Telecommunications
Information
Networking
Architecture
Consortium

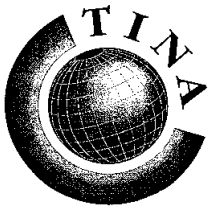
TINA-C Deliverable

Issue Status: Publicly Released

Service Component Specification

Version: 1.0 b

Date of Issue: January 19, 1998



Telecommunications
Information
Networking
Architecture
Consortium

TINA-C Deliverable

Issue Status: Final

Service Component Specification

Computational Model and Dynamics

Version: 1.0b Final

Date of Issue: January 19, 1998

Abstract: This document defines the computational model and dynamics for the TINA Service Component Specifications. The document should be seen as partly a proposal for structuring the internals of specific business administrative domains, and as such, complimentary to the inter-domain reference points, and partly as an aid to understand the interactions occuring across inter domain reference points. All component are described in plain language and specified in ODL and IDL. Several scenarios are given in order to show the dynamic behaviour of the components.

Note: This version has been coreteam and externally reviewed, and it is issued as baseline document.

Main Authors:	**C. Abarca, P. Farley, J. C. Garcia, T. Hamada, P. F. Hansen, P. Hellemans (Alcatel), C. A. Licciardi, K. Nakashiro, M. Yates**
Editor:	**Per Fly Hansen, Carlo Alberto Licciardi**
Stream:	**Service Stream**
Workplan Task:	**Service Architecture**
File Location:	**/u/tinac/97/services/docs/scs/compmod/final/comp.ps**

Table of Contents

1. Introduction

1.1 Purpose

This document aims to

- Assist understanding of the components specified in the TINA Service Architecture [1] by proposing specifications that show what interfaces those components support and require from other components, and the operations present on each of those interfaces.
- Show how the overall component model integrates the Ret Reference Point Specifications by associating Ret interfaces to service components.
- Assist understanding of protocols and event driven interactions amongst the service components and their relationship with dynamic behaviour described in the Ret Reference Point Specifications.

In other words, while the Service Architecture only identifies concepts and the service components (multiple interface objects) this document aims to give low level engineering detail useful to designers and implementors. To achieve this, the document must show how interfaces are grouped together to make a component, what operations are present on each interface and an explanation of the parameters and behaviors of those operations.

1.2 Audience

The document is intended for

- Readers with good knowledge of the TINA Service and Network Architectures and DPE, their rationale, concepts and objectives.
- Readers with good knowledge of TINA Reference Points and in particular the Ret Reference Point specification.
- Designers and implementors of TINA services and TINA service management infrastructures. Such readers are expected from organizations intending to supply service applications, service content, components of services, operate services, operate service content provision, inter-operate with services or other service operators.

1.3 Relationship to other Documents

There are three important related documents:

- The Service Architecture [1] which gives the concepts, components, objectives and rationale for the service architecture.
- The Service Components Specification Part 1 - Information Model which gives structured definitions and relationships amongst the information objects in the Service Architecture, using OMT, quasi-GDMO and GRM.
- Ret Reference Point (Ret RP) which specifies a small proportion of the system interfaces and event sequences that are found in the collection of service components in the system.. The Ret RP is a point of conformance in the service architecture whereas this document describes a possible specification of Service Archtecture.. Since they both define IDL and events their relationship is described in greater detail below.

The relationship between the computational specifications (this document) and the Ret RP Specification is shown in Figure 1-1. It should be understood that the current set of TINA Reference Points are positions of conformity in the TINA architecture. This document is not a prescription for

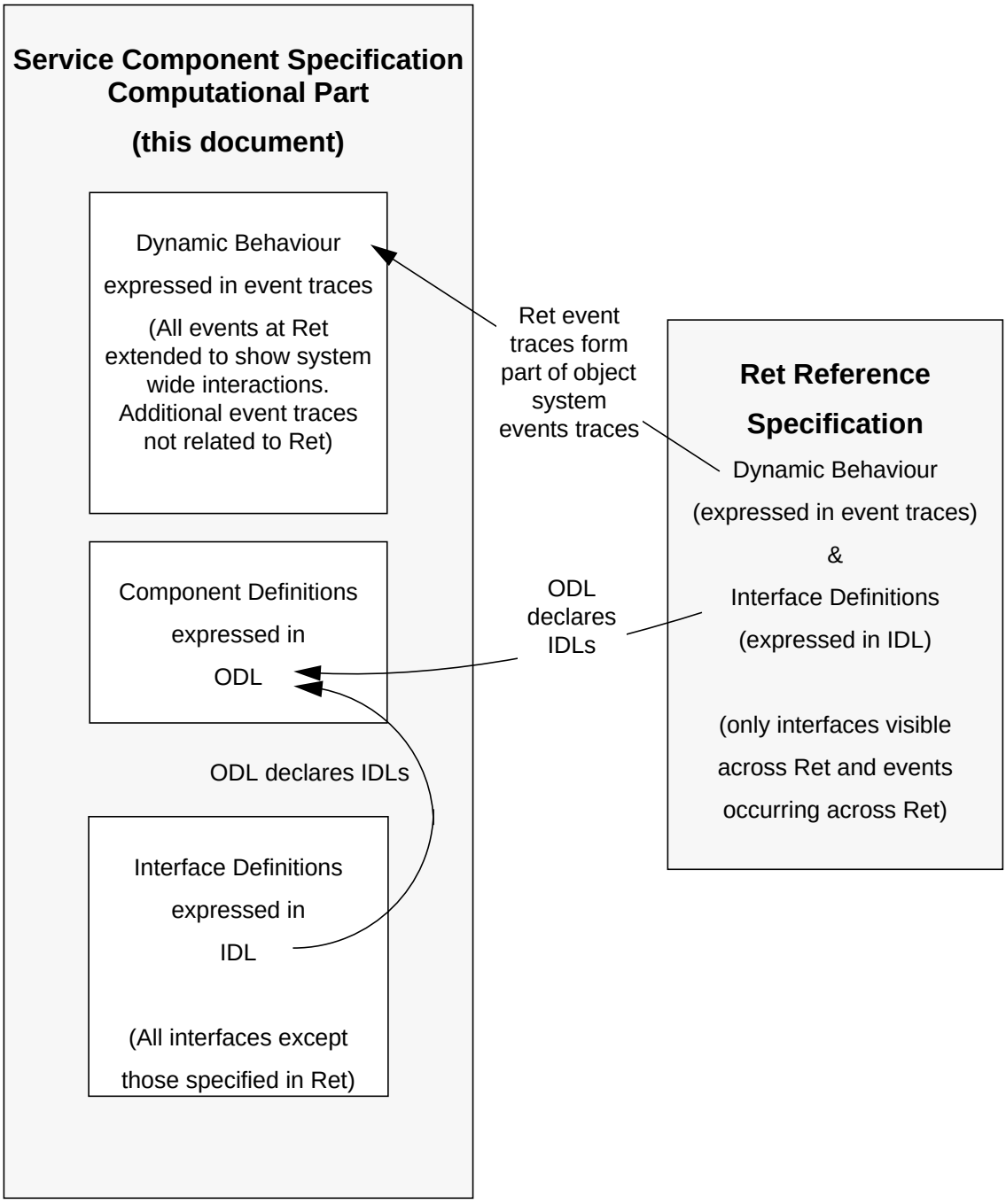


Figure 1-1. Illustration of the relationship between this document and the Ret Specification

conformance. It is a description of an overall service architecture expressed using ODL, IDL and event traces. The component specifications that are here described are entirely conformant to TINA Reference Points of which only Ret concerns this document. Note that

- Events occurring across Ret are extended to show their system wide propagation amongst objects;
- Additional event traces unrelated to Ret are shown here;

- Interfaces are defined here in IDL only if they are not visible across the Ret RP;
- ODL is used here to describe multiple interface objects and so ODL declares interfaces which come from scoped IDL definitions. These IDL definitions are found either in the Ret RP or in this document - but never both places. Some objects therefore declare interfaces exclusively defined here, while the Ret RP related objects have interfaces declared both here and in Ret RP.
- Without the use of ODL (or its equivalent) it is not possible to show the aggregation of interfaces onto multiple interface objects (i.e. service components)
- This document is subject to all changes at Ret

1.4 How to Read This Document

This document is a first version of computational model specifications and dynamics for service components. The readers are strongly recommended to read the Service Architecture [1] that gives the concepts and principles for the computational model, the TINA Business Model [2] giving the overall divisions of the various parts comprising the TINA Architecture, and the Ret specification [12]. In order to understand the contents and the notations of the specifications, the readers are referred to the “Object-Oriented Modeling and Design” [15], the “Computational Modelling Concepts” [4], and the ODL manual [5].

Section 2 gives an overview of all the components expected to be specified in the final version of this document, it gives a short explanation of the purpose of each component and lists the main interactions between components.

Section 3 provides detailed descriptions of a subset of the components. It defines interface and operation names and links to available event traces and IDL.

Section 4 shows a number of event traces, giving some of the dynamic behavior of the components.

ODL for the defined components is given in Annex 1.

Annex 2. gives IDL for the interfaces not defined in the Ret-RP document.

Annex 3. gives background material for defining the subscription management related parts.

Finally, Annex 4. proposes a decomposition of the Subscription management component into several components.

1.5 Current Status

The contents of this document should not be considered final in any sense, it reflects an attempt to stabilize the overall picture of how the service level of TINA based systems can be decomposed into components. It is compliant with the current Ret specifications and reflects the TINA Service Stream’s current view of the responsibilities assigned to individual components within business administrative domains. This decomposition is, however, subject to change and comments and suggestions for improvements are welcome. The ODL/IDL has been compiled, but not validated yet.

1.6 Expected Evolution

The current release of SCS is still incomplete. The following list contains issues that will be addressed/covered in a future release:

- specification of an interface for user-customisation on the UA,
- complete Service Life Cycle Management (SLCM) specification (IDLs);

- introduce specifications, detailed descriptions and scenarios for service composition and federation; to provide complete specification of PeerA, PeerUSM and CompUSM. This part depends on the first results in the process of defining the Retailer-to-Retailer and Third-Party reference points.
- specification of on-line accounting management feature set or specific interface to face the approach of accounting management as an ancillary service (opposite/complementary to the integrated facility approach),

The decomposition into components and the ODL is expected to be validated for the next release, both internally in the core-team and externally in Auxiliary Projects and/or TINA Work Groups.

2. Overall Description of SA Components

This chapter gives the overall view of the components stating the purpose of each component and a table showing which components interact.

The following figure shows all the components specified and shows where interactions occur:

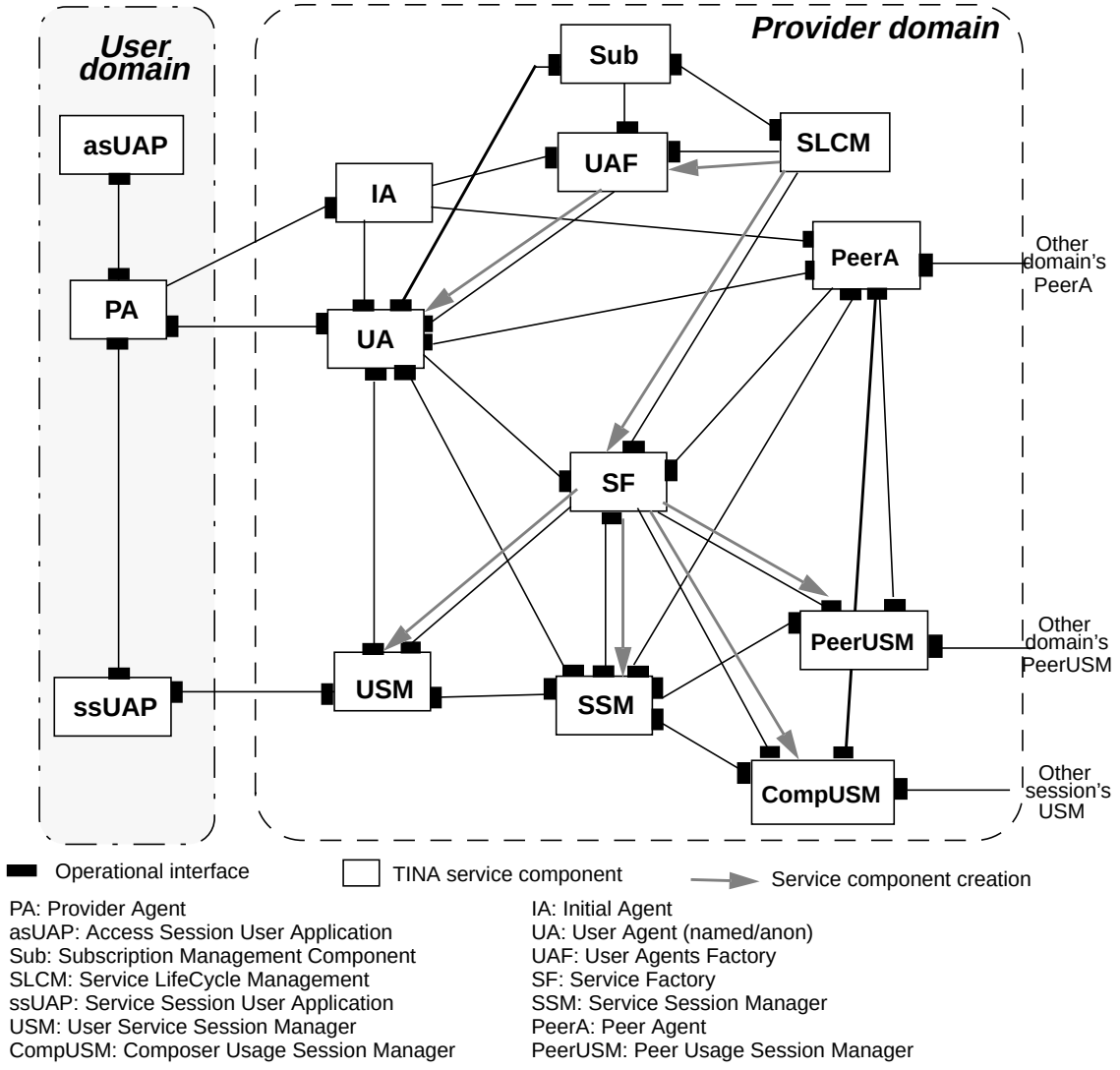


Figure 2-1. Service Components and Interactions

Each of the components are described briefly below.

2.1 asUAP - Access Session User Application

The User APplication (UAP) Service Component (SC) is defined to model a variety of applications in the User domain. A UAP SC represents one or more of these applications. A UAP can be used by human users, and/or other applications in the user domain and can be either or both an access session related and service session related SC. The access session related UAP is defined below. The service session related UAP is defined in Section 2.8.

As an access session related SC, the UAP enables a human user, or another application, to make use of the capabilities of a PA or PeerA, through an appropriate (user) interface. An access session related UAP supports part of the domain access session. A UAP instance may support only access session related capabilities or only service session related capabilities; or it may support both. Access session related UAPs may be specialized by a domain to interact with a specialized PA or PeerA.

2.2 PA - Provider Agent

The Provider Agent (PA) is a service independent SC, defined as the user's end-point of an access session. The PA is supported in a domain, acting in an access user role. The PA supports a user accessing their UA and making use of services, through an access session. The PA supports the user domain access session, in conjunction with access session related UAPs, and other user domain infrastructure.

For each concurrent access session a user has with a provider, there is one PA instance in the user's domain. Each PA may be associated (through an access session) with the same UA, or separate UA instances. A single PA is only ever associated with one UA through an access session. (When no access sessions exist, a user domain can still support a PA. It can be used to initiate an access session, and may receive invitations if registered.)

2.3 IA - Initial Agent

An Initial Agent (IA) is a user and service independent SC that is the initial access point to a domain. An IA is supported by domains taking both the provider and peer roles. An IA reference is returned to a PA or PeerA when it wishes to contact the domain. The IA supports capabilities to authenticate the requesting domain and establishing access sessions.

2.4 namedUA - Named User Agent

A Named User Agent (namedUA) is a service independent SC that represents a user in the provider's domain. The namedUA is a specialization of the UA for a user that is an end-user or subscriber of the provider. It is the provider domain's end-point of an access session with a user. It is accessible from the user's domain, regardless of the domain's location.

The namedUA acts as a single contact point to control and manage the life cycle of service sessions and user service sessions. It negotiates the session models and feature sets supported by a service session. It also manages the user's preferences on service access and service execution, and allows the user to use services from different types of terminals including support of user registration at different terminals to receive invitations.

Operations supported by a namedUA are service independent.

A namedUA may support one or more access sessions concurrently. Each access session is with a single, distinct PA.

2.5 anonUA - Anonymous User Agent

An Anonymous User Agent (anonUA) is a service independent SC that represents a user in the provider's domain. The anonUA is a specialization of UA for users that do not wish to disclose their identity to the provider. It is the provider domain's end-point of an access session with an anonymous user.

The anonUA acts as a single contact point to control and manage the life cycle of service sessions for the anonymous user. This management is possible during the anonymous user's current access session only. The anonUA provides no support for personal or session mobility.

2.6 UAF - User Agent Factory

A User Agent Factory (UAF) is a service independent SC that creates and initializes namedUA and anonUA upon request of Sub (Subscription Management Component) and IA. The UAF is created and managed by SLCM component.

2.7 Sub - Subscription Management Component

The Subscription Management Component (Sub) is a service independent SC, considered as the provider domain's control point of subscriber, user and subscription lifecycle. The Sub is unique in every provider domain. It allows the management of subscribers, subscriptions and users[1] for the whole set of services provided by the provider.

It is accessed by the access session components to retrieve the list of services the user/peer is associated to and the corresponding service profiles. Some specific management components (off-line applications or on-line service session components) also access this component to retrieve and modify the subscription data.

The Sub creates and initializes the access components (Named UAs and PeerAs) and updates them with changes in the corresponding user's subscription data.

It is aware of deployed and active service instances through its interaction with the SLCM.

2.8 ssUAP - Service Session User Application

The User APplication (UAP) represents a variety of applications in the User domain that interact with TINA services and support the service session. It may interact with other applications or humans by supporting appropriate, (undefined in TINA) programmatic or human-computer interfaces.

The ssUAP participates in a service session by supporting the User Domain Usage Service Session (UD_USS). The UAP is service specific and logically supports a combination of session controls; starting/ending the session, inviting other participants to join, joining an existing service session, adding/modifying stream bindings, modification of control session relationships, suspending participation or the whole session, and resuming the participation the whole session. The ssUAP may support and use the TINA Ret session model and feature sets.

The ssUAP may have zero or more stream interfaces attached which may be bound to those in other participants' or providers' domains. One ssUAP can be involved in one or more service sessions. For each service session the UAP is involved in, it interacts with a user session manager USM, or directly with a service session manager, SSM[2].

The ssUAP interacts with the PA to start, resume or join a service session and is initialized by the PA prior to initiating involvement with a service session. When actively involved in a service session it interacts with the USM to perform all service specific and general session control.

2.9 SF - Service Factory

A Service Factory (SF) is a service-specific object, which creates and manages the service session COs (i.e. USM, SSM, CompUSM and PeerUSM) for a service instance[3].

1. Relationship between subscriber, subscription and user are defined in Section 3.6.1. For definition of these concepts see Annex A-3.
2. It depends on the service. For example, in case of one party services the ssUAP interacts directly with the SSM (one example is on line subscription service).
3. The term *service instance* is used to refer to a particular provider's implementation of a service type. A *service type* is an abstraction of a service, that could be provided by more than one implementation (instance).

A request to create a service session of a particular service type results in the creation of one or more object instances. The SF creates and initializes the instances according to rules imposed by their implementation. The SF supports capabilities to manage the created objects (delete, suspend and resume them). The SF returns to the client one or more interface references to these components.

The SF assembles the resources necessary for the existence of a component it creates. Therefore, the SF represents a scope of resource allocation, which is the set of resources available to the SF.

2.10 SSM - Service Session Manager

The Service Session Manager (SSM) is a service component which comprises the service-specific and generic session control segments of the Provider Service Session (PSS). An SSM supports service capabilities that are shared among members (parties, resources, etc.) in a service session. Information related to a particular member of the service session are encapsulated in Member Usage Service Session Managers (MUSMs). An SSM is created by an SF, one per request for the corresponding service type. The deletion of an SSM is service specific, for a number of services it is deleted when all users have left the session, but for others (like chat services) it should be explicitly deleted. The life-span of an SSM is the same as the corresponding provider service session.

2.11 USM - User Service Session Manager

The USM comprises the service-specific and generic session control of the Provider Domain User Service Session (PD_USS). It is a specialization of the MUSM which represents and holds the context of a party, or resource in a service session. It has the same characteristics as the MUSM (with member replaced by party or resource as applicable). A USM is created by the SF, one per request for the corresponding service type (per PD_USS). It is deleted when the party leaves the service session. The life-span of a USM is the same as the corresponding PD_USS.

Within the service session the USM interacts with participants ssUAP or CompUSM. Interactions with the ssUAP are the subject of Ret-RP specification and defined there. The USM also interacts with the service core-logic held in the SSM to convey, where necessary, changes to the participant's involvement in the session and relevant service specific actions.

The USM supports interactions between the access session and service session through interactions with the UA. This is to support access session originated commands on the service session, for example to end or suspend. Additionally since the USM is responsible for one participant's share of accumulated session related charges it interacts with the participant's UA to advise and respond to accounting information.

2.12 PeerA - Peer Agent

The Peer Agent (PeerA) is a service independent SC that represents a peer in another peer's domain. It is supported by a domain acting in the access peer role. It is this domain's end-point of an access session with the peer domain. It supports a peer domain access session. It also represents another domain, or a member of another domain, to this domain, and holds the agreed contract between the domains. It is accessible from the peer's domain, regardless of that domain's location.

The PeerA should support the combined external capabilities of a UA and a PA. It may provide other functionality to ensure the initiator of the access session and responder both receive appropriate references to each other. Also, it ensures the coordination of these external capabilities and maintains consistency of requests and responses.

2.13 CompUSM - Composer Usage Session Manager

The Composer Usage Session Manager (CompUSM) is a service component, which supports composition of service sessions. The composition type supported is asymmetric with one domain taking the usage party role, and the other domain taking the usage provider role. CompUSM is a specialization of the MUSM and supports the Composer Domain Usage Service Session (CompD_USS) (see [1]).

The CompUSM allows a service or resource to act as a usage party in a service session in another domain. It supports usage party interfaces to the service session in the other domain (i.e., to the other service session, the CompUSM appears as if it were a UAP). The other session provides a USM (with usage provider interfaces) to interact with.

2.14 PeerUSM - Peer Usage Session Manager

The Peer Usage Session Manager (PeerUSM) is a service component which supports peer to peer relationships between service sessions in different domains. Federation is symmetric with both domains taking usage peer roles. It is a specialization of the MUSM and supports the Peer Domain Usage Service Session (PeerD_USS).

The PeerUSM allows a service session to interact with another service session in another domain. Both service sessions are peers, and both support a PeerUSM to interact through.

2.15 SLCM - Service Life Cycle Management

The Service Life Cycle Management component (SLCM) is a service-independent SC, which manages the life cycle of TINA services. It provides functionality for (1) deployment, configuration and withdrawal of a service network, (2) management of service types and (3) deployment, configuration, activation, deactivation and withdrawal of service instances. It has a strong interaction with underlaying DPE repositories and services and other facilities, like node management facilities, that allow the management of the service nodes composing the service network.

It interacts with the Sub component to update its information about available (subscribable and usable) services.

2.16 MUSM - Member Usage Service Session

The Member Usage Service Session Manager (MUSM) is an abstract service component, which comprises the service-specific and generic session control segments of the Domain Usage Service Session (D_USS) that interact with the PSS. The generic segments of the MUSM correspond to the TINA session model feature sets. It is specialized according to the role of session member supported by the D_USS:

- User Service Session Manager (USM) represents the UD_USS;
- Composer Usage Session Manager (CompUSM) represents the CompD_USS;

- Peer Usage Session Manager (PeerUSM) represents the PeerD_USS.

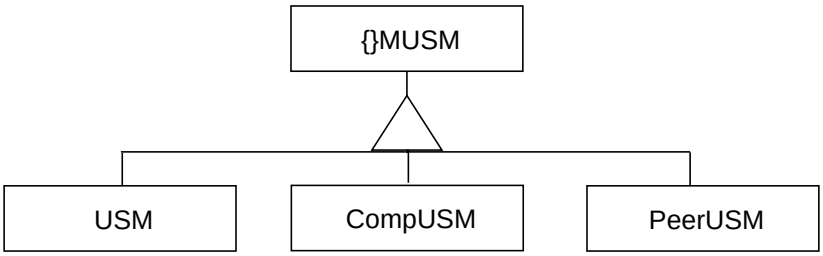


Figure 2-2. Inheritance hierarchy for Member Usage Service Session Manager[4].

