * i_LayerNetworkBindingNotificationControl interfaces
  * associated with particular bindings.
  */

 void get_LNB_ctrl_interfaces
   (in boolean allFlag, // set to true if control interfaces for
                        // all bindings have to be retrieved
    in t_LayerNetworkBindingNameSeq LayerNetworkBindingList,
    out t_LayerNetworkBindingInterfacesSeq lnbIfList )
   raises(InvalidLayerNetworkBindingName);

 /*
  * Operation for determining if two given ANfeps
  * satisfy the Can Be Bound relationship
  * Note:
  * This operation might be expanded in further release to support
  * multiple anfeps. The output would be groups of interconnectable
  * afneps.
  */

 boolean canBeBound
   (in m_NFEP::t_ANfep anfep1,
    in m_NFEP::t_ANfep anfep2)
   raises(NonExistentFlowEndPoints);


};

#endif /* i_LayerNetworkCoordinator_IDL */



/* IDL of interface i_LayerNetworkBindingNotificationControl_IDL */

#ifndef i_LayerNetworkBindingNotificationControl_IDL
#define i_LayerNetworkBindingNotificationControl_IDL

// #include "i_LayerNetworkBindingNotification.idl"

/*
 * One such interface is associated with each layer network coordinator.
 * This interface is used by the connectivity level to control the
 * emission of notifications by LayerNetworkBinding objects regarding
 * changes in the operational state of a number of previously specified
 * layer network bindings.
 */
```

```
interface i_LNBNotificationCtrl : i_LayerNetworkDefs
{

  void enable_LNBNotifications
    (in boolean allFlag, // set to true if notifications for all specified
                    // bindings are to be enabled
     in t_LayerNetworkBindingNameSeq LayerNetworkBindingList)
    raises(NotificationDestinationNotSet);

  void disable_LNB_Notifications
    (in boolean allFlag, // set to true if notifications for all specified
                    // bindings are to be disabled
     in t_LayerNetworkBindingNameSeq LayerNetworkBindingList)
    raises(NotAuthenticated, NotAuthorized);

  /*
   * This operation is used for setting the default destination notification
   * interface in the CS Profile object
   *
   */

  void set_LNB_NotificationDestination
    (in t_LayerNetworkBindingNameSeq LayerNetworkBindingList)
    in i_LayerNetworkBindingNotification destination)
  raises(NotAuthenticated, NotAuthorized);

};

#endif /* i_LayerNetworkBindingNotificationControl_IDL */
```

15.19lncfed.idl

```
#ifndef _LNCFED_IDL_
#define _LNCFED_IDL_

// To do:
// ======
// 1. Clean out datatypes (to appropriate headers)
// 2. Change towards Cons-RP approach on success and return of exceptions
// 3. Define appropriate exceptions
// 4. Verify operations on all interfaces
// 5. Verify parameter, exception lists of all operations

/*
 *     Comments:      UNDER CONSTRUCTION
 *
 *     History:
 *
 *                 97/08/11: Initial Contribution
 *                 97/12/01: Second Contribution
 *                       by Hyun
 *                 97/12/10: Revised by Takeo Hamada
 *
 */


#include <RACommon.idl>
```

```
#include <States.idl>

module m_LNCFED
{

  /* Exceptions */
  exception unknownEdge {
    EdgeId e;
  };

  enum WrongEndPointCause {unknownEP, busyEP, unreachable};
  exception wrongEndPoint {
    EpRef ep;
    WrongEndPointCause cause;
  };

  exception toomuchBandwidth {
    unsigned long maxAvailableBw;
  };

  exception qosNotAvailable { };

  exception notSupported { };

  /* local and remote is always from the callers viewpoint. */
  /* parameters for local and remote LTP have to be added such as */
  /* transmit and receive bandwidth as well as transmission delays  */

  // The t_LTP and t_FedLink datastructures are used for exchanging information
  // between the different involved COs concerning the topological link and its
  // terminationpoints used for federation. These structures may be complete or
  // partially filled-in dependent on the context.