The MUSM represents and holds the context of a member (party, resource, provider or peer) in a service session. As the MUSM is an abstract service component, no instances are created. Instances of the appropriate specialized service component are created to represent specific session members.

2.17 Overview of interactions

The following table shows which components interact[5]:

Table 2-1. Overview of interactions

Client \ Server	asUAP	PA	IA	anonUA	namedUA	Sub	ssUAP	SF	SSM	USM	*PeerA*	*CompUSM*	*PeerUSM*	SLCM
asUAP		X												
PA	X		X	X	X		X							
IA				X	X						X			
anonUA		X						X	X	X	X			
namedUA		X				X		X	X	X	X			
Sub					X						X			X
ssUAP		X								X				
SF									X	X		X	X	
SSM				X	X			X		X	X	X	X	
USM				X	X		X		X			X		
PeerA			X		X	X		X	X		X		X	
CompUSM									X	X				
PeerUSM									X		X		X	
SLCM						X		X						

2.18 Accounting Management in the Service Architecture

This section deals with the following two issues;

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

4. {} MUSM means an abstract class without any instance.
5. PeerA, CompUSM, and PeerUSM are not specified yet; entries in the table reflect expected interactions.

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 section.

2.18.1 Overview of Accounting Management in SA

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.

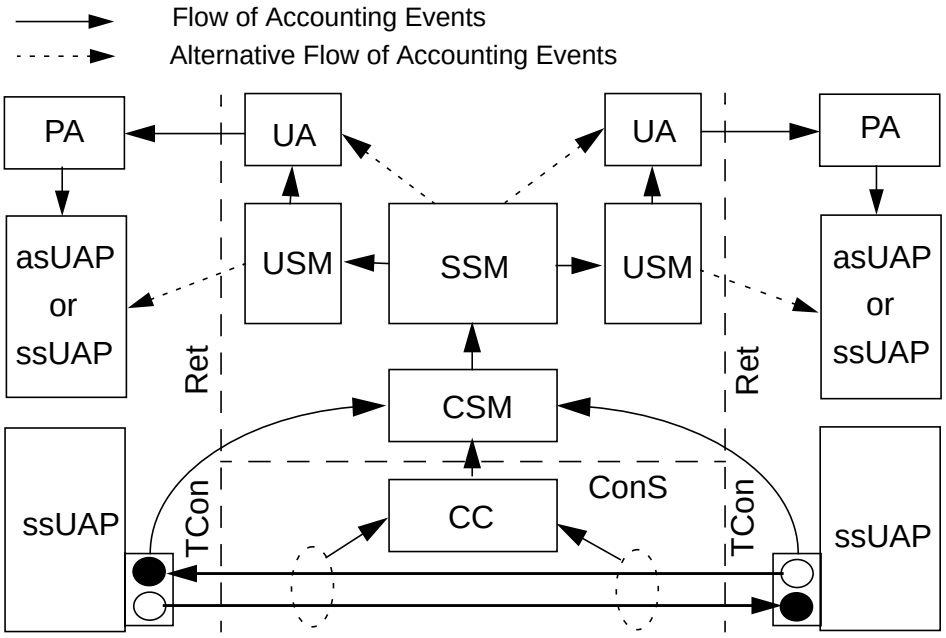


Figure 2-3. Overview of Accounting Management in TINA Service

Figure 2-3 shows an overview of accounting and billing management in a TINA service. Event traces for this scenario are described in Section 4.3.5 and Section 4.4.17. In this scenario, two UAPs in user domains communicate with each other through a bidirectional stream binding.

In the above scenario we make the following assumptions:

- *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 Network Flow End Point (NFEPs) and Stream Flow End Point (SFEPs) are already provisioned and 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.

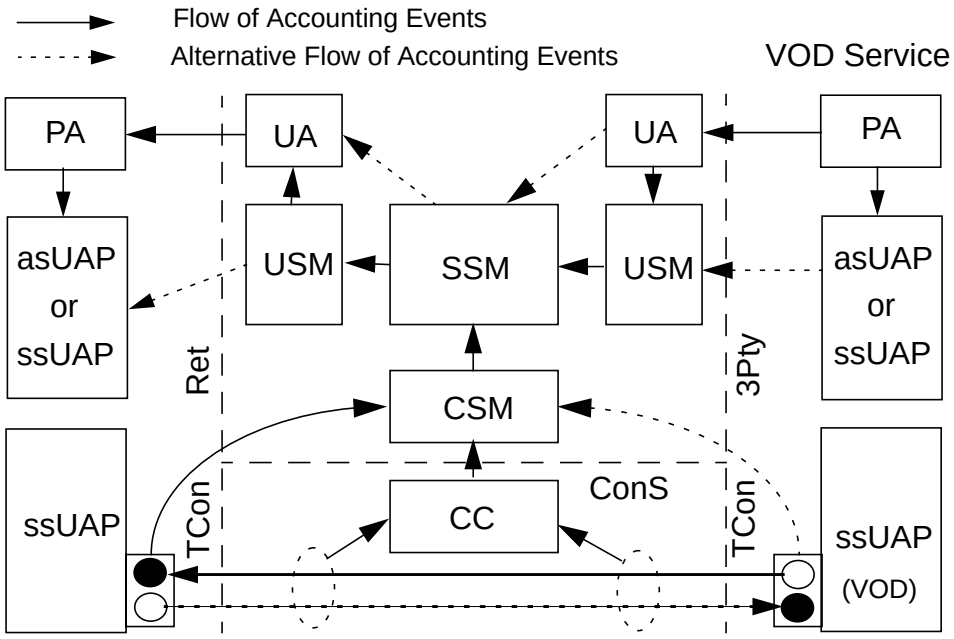


Figure 2-4. An Example of Third-Party Service Provider Accounting

Figure 2-4 illustrates an example 3Pty service provider accounting example. It is to be seen that basic structure of accounting management is exactly the same as Figure 2-3, except that flow of accounting events (and thus billing) are reversed at 3Pty reference point.[6] The accounting concepts introduced in this document, such as management context, service transaction, accounting management ladder, are applicable as they are in this revised figure.

We also assume that service transaction concept is in effect, i. e. billing information may be correlated with performance monitoring 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 connectivity 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 services, implicitly assumed in the current TINA reference points specifi-

6. At this point, very little has been defined on 3Pty reference point, and federation/composition concept in Service Architecture is in general still under developed. The figure is given only as an example, none of the component names should be taken definite.

cations. A billing agent does not necessary imply, however, that the connectivity provider (CP) be made invisible from the users. It can be made visible, i. e. a separate bill from the CP, or it can be made invisible, 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 visible billing context, 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 the visible billing context at the conclusion of service transactions.

2.18.2 Visibility of Billing Context

Before we proceed to explain the scenario depicted in Figure 2-3, 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.

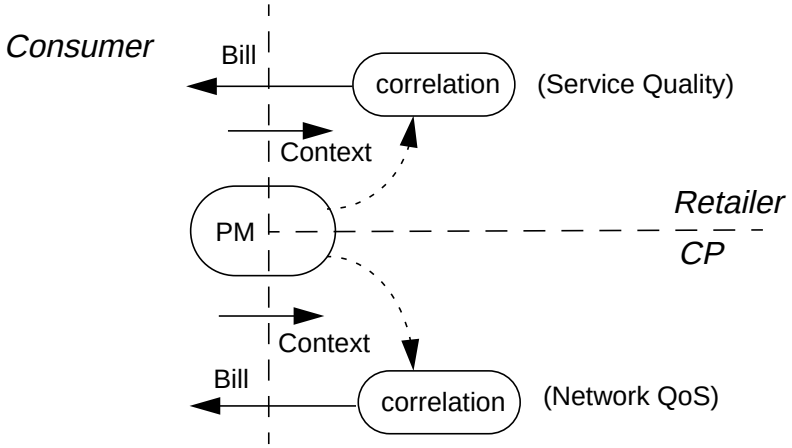


Figure 2-5. Visible Billing Context Example

Figure 2-5 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 [13], and service transaction is explained in TINA service architecture [1].

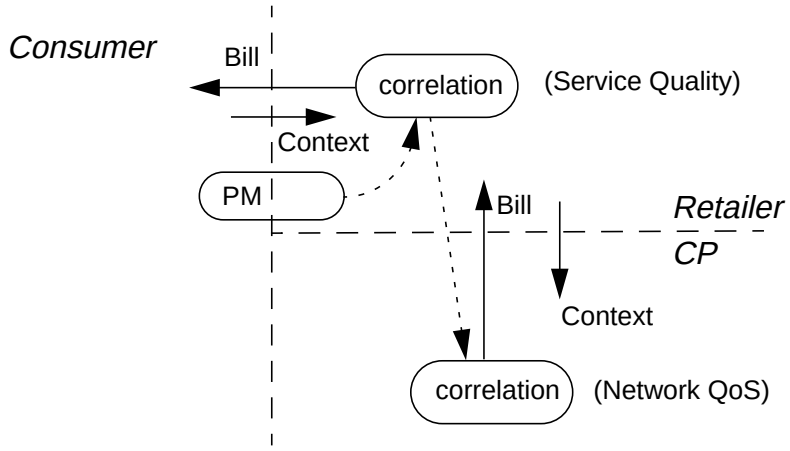


Figure 2-6. Invisible Billing Context Example

Figure 2-6 illustrates an 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.

2.18.3 Different Accounting Management Approaches

The on-line accounting management problem can be addressed in several ways, as any other management service or facility:

- *On-line accounting as an integrated facility*: it is provided through the access session (service independent) components. This is the case currently considered in this document. This facility definition is optional, not prescriptive, as a retailer could choose to follow the approach below.
- *On-line accounting as an ancillary service*: it is supported by service sessions. This case will be considered in Section 4.5.10. Two complementary solutions may be provided:
  - Per session on-line accounting could be supported by an accounting FS (to be defined) to be provided by the USM of those services that wish to provide session on-line accounting.
  - Overall user accounting could be also provided as a separate service (a specific USM/SSM) to control the user billing information in a retailer domain. In this case, an accounting management specific interface should be defined for this service (Section 4.5.10).

2.18.4 Accounting Management Ladder

As it can be easily seen from Figure 2-3, accounting events in on-line accounting are passed from providers (service or connectivity) to the users through service components in the retailer domain, being transformed successively from raw measurement accounting event to billing events. The same is true for Figure 2-4, where a third-party provider charges for its content-based service (in this example VOD), which will be ascribed to the user by adding the connectivity provider's usage charge and the retailer's commission.

2.18.5 Relevance to Resource/DPE Architectures

It would be of benefit to the reader to explain relevance of service level accounting to TINA resource/ DPE architectures.

- *DPE event management (optional):* availability of the DPE event management mechanism, which is to be compliant with CORBA COS event management service and CORBA notification service, is assumed. The current SCS specification, however, uses only push and pull interfaces attached to respective accountable objects (AmcLadderElement).
- *NCS accounting (required):* NCS accounting gives definitions and IDL specs. of basic accounting concepts, which are essential/non-essential accounting event, accounting object, usage metering log manager, etc. Though SCS accounting management section does not necessarily assume TINA resource architecture, i. e. it can be tapped onto a legacy connection management system, generic accounting management components defined in NCS accounting are also used in SCS.
- *NRA accounting management architecture (required):* NRA accounting management architecture explains basic concepts such as accounting management domain, and its usage in NRA. The same concepts are used in this SCS accounting.

We'd like to mention in particular that service components in the retailer domain (SSM, USM, etc.) can be made accountable objects such that their accounting activities can be independently controlled. When this is the case, the accounting management architecture in the retailer domain (SCS accounting) gives a superior example of accounting management ladder, a partially ordered set of accountable objects, which is capable of transforming accounting information into billing information.

Though it is also possible that event communication between service components is done using generic DPE event service, we assumed events are directly pushed from producers to consumers (e.g. from SSM to namedUA), for the purpose of simple engineering realization.

2.19 Management Context

At Ret reference point, a set of management contexts may be agreed and possibly exchanged between User and Retailer at the start of a service transaction, such that a consistent service management is performed throughout the user's session participation. These contexts must be consistent with the ToM (Terms of Management), which is part of the service contract agreed at subscription time [1].