  struct t_LTP{
        m_STATE::t_ManagementState      state;
        PacketStreamTrafficDescription  txPSTD;
        PacketStreamTrafficDescription  rxPSTD;
        };

struct t_FedLink {
        t_NetworkAddress        local_add;
        t_LTP                   local_ltp;
        t_NetworkAddress        remote_add;
        t_LTP                   remote_ltp;
        };

typedef sequence<t_FedLink> t_FedLinkList;


interface i_TCControl {                 // used by peer LNC

        void add_tc_branches (inout EpRefList leaves,
                        in t_IntRefList TLArefleaves)
        raises (wrongEndPoint, toomuchBandwidth);

        void delete_tc_branches (in EdgeIdList leaves)
        raises (unknownEdge);
```

```
    /* Activate/Deactivate operations */
    void activate_tc_branches (in EdgeIdList leaves)
    raises (unknownEdge);

    void deactivate_tc_branches (in EdgeIdList leaves)
    raises (unknownEdge);

    /* Activate/Suspend tc (all edges) */
    void activate_tc () ;

    void deactivate_tc ();

  /* Modification of tc bandwidth parameters */
  void modify_tc_traffic_description (
                        in PacketStreamTrafficDescription newBwin,
                        in PacketStreamTrafficDescription newBwout)
   raises (toomuchBandwidth, notSupported);

  /* Get tandem connection information */
  void get_tc_info(
            out t_CorrelationId correlation_id,
            out EpRef a,
            out EdgeId root,
            out t_IntRef TLArefroot,
            out EpRefList z,
            out EdgeIdList leaves,
            out t_IntRefList TLArefleaves,
            out ConnType trailtype,
            out PacketStreamTrafficDescription txbw,
            out PacketStreamTrafficDescription rxbw,
            out string notification_owner);

 }; /* end of m_LNCFED::i_TCControl interface */


interface i_TCSetup {                  // used by peer LNC

  i_TCControl setup_tc(
             in t_CorrelationId correlation_id,
             inout EpRef a,
             out EdgeId root,
             in t_IntRef TLArefroot,
             inout EpRefList z,
             out EdgeIdList leaves,
             in t_IntRefList TLArefleaves,
             in ConnType trailtype,
             in PacketStreamTrafficDescription txbw,
             in PacketStreamQos txqos,
             in PacketStreamTrafficDescription rxbw,
             in PacketStreamQos rxqos,
             in string notification_owner)
   raises (wrongEndPoint, toomuchBandwidth, qosNotAvailable);

  void release_tc (in i_TCControl t);
  /* suspend; detach and delete all edges; delete tc */
```

```
}; /* end of m_LNCFED::i_TCSetup interface */

//
///* This interface will be used for inter-LND configuration management */
////interface i_interLND_CM {
////
////    readonly attribute i_TCSetup federation;
////
////    /* Returns a list of reachable (prefix) domains through this Federation */
////    DomainList      get_reachable_domains(...);
////
////    /* Used by foreign LNC to request federation */
////    /* Returns true if federation is active for all requested toplinks */
////    /* The returned toplinks list should be examinated if returns false */
////    // This operation is used both ways, which means that also already
////    // pending federation requests (links that are not federated yet
////    // because the called party hadn't an LTP yet) will be completed
////    // by calling this operation at the other side.
////
////    boolean  FederationRequest(
////            in      t_SuccessCriterion      sc,
////            in      string                  foreignLnc,
////            in      i_interLND_CM           foreignCm,
////            in      DomainList              foreignDomains,
////            inout   t_FedLinkList           toplinks,
////            in      string                  localLnc,
////            out     DomainList              localDomains,
////            out     i_TCSetup               localFed)
////    raises(FedReqError);
////
////    // Used by foreign LNC to or check the current status concerning federation links */
////    boolean GetFedLinkLTPsState(inout t_FedLinkList toplinks)
////            raises();
////
////    // Used by foreign LNC to announce a status change concerning federation links
////    // In other words, allthough this might change in the future,
////    // this operation will be used to do link management.
////    boolean SetFedLinkLTPsState(
////            in      t_SuccessCriterion      sc,
////            inout t_FedLinkList     toplinks)
////    raises();
////
////    boolean AbortFederation (
////            in      t_SuccessCriterion      sc,
////            in      string                  localLnc,
////            out     t_FedLinkList           toplinks)
////    raises(UnknownLNC, NoFederation);
////
////
//// }; /* end of m_LNCFED::i_interLND_CM interface */
//
// /* This interface will be used by NRCM for federation configuration management */
// interface i_CM {
//
//    enum t_FedPolicy {
//          OnlyAcceptLocalInitializedLinks,        // local federate already happened
//          AcceptNonInitializedLinks               // only local ltps exist
```

```
//          };
//
////   readonly attribute m_LNCFED::i_interLND_CM inter_LND_CM;
//
//     void    set_fed_policy(t_FedPolicy policy);
//
//
//   // The federate operation is used by local configuration management to request a LNC
//   // to federate with a neigbour.
//   // If the local LTPs have already been created the information local t_LTP maybe empty.
//   // Else local t_LTP has to be complete so that it the points can be created.
//   // The operation returns true if the federation was completely successful.
//   // If false is returned while 'BestEffort' is requested, the toplinks should
//   // be checked for individual states.
//   // This operation maybe called multiple times by local NRCM for the same foreign LND.
//     boolean federate (
//          in     t_SuccessCriterion     sc,
//          in     string                 lnc,
//          in     EndPointGroupList      prefixes,
//          inout  t_FedLinkList          toplinks)
//     raises(UnknownLNC, DomainConflict);
//
//     boolean GetFedLinkLTPsState(inout t_FedLinkList toplinks)
//          raises();
//
//     boolean SetFedLinkLTPsState(
//          in     t_SuccessCriterion     sc,
//          inout t_FedLinkList     toplinks)
//     raises();
//
//     boolean RemoveFedLink(
//          in     t_SuccessCriterion     sc,
//          inout t_FedLinkList     toplinks)
//     raises();
//
//     boolean AbortFederation (
//          in     t_SuccessCriterion     sc,
//          in     string                 lnc,
//          out    t_FedLinkList          toplinks)
//     raises(UnknownLNC, NoFederation);
//
//  }; /* end of m_LNCFED::i_CM interface */

}; /* end of module CNP_LNCFED */

#endif
```

15.20media.idl

```
#ifndef _MEDIA_IDL_
#define _MEDIA_IDL_

// FILE:
//   sbcriteria.idl
// DESCRIPTION:
//   Stream Binding Criteria
```

```
//   Operational success criteria and recovery action and success
//   criteria for stream bindings.
// AUTHOR:
//   Stephanie Hogg
// CREATION DATE:
//  18/08/97
// UPDATES:
//

#include "attribute.idl"

// General type identifiers
typedef string t_Identifier;
typedef t_Identifier t_TypeId;

// Stream Binding type identifier
typedef t_TypeId t_SBType;

// General type descriptor
typedef t_Attrib t_TypeAttrib;
typedef t_AttribList t_TypeList;
struct t_TypeDesc
{
	t_TypeId id; // type of type
	t_TypeList desc; // type attributes list
};
typedef sequence<t_TypeDesc> t_TypeDescList;


// Media descriptors
typedef t_TypeDesc t_MediaDesc;
typedef t_TypeDescList t_MediaDescList;


#endif //  _MEDIA_IDL_
```

15.21naming.idl

```
#ifndef _NAMING_IDL_
#define _NAMING_IDL_

//
// IDL type definitions for TINA naming conventions
//

//
// TINA name is a sequence of strings, each having format
// "attribute=value".  Both the attribute names, and value
// semantics have to be documented for each named object when such
// object is defined.
//
// NOTE: Possible typedefs for various object name types must be
// done where the object is specified, NOT HERE!
//

typedef sequence<string>	t_TinaName;
typedef string		t_TinaNameAttribute;
```

```
typedef string          t_TinaNameValue;

#endif // _NAMING_IDL_
```

15.22nfep.idl

```
#ifndef _NFEP_IDL_
#define _NFEP_IDL_

//
// Created by Frank Steegmans (1997)
// History:
// 97/08/08 initial contribution by Frank Steegmans
//