To be more precise, ToM and management context relates in the following manner:

- *ToM:* at subscription time, a set of service profiles are defined as a part of subscription process, which may contain options for management services such as billing or performance management, adding to generic information describing the service itself, e.g. the name of the service, interface names, associated roles, etc. In other words, ToM, a part of the service contract, is represented as a part of the service profiles in that contract.

- *Management context:* management context is usually set-up within an access session, therefore it reflects more of per user or per session management needs of the user. Management context may or may not override ToM represented by a corresponding service profile, depending on the nature of the agreed terms, and the policy enforced by ToM. If management context is not specifically set up between the user and the retailer, however, the service profile will provide a default context, which will be applied to the user and his/her sessions.

Figure 2-7 illustrates structural relationship between ToMs and management contexts.

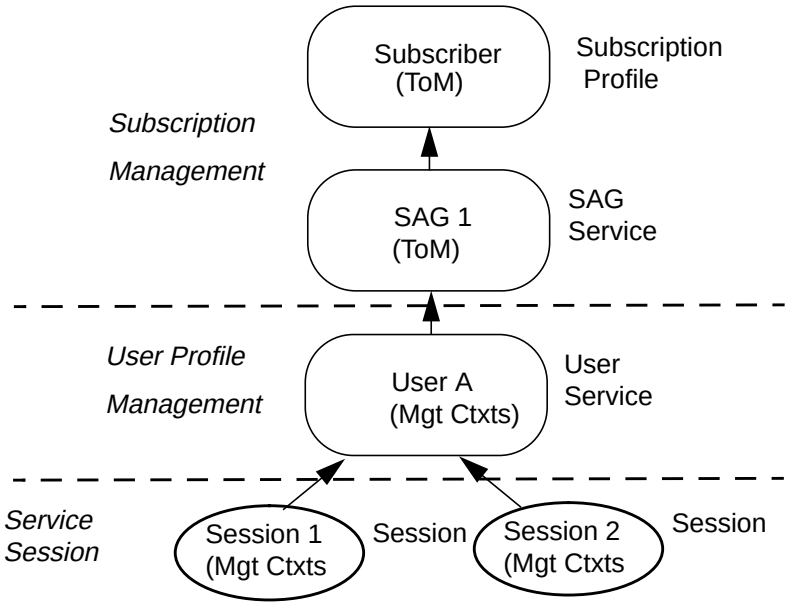


Figure 2-7. Hierarchical Subscriber Management

Subscriber and subscriber assignment groups (SAGs) define a set of service profiles specific to their classes, thus defining their ToMs. They can be seen similar to domains, whose constituencies are subscribers and users, and ToMs and management contexts are a sort of management rules (policies) for those domains. For example, SAG 1 can inherit its ToM (ToM 1) from Subscriber ToM, its parent domain. User A can inherit default context from ToM 1, adding its own requirements, which can be different from other user's.

Service subscriber and SAGs can be indeed made domains, by associating policy rules with them. For example, the top-level subscriber domain can control whether it allows or prohibits its sub-domains (SAG 1, SAG 2, etc. and the associated users) to override its top-level ToM, by asserting it using a policy rule.

3. Detailed Descriptions

This chapter gives detailed descriptions and for each component a table showing which clients are accessing which interfaces and operations and a table giving the description of each operation.

The following components are not fully specified for this version, but some components will be given supplementary explanation at the end of this section (Section 3.14, Yet-To-Be-Defined Components).

- Off-line components:
  - UAF - User Agent Factory
  - SLCM - Service Life Cycle Management
- Access related components
  - PeerA - Peer Agent
  - AnonUA - Anonymous User Agent
  - SecMgr - Security Manager (Please see Section 3.14.1)
- Usage related components
  - MUSM - ‘abstract’ Member Usage Session Manager
  - CompUSM - Composer Usage Session Manager
  - PeerUSM - Peer Usage Session Manager

3.1 asUAP

The User APplication (UAP) SC is defined to model a variety of applications in the User domain. A UAP SC represents one or more of these applications and programs. A UAP can be used by human users, and/or other applications in the user domain. A UAP can be either or both an access session related and service session related SC. The access session related UAP is defined below. The service session related UAP is defined in Section 3.7, "ssUAP".

As an access session related SC, the UAP enables a human user, or another application, to make use of the capabilities of a PA or PeerA, through an appropriate (user) interface. An access session related UAP supports part of the domain access session. The UAP provides capabilities for:

- request authentication information from the user, required by the PA (or PeerA) to set-up an access session with a UA (PeerA),
- the user to request the creation of new service sessions,
- the user to request to join an existing service session,
- alerting the user to invitations, which arrive at the PA or PeerA.

An access session related UAP may also support the following optional capabilities, when they are also supported by the PA or PeerA:

- allow the user to search for a provider, and register as a user of the provider’s services;
- allow the user to search for services and identify providers providing those services.

Zero or more stream interfaces [4] can be attached to a UAP. The stream interfaces can be bound to those in user systems and/or those in providers’ domains.

A user or peer domain contains one or more access session related UAPs. Any access session related UAP can request a PA or PeerA to establish an access session. One or more UAPs interact with a PA or PeerA to use its access session related capabilities within an access session.

A UAP instance may support only access session related capabilities or only service session related capabilities; or it may support both. Access session related UAPs may be specialized by a domain to interact with a specialized PA or PeerA.

Table 3-1. Interfaces, clients and where to find specs

Interface	Client(s)	Event traces	IDL
TINAScsASUAPIntra::i_Access	TINAObjPA	4.4.9 Invite a User to Join a Session	Annex 2.7

Table 3-2. Required interfaces

Server	Interfaces
TINAObjPA (Provider Agent)	TINAScsPAIntra::i_Initial TINAScsPAIntra::i_Access TINAScsPAIntra::i_AccountingPull
TINAObjUSM	TINAScsUSMIntra::i_MgmtCtxt

3.1.1 i_Access interface

`i_Access` interface allows the PA to inform the consumer of events associated with their access sessions, such as invitations, new sessions, etc.

The operations are not detailed here, but some operations can be found in the IDL section.

3.2 PA

The Provider Agent (PA) is a service independent SC, defined as the user's end-point of an access session. The PA is supported in a domain, acting in an access user role. The PA supports a (single) user accessing its UA and making use of services, through an access session. The PA supports the user domain access session, in conjunction with access session related UAPs, and other user domain infrastructure.

Capabilities supported by a PA:

- set-up a trusted relationship between[1] the user and the provider (an access session), by interacting with an Initial Agent[2], and gaining a reference to a UA[3];
- within an access session:
  - convey requests (from a user to a UA) for creating new service sessions,

1. Practically speaking, the user and the provider is in a trusted relationship when they mutually authenticated each other in a cryptographically strong manner, and both parties are ready to engage in a service session which may be associated with financial transactions.
2. The PA may use a location service to find an interface reference for the IA, or some other means. The PA will provide the retailer name, and possibly other information to scope the search of the location service. The capability of the location service to return this interface reference is an important part in enabling access irrespective of location, which is one important feature of personal mobility. This assumes that the location service can indeed be contacted, irrespective of location. Also, it may imply that interworking between location services in different domains is required. How the location service gains an interface reference of an initial agent is undefined. It is likely that the location service has to interact with an object in the provider's domain in order to gain the reference. This interaction is not defined at present.
3. This capability is an important element in support of personal mobility, as it allows a user to access a provider domain from various locations.

- convey request for discovering the services and their session models in order to be sure to have the right ssUAP or download it[4],
- convey requests for joining existing service session,
- receive invitations to join existing service sessions (from a UA) and alert the user[5],
- anonymously make use of a provider's services,
- deploy new components into the user's domain,
- support access to terminal configuration information from a provider's domain,
- register to receive invitations sent when no access session exists.

Operations supported by a PA are service independent.

For each concurrent access session a user has with a provider, there is one PA instance in the user's domain. Each PA may be associated (through an access session) with the same UA, or separate UA instances. A single PA is only ever associated with one UA through an access session. (When no access sessions exist, a user domain can still support a PA. It can be used to initiate an access session, and may receive invitations if registered.)

Table 3-3. Interfaces, clients and where to find specs

Interface	Client(s)	Event traces	IDL
TINAScsPAIntra:: i_Initial	TINAObjASUAP	4.3.1 Contact a provider 4.3.2 Login to a Provider as a Known User 4.3.3 Login to Provider as Anonymous User	Annex 2.8
TINAScsPAIntra:: i_Access	TINAObjASUAP TINAObjASUAP & TINAObjSSUAP	4.3.4 Logout from a Provider 4.3.6 List Subscribed Services 4.4.7 Resume a Service Session 4.4.8 Resume Participation in a Service Session 4.4.9 Invite a User to Join a Session (case 3) 4.5.6 Register to receive invitations outside of an access session 4.4.1 Start a Service Session and Selection Of session model	Annex 2.8
TINAProviderInitial:: i_ProviderAuthenticate[a]	TINAObjASUAP	4.3.2 Login to a Provider as a Known User	Ret-RP
TINAScsPAIntra:: i_AccountingPull	TINAObjASUAP	4.3.5 Check Accounting Information	Annex 2.8
TINAUserInitial:: i_UserInitial	TINAObjNamedUA & TINAObjAnonUA	4.4.9 Invite a User to Join a Session (case 2)	Ret-RP
TINAUserAccess:: i_UserAccess	TINAObjNamedUA & TINAObjAnonUA		Ret-RP
TINAUserAccess:: i_UserTerminal	TINAObjNamedUA & TINAObjAnonUA		Ret-RP
TINAUserAccess:: i_UserInvite	TINAObjNamedUA & TINAObjAnonUA	4.4.9 Invite a User to Join a Session (case 1)	Ret-RP
TINAUserAccess:: i_UserAccessSessionInfo	TINAObjNamedUA & TINAObjAnonUA		Ret-RP

4. For each service, the retailer might offer an associated service to download the ssUAP.
5. Using an access session related UAP.

Table 3-3. Interfaces, clients and where to find specs

Interface	Client(s)	Event traces	IDL
TINAUserAccess:: i_UserSessionInfo	TINAObjNamedUA & TINAObjAnonUA		Ret-RP
TINAScsPAIntra:: i_Init	TBD	None for this version	Annex 2.8

a. The interface TINAProviderInitial::i_ProviderAuthenticate might be slightly different from the one defined in Ret-RP. This issue needs further investigation.

Table 3-4. Required interfaces

Server	Interfaces
TINAObjASUAP	TINAScsASUAPIntra::i_Access
TINAObjSSUAP	TINAScsSSUAPIntra::i_AccessInitialise
TINAObjIA	TINAProviderInitial::i_ProviderInitial TINAProviderInitial::i_ProviderAuthenticate
TINAObjNamedUA	TINAProviderAccess::i_ProviderNamedAccess TINAScsNamedUAIntra::i_AccountingPull TINAProviderAccess::i_DiscoverServicesIterator
TINAObjAnonUA	TINAProviderAccess::i_ProviderAnonAccess

3.2.1 i_Initial interface

- **`contactProvider`**() allows the consumer to contact a retailer. The PA offers this operation to an asUAP, to contact a named provider, and so allow the consumer to establish and access session.
- **`requestNamedAccess`**() allows the consumer to identify himself to the retailer, and establish an access session. A secure context may have already been set-up between the consumer and the retailer using CORBA security services. In this case, this operation returns a reference to a `i_RetailerNamedAccess` interface. If the consumer has not already been authenticated, then an `e_AuthenticationError` exception will be raised. This contains a reference to a `i_RetailerAuthenticate` interface, which may be used to authenticate and set-up the secure context. Then this operation can be invoked again to retrieve the reference to the `i_RetailerNamedAccess` interface.
- **`requestAnonymousAccess`**() allows the consumer to establish an access session with the retailer without revealing his identity. The access session will provide access to some services, although the consumer may need to negotiate with the retailer over which services are available. (The services will obviously not be specialised to the consumer.) The consumer interacts with the retailer through a `i_RetailerAnonAccess` interface. This operation is otherwise the same as `requestNamedAccess`().

3.2.2 i_Access interface

`i_Access` interface allows a known consumer access to his subscribed services. The consumer uses it for all operations within a single access session with the retailer[6]. This interface is returned when the consumer has been authenticated by the retailer and an access session has been established. The interface reference is returned by calling `requestNamedAccess`() on the `i_Initial` interface.