#include "attribute.idl"     // t_AttribList

// The datastructures defined in this file are compliant with the NRA v3.0 and NRIM v2.1
// Minor additions have been done for passing the information in operations.
// Context:
// NFEP = Network Flow End Point (end point of a Network Flow Connection (NFC))
// An ANfep is the abstract superclass from which both the NfepPool and the
// plain NFEP are derived.
// The NFEP represents a specific Termination Point in a specific Layer Network.
// This is either a TP or TPpool. LNW specific details may be passed in the
// attribute list and will only be interpreted by the LNC level.
// However the t_LNType has to be specified and be different from 'NotDefined'.
// The NFEP pool is a collection of NFEPs and NFEP pools. The t_LNType may be
// specified if all contained NFEPs and NFEP pools are of the same t_LNType.
// (This is for processing optimizations.)

module m_NFEP {

typedef t_TinaName t_NfepName;

typedef string  t_LNType;

    // Folowing list of predefined LN types is not exhaustive and may be expanded
    // in the future. This is the main reason for using a string as identifier
    // and not an enum. The reason why a string and not an ordenary number is
    // that it allows a bigger name space to be used.
    // Therefore, following constants should be moved to some
    // implementation specific file.

const   t_LNType     ND_LNType="NotDefined";
    // NFEP pool containing NFEPs refering to different LNWs
const   t_LNType     IP_LNType="IP";
const   t_LNType     ATM_VC_LNType="ATM_VC";
const   t_LNType     ATM_VP_LNType="ATM_VP";


enum t_ANfepType {
  NFEPPool, NFEP
};

enum t_NfepDir {
```

```
  receiveNfep,
  transmitNfep,
  receiveAndTransmitCapabilty
};


// The following structure represent the nfep datastructures.
// It contains the attributes for the two derived classes because it is not
// possible to do any forward declarations in IDL except in a 'struct'.
// Otherwise the structure would have looked like the one defined in the
// comments below the actual structure.
// The attribute list of a nfep will contain more information related to
// the TP the nfep represents and is therefore layer network dependent.
// E.g. required Traffic and QoS parameters for the selected codecs.
// Hence, the symantics have to be specified in layer network dependent
// idls.
// Representing an ANfep as just an attribute list has not been done
// because it also requires that the syntax is specified somewhere else.

struct t_ANfep {
      t_NfepName      name;
      t_LNType        lnType;
      t_ANfepType     nfepType;

      // Following two attributes are only valid if the nfepType is NFEP
      t_NfepDir       dir;
      t_AttribList    attributes;

      // Following attribute is only valid if the nfepType is NFEPPool
#ifndef hidl
      sequence<t_ANfep>       pool;
#endif
};

/*********************************************************************

Following structure would be a better representation of the inheritance
relationship but it is not compliant with the IDL grammar.

struct t_Nfep {
      t_NfepDir       dir;            // Nfep directionality
      t_AttribList    attributes;     // Nfep attributes
};

struct t_ANfep {
      t_NfepName    name;
      t_LNType      lnType;
      union t_NfepSpecialization
            switch (t_NfepSubClass) {
            case Pool: sequence<t_ANfep> pool;
            case Plain: t_Nfep       nfep;
      } nfepSpec;
};

*********************************************************************/

typedef sequence<t_ANfep> t_ANfepList;
```

```
typedef sequence<t_NfepName> t_NfepNameList;

}; // m_NFEP

#endif  // _NFEP_IDL_
```

15.23sfep.idl

```
#ifndef _SFEP_IDL_
#define _SFEP_IDL_

#include "naming.idl"

//
// Stream Flow End Point type definitions
//

//
// At the moment only naming of SFEPs is included here (nothing else
// is required by the TCSM).
//

typedef t_TinaName t_SFEPName;
typedef sequence<t_SFEPName> t_SFEPNameList;