This interface allows also the client (usually the asUAP) to register/unregister interfaces to be used within or outside the access session by the PA. It provides the following operations:

- **`registerInterface`**() - allows the asUAP interface to be registered for use within the current access session. The registrations ends when the access session ends, or when the `unregisterInterface`() operation is called. An interfaceIndex is returned to allow the interface to be unregistered.
- **`registerInterfaceOutsideAccessSession`**() - allows the asUAP to register an interface for use outside an access session. (The interface registered should still be available when no access session exists between the user and provider).
- **`listRegisteredInterfaces`**() - allows the asUAP to list the interfaces which have been registered with the PA by her. The list defines which interfaces are registered for use inside an access session, and which are for use outside.
- **`unregisterInterface`**() - allows the asUAP to unregister an interface, so that the PA will not attempt to use that interface, (either inside or outside the access session).
- **`registerInterfaces`**() - allows the client to register a list of interfaces to use within the current access session. The registrations ends when the access session ends, or when the `unregisterInterfaces`() operation is called. An interfaceIndexList is returned to allow the interface to be unregistered.
- **`unregisterInterfaces`**() - allows the clientto unregister alist of interfaces, so that the PA will not attempt to use that interface, (either inside or outside the access session).
- **`listAccessSessions`**( ) - allows the consumer in this access session to find out about other access sessions that he has with this retailer. (e.g. A consumer is at work, but has an access session set up at home which runs an active security service session.)
- **`endAccessSession`**( ) - allows the consumer to end a specified access session, either the current one, or another, found using `listAccessSessions`(). The consumer can also specify some actions to do if there are active service sessions.
- **`getUserInfo`**( ) - gets the consumer's username, and other properties.
- **`getUserCtxtNames`**( ) - retrieves the names of the user contexts registered by the user agent.
- **`getUserCtxtNamesAccessSessions`**( ) - retrieves the names of the user contexts requested by the access session that are associated with given access sessions.
- **`listSubscribedServices`**( )- lists the services to which the consumer is subscribed. Scoping of subscribed services can be done using property lists. The operation returns sufficient information for the consumer to start a particular (subscribed)service.

6. Here we make the following implementation assumption: each PA deals with only a single consumer, and provides a separate interface for each access session, and so does each asUAP.

- **discoverServices**( ) - lists all the services available from the retailer. The consumer can scope the list by suppling some properties that the service should have, and a maximum number to return. A reference to an `i_DiscoverServicesIterator` interface can be used to retrieve the remaining services.
- **listServiceSessions**( ) - lists the service sessions of the consumer. The request can be scoped by the access session, and session properties, (e.g. active, suspended, service type, etc.).
- **getSession{Models,InterfaceTypes,Interface,Interfaces}**( ) - all retrieve information on a particular session.
- **listSessionInvitations**( ) - lists the invitations to join a service session that have been sent to the consumer.
- **listSessionAnnouncements**( ) - lists the service sessions with have been announced. It can be scoped by some announcement properties.
- **startService**( ) - allows the consumer to start a service session. It is used by a ssUAP to start a service session, and have interfaces associated with the session returned to it.
- **startServiceWithUAP**( ) - allows the consumer to start a service session, using a specified UAP. It is used by an asUAP to (possibly download and then) launch a ssUAP, which will be used with the service session.
- **endSession**( ) - allows the consumer to end a service session.
- **endMyParticipation**( ) - allows the consumer to end his participation in a service session.
- **suspendSession**( ) - allows the consumer to suspend a service session.
- **suspendMyParticipation**( ) - allows the consumer to suspend his participation in a service session.
- **resumeSession**( ) - allows the consumer to resume a service session.
- **resumeMyParticipation**( ) - allows the consumer to resume his participation in a service session.
- **joinSessionWithInvitation**( ) - allows the consumer to join a service session, to which he has been invited.
- **joinSessionWithAnnouncement**( ) - allows the consumer to join a service session, which has been announced.
- **replyToInvitation**( ) - allows the consumer to reply to an invitation. It can be used to inform the service session to which they have been invited, that they will/will not be joining the session, or to send the invitation somewhere else. (It does not allow the consumer to join the session.)
- **receiveInvitationsOutsideAccessSession**( ) - allows the consumer to request that invitations are sent to a specified user context, when this access session ends.

3.2.3 i_AccountingPull

This interface allows the asUAP to retrieve accounting data from the PA. It provides the following operations:

- **GetUserLogEntries**() - allows asUAP to retrieve accounting data about the user specifying a time interval.
- **GetSessionLogEntries**() - allows asUAP to retrieve accounting data about a specific service session the user is/has been taking part in.

3.3 IA

An Initial Agent (IA) is the initial access point to a domain. An IA reference is returned to the requesting domain (PA or PeerA) when it wishes to contact the domain. The IA supports capabilities to:

- authenticate the requesting domain and set up a trusted relationship between the domains (an access session) by interacting with the PA or PeerA,
- establish an access session, but allowing the requesting domain to remain anonymous. The type of UA accessed in this way is an anonymous user agent.

An IA supports requests from one PA/PeerA at a time. The PA/PeerA requests to contact the domain and is given a reference to an IA. When the PA/PeerA has interacted with the IA to establish an access session with a UA/PeerA, the reference to the IA may become invalid. Subsequently, the IA may be contacted by another PA/PeerA. It relies on interfaces on the UA (both named and anonymous) and the PeerA for initialization purposes and an interface on an authentication server (not specified by TINA).

Table 3-5. Interfaces, clients and where to find specs

Interface	Client(s)	Event traces	IDL
TINAProviderInitial:: i_ProviderInitial	TINAObjPeerA TINAOnjPA	4.3.2 Login to a Provider as a Known User 4.3.3 Login to Provider as Anonymous User	Ret section C.3
TINAProviderInitial:: i_ProviderAuthenticate	TINAObjPeerA TINAObjPA	4.3.2 Login to a Provider as a Known User 4.3.3 Login to Provider as Anonymous User	Ret section C.3

Table 3-6. Required interfaces

Server	Interfaces
TINAObjAnonUA	i_Initial
TINAObjNamedUA	i_Initial
TINAObjPeerA	*i_Initial (unspecified)*
Authentication Server	*Currently unspecified*

Both `i_ProviderInitial` and `i_ProviderAuthenticate` are defined as in Ret [12].

3.4 namedUA

The namedUA supports the following capabilities:

- Within an access session
  - Act as a single contact point to control and manage (create/suspend/resume/delete) the life-cycle of service sessions and user service sessions, taking into account restrictions posed by the subscriber and the user.
  - Suspend/resume existing user service sessions and service sessions. This includes support for session mobility.
  - Manage the user’s preferences (choices or constraints) on service access and service execution (this is supported by starting a provider specific service session.).

- Resolve the service execution environment for the user, allowing him/her to use services from many different types of terminals. This requires resource configuration information of the user system (which includes terminals and their access points being used by or available for the user; access to this information may be restricted by the user/PA.) This includes support for personal mobility.
- Register user at a terminal to receive invitations. This includes support for personal mobility.
- Allow the user to define user private/public policies (this is supported by starting a provider specific service session).
- Negotiate the session models and feature sets supported by a service session, in order for it to interact with a UAP in the user's domain.

• Accept invitations from users to join a service session.

• Deliver invitations to a terminal, previously registered by the user with the namedUA. No access session would be required to allow this delivery of invitations.

The namedUA may support the following optional capabilities:

• Perform actions on behalf of the user, when the user is not in an access session with the namedUA.

• Initiate an access session with a PA.

• Support additional authentication of the user. This may be tailored to the user and the usage context.

Table 3-7. Interfaces, clients and where to find specs

Interface	Client(s)	Event traces	IDL
TINAProviderAccess:: i_ProviderNamedAccess	TINAObjPA	4.3.2 Login to a Provider as a Known User 4.3.4 Logout from a Provider 4.4.1 Start a Service Session and Selection Of session model 4.4.8 Resume Participation in a Service Session 4.4.7 Resume a Service Session	Ret Annex A section 0.3
TINAScsNamedUAIntra:: i_Initial	TINAObjIA		Annex 2.9
TINAScsNamedUAIntra:: i_SessionInfo	TINAObjUSM	4.4.6 Suspend Participation in a Service Session 4.4.3 End Service Session via Access Session 4.4.5 Suspend a Service Session 4.4.2 End a Service Session	Annex 2.9
TINAScsNamedUAIntra:: i_InvitationDelivery	TINAObjSSM TINAObjPeerA	4.4.9 Invite a User to Join a Session	Annex 2.9
TINAProviderAccess:: i_DiscoverServicesIterator	TINAObjPA	None for this version	Ret-RP
TINAProviderAccess:: i_ProviderAccess	TINAObjPA	None for this version	Ret-RP
TINAProviderAccess:: i_ProviderAccessGetInterfaces	TINAObjPA	None for this version	Ret-RP

Table 3-7. Interfaces, clients and where to find specs

Interface	Client(s)	Event traces	IDL
TINAProviderAccess:: i_ProviderAccessInterfaces	TINAObjPA	None for this version	Ret-RP
TINAProviderAccess:: i_ProviderAccessRegisterInterfaces	TINAObjPA	None for this version	Ret-RP
TINAScsNamedUAIntra:: i_AccountingPull	TINAObjASUAP TINAObjPA	4.3.5 Check Accounting Information	Annex 2.9
TINAScsNamedUAIntra:: i_AccountingPush	TINAObjSSM TINAObjUSM	4.4.17 Service Session Accounting	Annex 2.9
TINAScsNamedUAIntra:: i_SubscriptionNotify	TINAObjSub	4.5.1 Subscribe a New Customer 4.5.2 Modify Subscriber Information 4.5.3 Contract a New Service 4.5.4 Modify Service Contract 4.5.5 Unsubscribe	Annex 2.9
TINAScsNamedUAIntra:: i_ServiceProfileCustomization	TBD	None for this version	Annex 2.9
TINAScsNamedUAIntra:: i_UserProfileManagement	TBD	None for this version	Annex 2.9
TINAScsNamedUAIntra:: i_Init	UAF (TBD)	None for this version	Annex 2.9

Table 3-8. Required interfaces

Server	Interfaces
TINAObjPA	TINAUserAccess::i_UserInvite TINAUserAccess::i_UserAccess TINAUserAccess::i_UserTerminal TINAUserAccess::i_UserAccessSessionInfo TINAUserAccess::i_UserSessionInfo TINAUserInitial::i_UserInitial
TINAObjSSM	TINAScsSSMIntra::i_Join
TINAObjSF	TINAScsSF::i_SSCreate TINAScsSF::i_SSManage TINAScsSF::i_Resume
TINAObjSub	TINAScsSubInitial::i_InitialAccess TINAScsSubscriberInfoAccess::i_SubscriberInfoQuery

3.4.1 i_ProviderNamedAccess interface

`i_ProviderNamedAccess` interface allows a known user access to his subscribed services. The user uses it for all operations within an access session with the provider. This interface is returned when the user has been authenticated by the provider and an access session has been established. It is returned by calling `requestNamedAccess`() on the `i_ProviderInitial` interface.

It provides the following operations:

- **`setUserContext`**`()` - allows the user to inform the provider about interfaces in the user domain, and other user domain information. (e.g. user applications available in the user domain, operating system used, etc.). It should be called immediately after receiving the reference to this interface, or subsequent operations may raise an exception.
- **`listAccessSessions`**`()` - allows the user in this access session to find out about other access sessions that he has with this provider. (e.g. A user is at work, but has an access session set up at home which runs an active security service session.)
- **`endAccessSession`**`()` - allows the user to end a specified access session, either the current one, or another, found using `listAccessSessions`(). The user can also specify some actions to do if there are active service sessions.
- **`getUserInfo`**`()` - gets the user's username, and other properties.
- `listSubscribedServices()`- lists the services to which the user is subscribed. Scoping of subscribed services can be done using property lists. The operation returns sufficient information for the user to start a particular (subscribed) service.
- **`discoverServices`**`()` - lists all the services available from the provider. The user can scope the list by suppling some properties that the service should have, and a maximum number to return. A reference to an `i_DiscoverServicesIterator` interface can be used to retrieve the remaining services.
- **`listServiceSessions`**`()` - lists the service sessions of the user. The request can be scoped by the access session, and session properties, (e.g. active, suspended, service type, etc.).
- **`getSession{Models,InterfaceTypes,Interface,Interfaces}`**`()` - all retrieve information on a particular session.
- **`listSessionInvitations`**`()` - lists the invitations to join a service session that have been sent to the user.
- **`listSessionAnnouncements`**`()` - lists the service sessions with have been announced. It can be scoped by some announcement properties.
- **`startService`**`()` - allows the user to start a service session.
- **`endSession`**`()` - allows the user to end a service session.
- **`endMyParticipation`**`()` - allows the user to end his participation in a service session.
- **`suspendSession`**`()` - allows the user to suspend a service session.
- **`suspendMyParticipation`**`()` - allows the user to suspend his participation in a service session.
- **`resumeSession`**`()` - allows the user to resume a service session.
- **`resumeMyParticipation`**`()` - allows the user to resume his participation in a service session.
- **`joinSessionWithInvitation`**`()` - allows the user to join a service session, to which he has been invited.
- **`joinSessionWithAnnouncement`**`()` - allows the user to join a service session, which has been announced.
- **`replyToInvitation`**`()` - allows the user to reply to an invitation. It can be used to inform the service session to which they have been invited, that they will/will not be joining the session, or to send the invitation somewhere else. (It does not allow the user to join the session.)