#endif //  _SFEP_IDL_
```

15.24sfepcoms.idl

```
#ifndef _SFEP_COMS_IDL_
#define _SFEP_COMS_IDL_

// FILE:
//   sfepcoms.idl
// DESCRIPTION:
//   SFEP Communication Session Descriptor
//   Gives the SFEP description structure used by the communication
//   session.  This structure is included in the service session
//   descriptor.
// AUTHOR:
//   Stephanie Hogg
// CREATION DATE:
//  18/08/97
// UPDATES:
//      19/08/97 by Frank Steegmans
//      Removed dependency to TINACommonTypes.idl (too much overhead)
//      05/12/97 by Takeo Hamada
//

// #include "TINACommonTypes.idl"
#include "sfep.idl"
#include "States.idl"
#include "media.idl"
```

```
enum t_SFlowDirection { SFlowSink, SFlowSource };
// It might make sense to replace 'SFlow' in this definition
// with 'Sfep'.


// typedef TINACommonTypes::t_Interface t_Interface;
typedef m_STATE::t_AdministrativeState t_AdministrativeState;
typedef /*CORBA::*/Object t_Interface;

struct t_SFEPComDesc {
      t_SFEPName name; // Unique SFEP id for terminal/local domain
      t_SFlowDirection dir; // sink, source, etc.
      t_AdministrativeState adState; // Current state: active/inactive
      t_MediaDesc media; // High level typing, qos parameters
      t_Interface ifRef; // Associated TCSM interface ref (cast to tfcontrol)
      //i_TerminalFlowControl ifRef;   // Associated TCSM interface

};
typedef sequence<t_SFEPComDesc> t_SFEPComDescList;

#endif // _SFEP_COMS_IDL_
```

15.25tcsm.idl

```
#ifndef _TCSM_IDL_
#define _TCSM_IDL_

//
// IDL interface specifications for the TCSM component
//
// Created by Jarno Rajahalme
// Updates:
// 97/08/06 by Frank Steegmans
//      Changes to get the file compiled
// 97/08/08 by Frank Steegmans
//      Complete part with respect to NFEPs
// 97/08/20 by Frank Steegmans
//      Updated to support multiple SFC to multiple NFC mapping
//
//
// REMARK:
// This version is not compiled yet.
// Detailed exceptions have to be worked out.


#include "exceptions.idl"
#include "media.idl"
#include "sfep.idl"
#include "capability.idl"
#include "naming.idl"
#include "nfep.idl"
#include "ComSCommonDefs.idl"
#include "CLNCommonDefs.idl"


module m_RetComS {
```

```
 interface i_RetComSCommonDefs : i_ComSCommonDefs {

   // Data structure used at Ret-RP ComS to exchange the SFEP related info
   struct t_SFEPDesc {
     t_SFEPName        name;
     t_MediaDesc       media;
     t_SFCName         sfc;
   };

   typedef sequence<t_SFEPDesc> t_SFEPDescList;

   // Data structure that describes the capability set of a SFEP with
   // a particular mediaQoS within a terminal.
   // The sfc is used as a crossreference across the communication
   // session for capability matching.
   struct t_SFCCapabilityMapping {
     t_SFCName         sfc;
     t_CapabilitySet   capabilitySet;
   };

   // If more than one SFEP in a terminal is hooked up to the same SFC
   // this following list will of course contain more than one mapping
   // related to one SFC. This will not be a problem since the SFEP
   // instance as such is not important in the matching of capabilities
   // related to one SFC.
   typedef    sequence<t_SFCCapabilityMapping>      t_SFCCapabilityMapList;

   // t_NFCUsage indicates the purpose of the NFC in the sfc to nfc mapping.
   // The terminal that requests the creation of nfc has to fill out the
   // purpose if there is more than one nfc involved in the mapping.
   // The purpose string has to be recognizable by the other terminals
   // so that they can add the appropriate nfeps.
   struct t_NFCUsage {
     t_NetworkFlowConnectionName      nfcName;  // Naming might be aligned.
     string                           purpose;
   };
   typedef    sequence<t_NFCUsage>    t_NFCUsageList;

   struct t_SfcNfcMapping {
     t_SFCNameList     sfcList;
     t_NFCUsageList    nfcList;
   };
   typedef    sequence<t_SfcNfcMapping>      t_SfcNfcMappingList;

   struct t_NFCInfo
   {
     t_NetworkFlowConnectionName      nfcName;
     t_ConnTopology                   topology;
     t_ReliabilityClass               relClass;
//     t_RoutingConstraint            routeConstraint;
   };
   typedef    sequence<t_NFCInfo>     t_NFCInfoList;