- **`getInterfaceTypes`**() - allows the user to discover all of the interface types supported by the provider domain.
- **`getInterface`**() - allows the user to retrieve an interface reference, giving the interface type and properties.
- **`getInterfaces`**() - allows the user to retrieve a list of all the interfaces supported by the provider.
- **`registerInterface`**() - allows a user interface to be registered for use within the current access session. The registrations ends when the access session ends, or when the `unregisterInterface`() operation is called. An interfaceIndex is returned to allow the interface to be unregistered.
- **`registerInterfaceOutsideAccessSession`**() - allows a user to register an interface for use outside an access session. (The interface registered should still be available when no access session exists between the user and provider).
- **`listRegisteredInterfaces`**() - allows the user to list the interfaces which have been registered with the provider by her. The list defines which interfaces are registered for use inside an access session, and which are for use outside.
- **`unregisterInterface`**() - allows the user to unregister an interface, so that the provider will not attempt to use that interface, (either inside or outside the access session).

3.4.2 i_Initial

This interface allows its clients to get a reference of interface i_ProviderNamedAccess that is necessary for access session interaction to take place.

It provides the following operation:

`setupAccessSession`() - allows its clients to get a reference of interface i_ProviderNamedAccess for an access session and the access session identifiers.

3.4.3 i_SessionInfo

This interface allows its clients to give the namedUA the information necessary to keep an updated list of sessions the corresponding user is participating in and the status of both the session and the user's participation.

It provides the following operations:

- **`participationSuspended`**() - allows its clients to notify the namedUA of the suspension of the user's participation in a service session, and provide it with a reference to an interface of a SF that will be used for resuming the participation and the relevant accounting information for the suspended service session.
- **`participationEnded`**() - allows its clients to notify the namedUA of the end of the user's participation in a service session, and provide it with the relevant accounting information for the ended service session.
- **`sessionSuspended`**() - allows its clients to notify the namedUA of the suspension of a service session where the user was taking part, and provide it with a reference to an interface of a SF that will be used for resuming the service session and the relevant accounting information for the ended service session.
- **`sessionEnded`**() - allows its clients to notify the namedUA of the end of a service session where the user was taking part, and provide it with the relevant accounting information for the ended service session.

- **sessionResumed**() - allows its clients to notify the namedUA that a previously suspended service session, where the user was taking part, has been resumed.

3.4.4 i_InvitationDelivery

This interface allows its clients to send invitations to the namedUA's user or cancel them.

It provides the following operations:

- **Invite**() - allows its clients to send an invitation to the service session to the namedUA's corresponding user. An identifier for the service the user is being invited to join a session of, as well as the name of the inviting party, the purpose of the session, and the reason for the invitation, are included in the invocation to allow for different invitation screening policies. An invitation identifier unique for this user and provider is given as well.
- **Cancel**() - allows its clients to cancel an invitation previously issued for reasons like the end of the service session before the invited user joins; the unique invitation identifier is passed to identify the invitation to be cancelled.

3.4.5 i_AccountingPull

This interface allows its clients to retrieve accounting data from the namedUA.

It provides the following operations:

- **GetUserLogEntries**() - allows its clients to retrieve accounting data about the namedUA's user specifying a time interval.
- **GetSessionLogEntries**() - allows its clients to retrieve accounting data about a specific service session the namedUA's user is/has been taking part of.

3.4.6 i_AccountingPush

This interface allows its clients to store accounting data in the namedUA.

It provides the following operations:

- **StoreBillingEvent**() - allows its clients to store a specific billing event in the namedUA.
- **StoreBillingEventList**() - allows its clients to store a list of billing events in the namedUA.
- **RemoveBillingEvent**() - allows its clients to remove a specified billing event previously stored in the namedUA.
- **RemoveBillingEventList**() - allows its clients to remove a list of billing events previously stored in the namedUA.
- **RemoveUserLogEntries**() - allows its clients to remove log entries corresponding to a specified time interval.

3.4.7 i_ServiceProfileCustomization

This interface allows the customization of user service profiles. The user service profile includes a service part, describing customized service characteristics, and a service management part, detailing the management contexts for the different management areas (FCAPS).

The main operations are:

- **getUserServiceProfile**() - It returns the service profile defined for the user for a specific service. If it has not been customized yet it will match with the SAG service profile corresponding to the SAG the user belongs to (for that service).

- **`setUserServiceProfile`**() - It allows to define the user service profile for a specific service. If it is consistent with the SAG Service Profile (profile for the group the user belongs to) the operation will succeed. If not, an exception is returned.

3.4.8 i_UserProfileManagement

This interface allows to manage the user profile. This profile contains information like: *usage context*, defining the user location (and/or terminal) registration and local context in every location (and/or terminal), and *personal configuration*, like invitation handling policies and registration schedule. The usage context allows the client to locate the user and know about the context (terminal type, NAP type, and available service capabilities) in the current location. The personal preferences allow to model the behavior of service components in access and service sessions depending on certain context conditions (time, date, location, session owner or participants, etc.).

It provides the following operations:

- **`getUserInformation`**( ):.
- **`setUserInformation`**():.

3.4.9 i_SubscriptionNotify

This interface allows its clients to notify the namedUA about new, modified or withdrawn services in the portfolio of the namedUA's corresponding user.

It provides the following operation:

- **`Notify`**() - allows its clients to notify the namedUA of new services added to the user's portfolio, or modifications or withdrawal of existing ones. It includes the list of services to which the notification refers and their corresponding service profiles.

3.5 anonUA

Editor's note: Not completed, no IDL available...

An Anonymous User Agent (anonUA) is a service independent SC that represents a user in the provider's domain. The anonUA is a specialization of UA for users that do not wish to disclose their identity to the provider. It is the provider domain's end-point of an access session with the anonymous user.

The anonUA supports all of the capabilities which are defined for UA. In addition, it supports the following capabilities:

- Support a trusted relationship between the user and the provider (an access session) by referencing the user's PA. The provider does not know the identity of the user. ('Trust' is not guaranteed by identifying the user, as for the namedUA, but may be ensured by, for example, pre-payment.);
- within an access session:
  - Suspend/resume existing user service sessions and service sessions within an access session. (Suspended sessions cannot be resumed in a different access session.[7]);
  - Manage the user's preferences (choices or constraints) on service access and service execution. (These would have to be determined during the access session, and could not be re-used in a separate access session.);

7. It is assumed suspended sessions are ended by the provider if the access session is ended.

- Provide access to a user's contract information with the provider. (This contract would be defined at the start of the access session and terminated at the end of the access session.);
- Define user private /public policies. (This may be supported by starting a provider specific service session. This information would only be maintained during this access session.);
- Allow the anonymous user to register as a user of the provider (i.e., set-up a contract with the provider for longer than a single access session);
- Negotiate the session models and feature sets supported by a service session, in order for it to interact with a UAP in the user's domain.

The anonUA provides no support for personal or session mobility.

Table 3-9. Interfaces, clients and where to find specs

Interface	Client(s)	Event traces	IDL
i_Access	TINAObjPA	4.3.3 Login to Provider as Anonymous User	Ret
i_AccountingPull	TINAObjPA		
i_Initial	TINAObjIA		

3.5.1 i_ProviderAnonAccess interface

This interface defines no operations. It inherits from `i_ProviderAccess` defined in Ret.

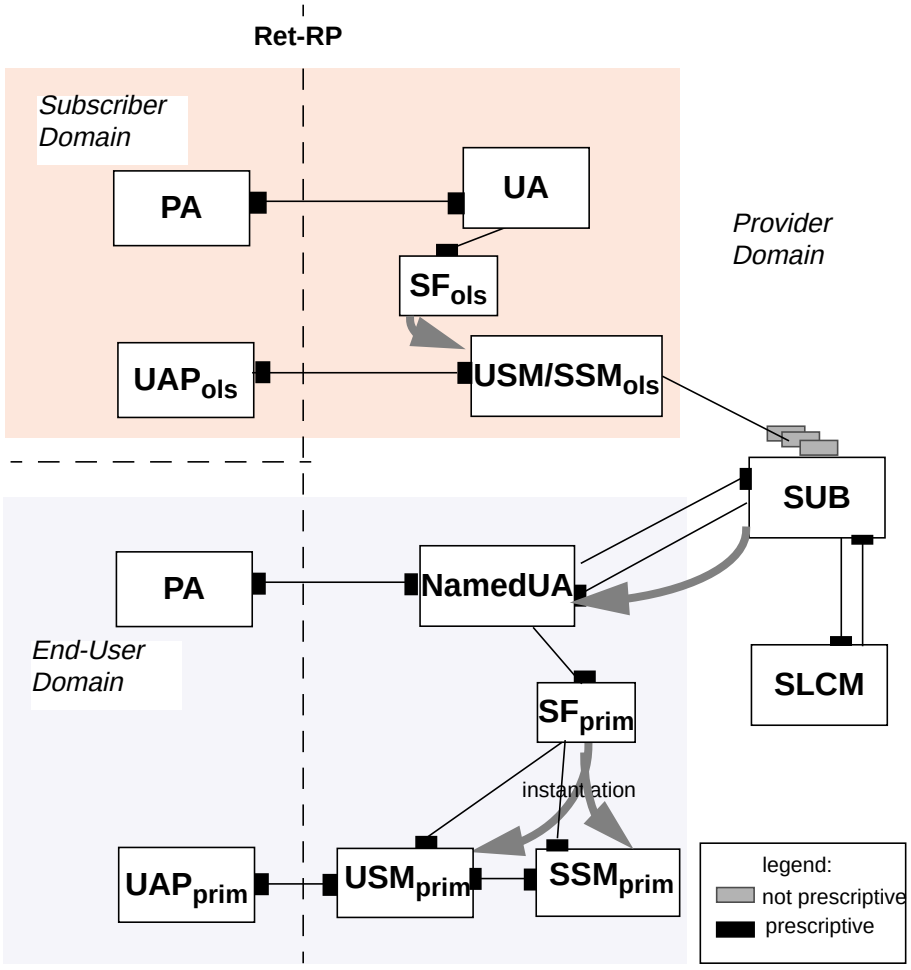
3.5.2 i_AccountingPull

3.5.3 i_Initial

3.6 Sub - Subscription Management Component

The Subscription Management Component (Sub) allows the management of subscribers, subscriptions and users for the whole set of services provided by a provider. The main functionality offered by this component is:

- creation, modification, deletion and query of subscribers,
- creation, modification, deletion and query of subscriber related information (associated end users, end user groups, etc.),
- creation, modification, deletion and query of service contracts (definition of subscribed service profiles),
- retrieval of the list of services, either the ones available in the provider domain or the subscribed ones,
- retrieval of the service profile (SAGServiceProfile) for a specific user (or terminal or NAP).
- creation and deletion of access components,
- notification of changes to the affected users' access components.



Note: *The USM/SSM_{ols} is a specific session manager for the on-line subscription management service. Interfaces between SUB and this component are described but NOT prescribed.*
USM_{prim} and SSM_{prim} are the session managers for the primary services (those the subscriber contracts and his associated end-users use).

Figure 3-1. Overview of Sub interfaces.

The figure shows the interactions of Sub with the rest of service components. Some of these interactions are described but not prescribed here. The rationale behind this decision is the belief that most providers will keep on using their subscriber databases and legacy interfaces for accessing them, but will have to interact with TINA service components (access components and SLCM) in a standard way.