#ifdef debug
```

```
  enum  t_NfepUse             {Root, Leaf};
#endif

    // The tfcBranchRef is assigned by the terminal and will be used
    // to associate the finally selected nfep with the tfc branch.
    // As such it replaces the correlationId, which is not adequate
    // in the current situation were multiple sfcs may be mapped to
    // multiple nfcs
    struct t_NFCBranch
    {
      t_NetworkFlowConnectionName       nfcName;
      m_NFEP::t_ANfep                   nfepDescription;
      t_NfepUse                         use;
      t_TinaName                        tfcBranchRef;
    };
    typedef sequence <t_NFCBranch> t_NFCBranchList;

    // t_NetworkRequirements
    // The terminals will add their network requirements to this
    // structure. It is up to the CSM to pass or drop information
    // from these requirements from one terminal to the other.
    // Either way, the CSM has to compile the returned requirements so
    // that it can do the appropriate calls to the connectivity
    // providers.
    // The CSM has to figure the deletion of NFCs out by relating
    // the deletion of branches.
    struct t_NetworkRequirements {
      t_NFCInfoList     existingNfcs;
      t_NFCInfoList     createNewNfcs;
      t_NFCBranchList   addNfcBranches;
      t_NFCBranchList   deleteNfcBranches;
      t_NFCBranchList   modifyNfcBranches;
    };

    typedef sequence <t_SFEPNameList> t_SuccSfepList;

    enum        t_SfepFailureCode {SfepUnknown,
                                   SfepLocked,
                                   SfepActive,
                                   SfepInactive,
                                   QoSCanNotBeSupported};
    // These failure codes should be reviewed.

    struct t_FailedSfep
    {
      t_SFEPName              sfep;
      t_SfepFailureCode code;
    };
    typedef sequence <t_FailedSfep> t_FailedSfepList;

  }; // interface i_RetComSCommonDefs

  interface i_TerminalComSControl : i_RetComSCommonDefs
    {

      // The selectCapabilities operation is used to add additional
      // SFEPs to the session or to modify the mediumQoS.
```

```
    void selectCapabilities
      (in     t_SuccessCriterion      criterion,
       in     t_SFEPDescList          sfeps,
       inout  t_CapabilityList        capabilitySetList,
       inout  t_SfcNfcMappingList     sfcNfcMappinglist,
       inout  t_NetworkRequirements   networkRequirements)
      raises(e_Error, e_ServerError);

    // The releaseSFEPs operation disconnects the SFEPs from the
    // TFC and will indicate the changes to the network part.
    // The TFC will be deleted when the last associated SFEP is
    // released.
    void releaseSFEPs
      (in     t_SuccessCriterion      criterion,
       in     t_SFEPNameList          sfeps,
       inout  t_NetworkRequirements   networkRequirements,
       out    t_SuccSfepList          succList,
       out    t_FailedSfepList        failList)
      raises(e_Error, e_ServerError);

    void ActivateSFEPs
      (in     t_SuccessCriterion      criterion,
       in     t_SFEPNameList          sfeps,
       out    t_SuccSfepList          succList,
       out    t_FailedSfepList        failList)
      raises(e_Error, e_ServerError);

    void DeactivateSFEPs
      (in     t_SuccessCriterion      criterion,
       in     t_SFEPNameList          sfeps,
       out    t_SuccSfepList          succList,
       out    t_FailedSfepList        failList)
      raises(e_Error, e_ServerError);

    void ActivateTerminalCommunicationSession
      (in     t_SuccessCriterion      criterion,
       out    t_SuccSfepList          succList,
       out    t_FailedSfepList        failList)
      raises(e_Error, e_ServerError);

    void DeactivateTerminalCommunicationSession
      (in     t_SuccessCriterion      criterion,
       out    t_SuccSfepList          succList,
       out    t_FailedSfepList        failList)
      raises(e_Error, e_ServerError);