In order to allow the insertion, modification and deletion of the subscription data Sub provides interfaces to specific clients. These clients can be specialized service session components supporting an on-line subscription management service or off-line provider subscription management applications. The functionality provided by this component includes:

- new customers subscription and contract of new services (interface `i_Subscribe`),
- handling of service contract information (`i_ServiceContractInfoMgmt`), and
- handling of subscriber information (`i_SubscriberInfoMgmt`)

The access and service sessions have to behave in line with the contracted service characteristics. For that reason, Sub stores the subscription information representing the terms of the contract and offers the `i_SubscriberInfoQuery` to retrieve it from the access components.

The `i_ServiceNotify` interface is offered to the SLCM to receive notifications about services newly deployed or upgraded that are available in the service network for subscription and use, and about the withdrawal of previously existing ones.

Table 3-10. Interfaces, clients and where to find specs

Interface	Client(s)	Event traces	IDL
TINAScsSubscriberInfoAccess:: i_SubscriberInfoQuery	TINAObjNamedUA TINAObjPeerA	4.3.6 List Subscribed Services	Annex 2.14
TINAScsSubInitial:: i_Subscribe	TINAObjUSM/SSM_{OLS}	4.5.1 Subscribe a New Customer 4.5.2 Modify Subscriber Information 4.5.5 Unsubscribe 4.5.3 Contract a New Service 4.5.4 Modify Service Contract	Annex 2.13
TINAScsServiceContractInfoAccess:: i_ServiceContractInfoMgmt	TINAObjUSM/SSM_{OLS}	4.5.1 Subscribe a New Customer 4.5.3 Contract a New Service 4.5.4 Modify Service Contract 4.5.5 Unsubscribe	Annex 2.10
TINAScsSubscriberInfoAccess:: i_SubscriberInfoMgmt	TINAObjUSM/SSM_{OLS}	4.5.1 Subscribe a New Customer 4.5.2 Modify Subscriber Information 4.5.5 Unsubscribe 4.5.3 Contract a New Service 4.5.4 Modify Service Contract	Annex 2.14
TINAScsSubInitial:: i_ServiceNotify	TINAObjSLCM	4.5.7 Register a new service 4.5.8 Modify an existing service 4.5.9 Withdraw a service	(Annex 2.13)
TINAScsSubInitial:: i_InitialAccess	TINAObjNamedUA TINAObjPeerA TINAObjUSM/SSM_{OLS}	(Not shown)	Annex 2.13

Table 3-11. Required interfaces

Server	Interfaces
TINAObjNamedUA	TINAScsNamedUAIntra::i_SubscriptionNotify
TINAObjSLCM	TINAScsServiceLCMgmt::i_ServiceQuery

Other interface (`i_InitialAccess`) is offered to allow clients to retrieve the interfaces according to their needs.

3.6.1 Subscription Management Type Definitions

In this section, the IDL definition of the information required to handle subscriptions, subscribers and end-users in a provider domain is included. This will allow to understand more clearly the interface descriptions.

Figure 3-2 represents mainly the relationship between a service and a subscriber, described in terms of a number of service profiles (service template, subscription profile and SAG service profile). This information model is described in [Information Model document].

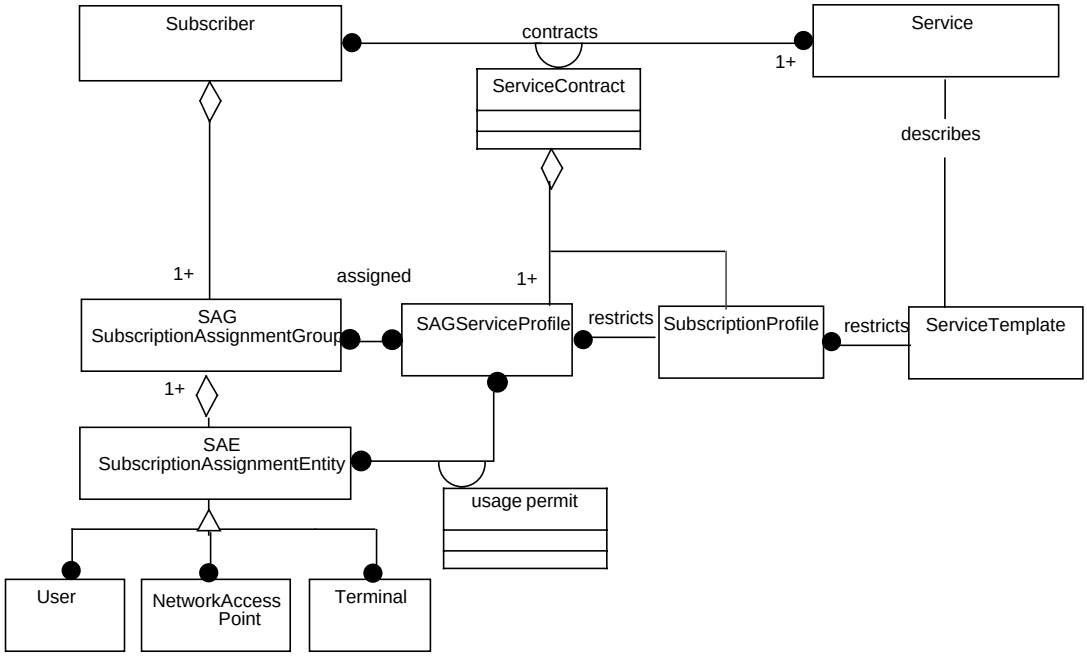


Figure 3-2. Subscription Management Information Model

A *Subscriber contracts* a number of *Services*; at least one to be considered as such. The information associated with a subscriber is:

```
struct t_Subscriber {
        t_AccountNumber            accountNumber;
        TINACommonTypes::t_UserId  subscriberName;
        t_Person                   identificationInfo;
        t_Person                   billingContactPoint;
        string                     RatePlan;
        any                        paymentRecord;
        any                        credit;
};
```

The `accountNumber` is generated by the provider and unique in its domain. It is used inside the provider domain to identify the subscriber and, probably, in the bills as well. The `subscriberName` is the name the subscriber wants to be named by[8]. It will be usually more friendly than the `accountNumber`. There will be a one-to-one mapping between these two identifiers. The field `identificationInfo` stores information like subscriber name, address, etc. The `billingContactPoint` keeps the information required to send the invoices for billing. The `paymentRecord` contains information about last paid bills to check the billing status. The `credit` field stores information about deposits, credits granted to the subscriber, etc.

8. It might be used to generate user identifiers. For instance, user X in provider A could be given an identifier like X_A.

The agreed *Service Contract* defines the conditions of the service provision for each of the service subscriptions. It is defined as:

```
struct t_ServiceContract {
        TINAAccessCommonTypes::t_ServiceId   serviceId;
        t_AccountNumber                     accountNumber;
        short                               maxNumOfServiceProfiles;
        t_DateTime                          actualStart;
        t_DateTime                          requestedStart;
        t_Person                            requester;
        t_Person                            technicalContactPoint;
        t_AuthLimit                         authorityLimit;
        t_SubscriptionProfile               subscriptionProfile;
        t_SagServiceProfileList             sagServiceProfileList;
};
```

The `serviceId` and the `accountNumber`, together, identify uniquely a service contract. The profiles are the main part of the service contract. Other fields provide additional information about the contract (starting date, requested starting date, requester, technical contact point, etc.).

A *Service Template describes* the characteristics of the *Service* provided by the provider.

```
struct t_ServiceTemplate {
        TINAAccessCommonTypes::t_ServiceId          serviceInstanceId;
        TINAAccessCommonTypes::t_UserServiceName    serviceInstanceName;
        t_ServiceIdList                             requiredServices;
        t_ServiceDescription                        serviceDescription;
};
```

The *Service Description* contains the characteristics of a generic service type. It is reused in the service template to describe the characteristics of a specific service instance (particular implementation of a service type) and in the service profiles to represent the characteristics of the service contracted by a subscriber (for the whole set of associated users or for a group of them).

```
struct t_ServiceDescription {
        TINAAccessCommonTypes::t_ServiceId           serviceTypeId;
        TINAAccessCommonTypes::t_UserServiceName     serviceTypeName;
        t_ParameterList                              serviceCommonParams;
        t_ParameterList                              serviceSpecificParams;
};
```

The *Parameter List* consist of a sequence of triples composed of parameter name, parameter configurability and parameter value.

```
typedef string  t_ParameterName;
enum t_ParameterConfigurability {
    FIXED_BY_PROVIDER, CONFIGURABLE_BY_SUBSCRIBER, CUSTOMIZABLE_BY_USER
};
typedef any     t_ParameterValue;
struct t_Parameter {
    t_ParameterName             name;
    t_ParameterConfigurability  configurability;
    t_ParameterValue            value;
};
typedef sequence<t_Parameter> t_ParameterList;
```

The provider may give the subscriber the option to select specific service parameters to apply to all its associated entities[9] -*Subscription Profile*- or to a group of them -*SAG Service Profile*-, reducing the alternatives (*restricts* association in Figure 3-2) given in the service template. These profiles are the main part of the service contract.

```
typedef string t_ServiceProfileId;
struct t_ServiceProfile {
        t_ServiceProfileId                      spId;
        t_ServiceDescription                    serviceDescription;
};
typedef t_ServiceProfile t_SagServiceProfile;
typedef t_ServiceProfile t_SubscriptionProfile;
```

A set of entities, *Users*, *Terminals* or *NAPs*, can be associated to a subscriber. Let's call them *Subscription Assignment Entities* (*SAE*).

```
enum t_entityType {user, terminal, nap};
/**
 * Entity Id identifies uniquely a SAE inside the provider domain.
 **/
union t_entityId switch (t_entityType) {
        case user:     TINACommonTypes::t_UserId             userId;
        case terminal: TINAAccessCommonTypes::t_TerminalId   terminalId;
        case nap:      TINAAccessCommonTypes::t_NAPId        napId;
};
typedef sequence<t_entityId> t_entityIdList;
/**
 * A SAE is characterized by an identifier, a name and a set of properties.
 **/
struct t_Sae {
        t_entityId                          entityId;
        string                              entityName;
        TINACommonTypes::t_PropertyList     properties;
};
```

The subscriber may not want to grant all of them with the same service characteristics (or privileges). For this reason, the subscriber can group them in a set of *Subscription Assignment Group*s (SAG):

```
typedef short    t_SagId;
/**
 * A SAG is characterized by its identifier, a textual description of the
 * group and the list of entities composing it.
 **/
struct t_Sag {
        t_SagId          sagId;
        string           sagDescription;
        t_entityIdList   entityList;
};
```

The subscriber can then assign particular service profiles (*SAG Service Profile*) to each group. The main reason for using this grouping is to ease the subscription process (assignment of profiles to users) in subscriber domains where end-users are naturally classified in categories, organizational or geographical areas, etc., requiring the same service usage privileges. The only restriction to apply is that every SAE must be assigned to one and only one SAG Service Profile for every service.

For every subscriber a default SAG is created with SAGId (0). Every SAE is always assigned to this SAG even if it is assigned to other particular SAG. If a user is removed from any SAG, it will still be associated to this SAG by default. The SAG by default can not be associated to service profiles and users can not be assigned to this SAG explicitly (they are implicitly assigned to it on creation).

9. Users, terminals or network access points.

It is also possible to assign and remove individual SAEs to/from service profiles. This is specially interesting in small subscriber organizations (like residential customers), where the definition of user groups is not strictly needed and does not help the subscriber in the subscription management. Additionally, this provides a lot of flexibility in the service profile assignment, as some users in a group can be discriminated for the access to a specific service, without the need of removing them from the group or defining a new group.

A structure like t_UsagePermit may help in the definition of this restrictions:

```
enum t_UsagePermitFlag {USAGE_ALLOWED, USAGE_DISALLOWED};
struct t_UsagePermit {
        t_entityId                      entityId;
        t_ServiceProfileId              serviceProfileId;
        t_UsagePermitFlag               flag;
};
```

3.6.2 Subscriber Management Interfaces

Sub offers two interfaces for subscriber information access:

- `i_SubscriberInfoQuery`: to query subscriber information, and
- `i_SubscriberInfoMgmt`: to update subscriber information.

Only the first one is prescribed. The second is given for completeness.

3.6.2.1 i_SubscriberInfoQuery

This interface is offered to the access components (NamedUA, PeerA) to allow them to retrieve the subscription information associated to a particular user, the one that the access components represent.

The most important operations in this interface are:

- **`listServices`**() - returns the list of services the user is subscribed to, indicating which ones are available with the current terminal configuration.
- **`getServiceProfiles`**() - returns the profile assigned to the user (SAG service profile) for a specific service or a list of services.
- **`checkServiceProfile`**() - It checks whether the User Service Profile (customized by the end user) is compatible with the subscribed profile (corresponding SAG Service Profile).