  }; // interface i_TerminalComSControl


interface     i_TerminalComSSetup : i_RetComSCommonDefs
  {

    // The queryCapabilities operation returnes a list of capability
    // sets related to a certain SFEP with a particular mediaQoS.
    // However this relationship is expressed in terms of the SFC.
    void queryCapabilities (in      t_SFEPDescList      sfeps,
                            out     t_CapabilityList    capabilities)
```

```
      raises (e_Error, e_ServerError);

    // The setupTerminalCommunicationSession creates a terminal
    // communication session and creates the related TFC based on
    // the sfep description list and the combined list of capabilities
    // comming from all terminals.
    // It modifies the combined list of capabilities based on the
    // local mapping. This means that capabilities related to a SFC not
    // terminated in this terminal will not be changed. The capabilities
    // related to a terminated SFC will be reduced to reflect the
    // local selection. (e.g. all capability sets related to one SFC
    // are replaced by one set per mediaQoS). This modified used list
    // will be used for setting up the TFCs (session) in other terminals.
    // The CSM will create the first input list by combining the results
    // of the query operation.
    // Furthermore, the operation modifies the network
    // requirements list and returns a reference to the control interface
    // of the session.
    void setupTerminalCommunicationSession
      (in     t_SuccessCriterion      criterion,
       in     t_SFEPDescList          sfeps,
       inout  t_CapabilityList        capabilitySetList,
       inout  t_SfcNfcMappingList     sfcNfcMappinglist,
       inout  t_NetworkRequirements   networkRequirements,
       out    i_TerminalComSControl   tcsmControlIf)
      raises(e_Error, e_ServerError);


  }; // interface i_TerminalComSSetup


}; // module m_RetComS


//
// Following module can be removed for the Ret specs
//

module m_TcsmTermInt { // Terminal internal interfaces of the tcsm

 interface i_TlaTcsm {

   // This operation is used to announce the resolved  NFEP to the tcsm.
   // This will complete the (part of the) network part of the TFC.
   // It still needs to be activated with activateTFC().
   void associateNFEP (in t_TinaName   tfcBranchRef,
                   in m_NFEP::t_ANfep      nfep)
     raises(e_Error, e_ServerError);

     // Following operation is used to release an associated NFEP from
     // the TFC. Hence, the related TFC branch will become not operational.
     void releaseNFEP (in t_TinaName   tfcBranchRef)
       raises(e_Error, e_ServerError);

 }; // i_TlaTcsm

}; // module m_TcsmTermInt
```

```
#endif // _TCSM_IDL_
```

15.26tla.idl

```
// The file Tla.idl contains the idl specification of the TLA.
// The module Ifs_TLA contain the specifications of the interfaces that
// constitute the CNS_TLA CO

#ifndef _TLA_IDL
#define _TLA_IDL

// Updates:
// 97/11/06 by Hyun C. Kim

#include <RACommon.idl>
#include <naming.idl>
#include <States.idl>

module m_TLA
{

  enum CorrelationErrorType { corrIdIsAlreadyUsed,
                        corrIdIsNoValid,
                        NFEPisNotCorrelated
  };

  enum TTPErrorType {InsufficientResources,
                InsufficientBandwidth,
                QoSCannotBeMet,
                NonExistentEndPoint,
                EndPointAlreadyInUse,
                EndPointAlreadyInActive,
                EndPointNotActive,
                ResourceError };

  enum tlaErrorType { SWError,
                HWError,
                ApplicationError,
                MemoryFailure,
                SocketFailure,
                UAPisNotResponding,
                TCSMisNotResponding,
                TLAisNotResponding,
                NullReference,
                DeviceNotSupported,
                ResourceNotAvailable };

  enum IllegalOperationType {OperationalStateError,
                      UsageStateError,
                      ResolutionStateError };

  exception IllegalOperation {
    IllegalOperationType error;
  };
```

```
 exception nfepError {
   tlaErrorType error;
 };

 exception CorrelationError
   {
     CorrelationErrorType errorType;
   };

 exception TTPError
   {
     TTPErrorType errorType;
   };


 /*
  * Data structures
  */

 struct t_CorrelationRelation
 {
   t_CorrelationId   correlation;
   long nfepIndex;
 };

 typedef sequence<t_CorrelationRelation> t_CorrelationTable;
 typedef long t_CnsElementId;



////////////////////////////////////////////////////////////////
//
// Interface I_tcontlaNfepControl
//
// required interfaces:
// supported operations: NfepModifyTtpDescr
//                       NfepModifyTtpName
//                       NfepModifyTtpQos
//
// provided to: LNC
//
////////////////////////////////////////////////////////////////


interface I_tcontlaNfepControl
{

    enum t_NegotiationErrors { QoS, EpRef, Both };

    typedef string t_QoSErrorType;

    typedef string t_EpRefErrorType;

    struct t_QosEprefType {
        t_QoSErrorType QoSError;
        t_EpRefErrorType EpRefError;
    };
```

```
    union t_NfepNegotiationErrorType switch(t_NegotiationErrors)
    {
        case QoS:  t_QoSErrorType QoSError;
        case EpRef: t_EpRefErrorType EpRefError;
        case Both: t_QosEprefType QoSEpRefError;
    };


    exception t_NfepNegotiationError
    {
        t_NfepNegotiationErrorType errorType;
    };


//  void NfepModifyTtpName (in t_CorrelationId correlId,
//                  inout UserTpName tpName)
//  raises(nfepError, CorrelationError, TTPError,
//      t_NfepNegotiationError,
//      IllegalOperation);

    void activate_nfep (in t_CorrelationId corrid)
    raises(nfepError, CorrelationError);

    void deactivate_nfep (in t_CnsElementId nfepId)
    raises(nfepError, CorrelationError);

    void modify_nfep_qos (in t_CorrelationId corrId,
            in ConnectionDescription nfepQos)
    raises(nfepError, CorrelationError, IllegalOperation);

};



 ////////////////////////////////////////////////////////////////
 //
 // Interface I_tlaNfepSetup
 //
 // required interfaces:
 // supported operations: SetupNfep
 //                 ReleaseNfep
 //                 NfepEnableOperation
 //                 NfepDisableOperations
 //
 // provided to: TCSM
 //
 ////////////////////////////////////////////////////////////////


interface I_tlaNfepSetup
{

    typedef string t_MediaType;

    void setup_nfep (
            in t_CorrelationId corrId,
```

```
                    in t_MediaType mt,
                    in Direction ndir,
                    in ConnectionDescription nfepQos,
                    out t_NetworkAddress netAdrs)
    raises(nfepError, CorrelationError);


    void release_nfep (in t_CorrelationId corrId)
    raises(nfepError, CorrelationError);

};


interface I_tcontlaConfQuery
{

    void get_nfep_pools (out t_NFEPpools nfeps)
    raises(nfepError);

};


interface I_tcontlaNotification
  {

   oneway void NfepStatusChange (in t_CorrelationId correllId,
                              in t_NFEPStatusType newStatus);

  };

};

//////////////////////////////////////////////////////////////////////
//
// INTERFACE:   CNS_TLA
//
// BEHAVIOR:   This interface models the "TLA" Computational Object.
//       It inherits all the interfaces declared in the "TLA" module
//
// OPERATIONS:
//      All included in the "Ifs_TLA::I_tcontlaNfepControl" interface.
//      All included in the "Ifs_TLA::I_tlaNfepSetup" interface.
//      All included in the "Ifs_TLA::I_tcontlaConfQuery" interface.
//      All included in the "Ifs_TLA::I_tcontlaNotification" interface.
//
//////////////////////////////////////////////////////////////////////

//interface CNS_TLA :   Ifs_TLA::I_tcontlaNfepControl,
//  Ifs_TLA::I_tlaNfepSetup,
//  Ifs_TLA::I_tcontlaConfQuery,
//  Ifs_TLA::I_tcontlaNotification
//{};

#endif
```