3.6.2.2 i_SubscriberInfoMgmt

This interface allows to add, modify or delete subscriber related information like associated entities (users, terminals or NAPs), subscription assignment groups and associations of service profiles to SAGs. From this interface, the client can access only to the information of a particular subscriber.

- **`createSAEs`**() - it creates the entities specified as a parameter returning an identifier for each of them.
- **`deleteSAEs`**() - it deletes the entities specified as a parameter. It removes any existing assignment to SAGs these entities could have.
- **`createSAGs`**() - it creates a (number of) SAG(s). A list of entities for every SAG can be specified. A SAG identifier is returned to ease its further management.
- **`assignSAEs`**() - It assigns a list of entities to a SAG.

- **removeSAEs**() - It removes a list of entities from a SAG.
- **listSAEs**() - It returns the list of entities associated to the subscriber. If a (list of) SAG identifier(s) is specified, it returns only the users assigned to that(those) SAG(s).
- **listSAGs**() - It returns the list of SAGs (ids) for that subscriber.
- **getSubscriberInfo**() - It returns the information about the subscriber.
- **setSubscriberInfo**() - It modifies the information about the subscriber.
- **listSubscribedServices**() - It returns the list of services subscribed by the subscriber. If a user is specified, it returns the list of services granted to that specific user by the subscriber.

3.6.3 Service Contract Management Interface

3.6.3.1 i_ServiceContractInfoMgmt

`i_ServiceContractInfoMgmt` provides functions to define and modify a service contract. It allows the modification and query of the service contract information.

The main operations are:

- **listServiceProfiles**() - It returns the list of identifiers for the service profiles associated with the service contract.
- **getServiceTemplate**() - It returns the template for service profile definition.
- **defineServiceContract**() - It allows to define the service contract. This contract includes, amongst other contractual information, the set of service profiles composing the service contract, namely the subscription profile (applicable to all users) and the set of SAG service profiles (each one applicable to a SAG and consistent with the subscription profile). It returns a list of SAG profile identifiers to ease their future reference. It is used to define and redefine (modify) service contracts.
- **defineServiceProfiles**() - It allows to define a set of service profiles for the service contract, namely the subscription profile and the set of SAG service profiles. It returns a list of SAG profile identifiers to ease their future reference. It is used to define and redefine (modify) service profiles.
- **deleteServiceProfiles**() - It deletes a service profile removing the SAG that could be associated to it.
- **getServiceContractInfo**() - It returns the information related to the service contract. If a list of SAGs is specified, the set of associated SAG service profiles defined for those SAGs in the contract is returned.
- **assignServiceProfile**() - It associates a list of SAEs and SAGs with a SAGServiceProfile[10]. If the service profile is active, the SAEs (the explicitly stated and the ones included in the SAGs) will be able to use the service. In this case, the SAEs' access components will be notified and the SAG service profile will be made available for them.
- **removeServiceProfile**[11]() - It disassociates a list of SAEs and SAGs from a SAGServiceProfile. The specified SAEs (individually specified or inside a SAG) will no longer be able to use the service, until associated with another active service profile.

10. The operations for service profile assignment allow the subscriber to define the access control list for every profile. This list is composed of a number of individual users and user groups.

11. The subscriber can also indicate the exclusion of some SAEs belonging to a SAG from the service profile assignment for that SAG.

- **`activateServiceProfiles`**`()` - It activates a list of SAG service profiles making them available for use. Only SAEs and SAGs assigned to an active Service Profile can make use of the service. These SAEs' access components will be notified and the SAG service profile will be made available for them.
- **`deactivateServiceProfiles`**`()` - It deactivates a list of SAG service profiles. Users (or SAEs) assigned to these service profiles will not be able to use the service.

3.6.4 Subscription Initial Interfaces

These interfaces represent the initial access point for Sub clients.

It provides interfaces to clients to get the appropriate interface references (`i_InitialAccess`) to perform its corresponding subscription management operations.

An interface (`i_Subscribe`) is offered to allow clients to apply for or cancel service contracts or subscriptions to the provider domain.

This set of interfaces includes an interface (`i_ServiceNotify`) to the SLCM to receive notifications about new available services or modification/withdrawal of existing ones.

All these interfaces are external (their clients are not subscription management components). Only the last one is prescribed. The other two are given just to provide a complete solution. The rationale behind this decision is that in a large number of cases the providers will wish to keep on using their legacy subscription systems.

3.6.4.1 i_InitialAccess

It allows a client to request an interface to access to the subscription management functionality. In case the client is an access component, it returns an `i_SubscriberInfoQuery` interface reference and, in case it is a SSM_{ols}, an `i_Subscribe` interface reference. A terminate operation is provided to release the interfaces once they are not needed.

- **`init`**() - returns the list of interface references corresponding to the client that makes the request.
- **`terminate`**() - release the resources that were allocated in the `init` operation.

3.6.4.2 i_Subscribe

It allows to create a subscription contract for a subscriber. These are the main operations in this interface:

- **`getReferences`**() - It returns the references to interfaces to modify subscriber info or service contract info.
- **`listServices`**() - It returns the list of services provided by the provider.
- **`subscribe`**() - It allows to create a subscription contract with the provider. As input parameters it has the subscriber information and a list of services the subscriber is willing to subscribe to. It returns a subscriber identifier, a reference to the subscriber information management interface (`i_SubscriberInfoMgmt`) and a list of interface references for contracting each of the specified services (`i_ServiceContractInfoMgmt`).
- **`unsubscribe`**() - It allows a subscriber to delete a (list of) service contract(s) or the whole relationship with the provider.
- **`contractService`**() - It subscribes a subscriber to a service and returns an interface reference where he can define the service contract (`i_ServiceContractInfoMgmt`).
- **`listSubscribers`**() - It returns the list of subscribers. If a service Id is specified, it returns the list of subscribers for that service. Only accessible for a provider operator.

- **`listServiceContracts`**() - It returns the list of service contracts. If a service Id is specified, it returns the list of service contracts for that service. If a subscriber is specified, it acts as the `listSubscribedServices` in the `i_SubscriberInfoMgmt` interface for that particular subscriber. Only accessible for a provider operator.
- **`listUsers`**() - It returns the list of users for a specified service. Only accessible for a provider operator.

3.6.4.3 i_ServiceNotify

This interface allows the SLCM to notify the Sub about new services deployed and available for subscription and use, or about modification or withdrawal of existing ones.

- **`notify`**() - it notifies Sub about the deployment, upgrade or withdrawal of services in the network, so that Sub can have updated information about subscribable and available services.

3.6.5 Subscription Management Service Interfaces

These are a set of service specific interfaces offered through Ret reference point for the online management of service subscriptions. There are different interfaces for every user type (subscribers or retailer operators).

3.6.5.1 i_SubscriberSubscriptionMgmt

It provides operations for subscribing the retailer, contracting services and defining subscriber and service contract information. It allows the subscription and service contract cancellation and modification and the query of all the subscriber related information.

Operations for applying for service contracts, subscriptions and cancellations:

- **`listServices`**() - It returns the list of services provided by the retailer.
- **`subscribe`**() - It allows to create a subscription contract with the retailer. As input parameters it has the subscriber information and a list of services the subscriber is willing to subscribe to. It returns a subscriber identifier and a list of service contract identifiers. These will be used in the following for making reference to specific service contracts.
- **`unsubscribe`**() - It allows to delete a (list of) service contract(s) or the whole relationship with the retailer.
- **`contractService`**() - It subscribes a subscriber to a service and returns an interface reference where he can define the service contract (`i_ServiceContractInfoMgmt`).
- **`listSubscribedServices`**() - It returns the list of contracted services (just the identifiers) and related service contract identifiers. If a user is specified, it returns the list of services granted to that specific user by the subscriber. The subscriber identifier (account number) is indicated as an input parameter.

Operations for handling subscriber information:

- **`listSAEs`**() - It returns the list of entities associated to the subscriber. If a (list of) SAG identifier(s) is specified, it returns only the users assigned to that(those) SAG(s).
- **`listSAGs`**() - It returns the list of SAGs (ids) for that subscriber.
- **`getSubscriberInfo`**() - It returns the information about the subscriber.
- **`createSAEs`**() - it creates the entities specified as a parameter returning an identifier for each of them.

- **`deleteSAEs`**() - it deletes the entities specified as a parameter. It removes any existing assignment to SAGs these entities could have.
- **`createSAGs`**() - it creates a (number of) SAG(s). A list of entities for every SAG can be specified. A SAG identifier is returned to ease further management.
- **`assignSAEs`**() - It assigns a list of entities to a SAG.
- **`removeSAEs`**() - It removes a list of entities from a SAG.
- **`setSubscriberInfo`**() - It modifies the information about the subscriber.

Operations for defining, modifying and querying service contracts:

- **`getServiceTemplate`**( ) - It returns the template for service profile definition for the specified service.
- **`defineServiceContract`**() - It allows to define the service contract for a specific service. This contract includes, amongst other contractual information, the set of service profiles composing the service contract, namely the subscription profile (applicable to all users) and the set of SAG service profiles (each one applicable to a SAG and consistent with the subscription profile). It returns a list of SAG profile identifiers to ease their future reference. It is used to define and redefine (modify) service contracts.
- **`defineServiceProfiles`**() - It allows to define a set of service profiles for a service contract, namely the subscription profile and the set of SAG service profiles. It returns a list of SAG profile identifiers to ease their future reference. It is used to define and redefine (modify) service profiles.
- **`deleteServiceProfiles`**() - It deletes a service profile.
- **`getServiceContractInfo`**() - It returns the information related to the service contract which identifier is passed as parameter. If a list of SAG profile identifiers is specified, the set of associated SAG service profiles is returned.

Operations for authorization and activation of service profiles:

- **`assignServiceProfile`**() - It associates a list of SAEs and SAGs with a SAGServiceProfile. If the service profile is active, the SAEs (the explicitly stated and the ones included in the SAGs) will be able to use the service. These SAEs' access components will be notified and the SAG service profile will be make available for them. From this profile, the SAE will be able to customize its own user service profile using the service profile management service.
- **`removeServiceProfile`**() - It disassociates a list of SAEs and SAGs from a SAGServiceProfile. The specified SAEs (individually specified or inside a SAG) will no longer be able to use the service, unless associated with another active service profile.
- **`activateServiceProfiles`**( ) - It activates a list of SAG service profiles making them available for use. Only SAEs and SAGs assigned to an active Service Profile can make use of the service.
- **`deactivateServiceProfiles`**( ) - It deactivates a list of SAG service profiles. Users (or SAEs) assigned to these service profiles will not be able to use the service.

3.6.5.2 i_RetailerSubscriptionMgmt

It provides full capabilities for subscribing customers, add, modify, cancel and query service contracts and adding, removing, modifying and querying subscriber information. In general, it provides access to the whole subscription database. It is thus a superset of the previous interface.

Additional operations are:

- **`listSubscribers`**() - It returns the list of subscribers (identifiers). If a service Id is specified, it returns the list of subscribers for that service.
- **`listServiceContracts`**() - It returns the list of service contracts (identifiers). If a service Id is specified, it returns the list of service contracts for that service. If a subscriber is specified, it acts as the `listSubscribedServices` in the `i_SubscriberInfoMgmt` interface for that particular subscriber.
- **`listUsers`**() - It returns the list of users (identifiers) for a specified service.

3.7 ssUAP

Table 3-12. Interfaces, clients and where to find specs

Interface	Client(s)	Event traces	IDL
TINAScsSSUAPIntra:: i_AccessInitialise	TINAObjPA	4.4.1 Start a Service Session and Selection Of session model 4.4.10 Join a Service Session with invitation	Annex 2.17
TINAPartyBasicExtUsage:: i_PartyBasicExt TINAPartyMultipartyUsage:: i_PartyMultipartyExe TINAPartyMultipartyUsage:: i_PartyMultipartyInfo TINAPartyMultipartyUsage:: i_PartyMultipartyInd TINAPartyVotingUsage:: i_PartyVotingInfo TINAPartyControlSRUsage:: i_PartyControlInd TINAPartyControlSRUsage:: i_PartyControlInfo TINAPartyPaSBUsage:: i_PartyPaSBExe TINAPartyPaSBUsage:: i_PartyPaSBInfo TINAPartyPaSBIndUsage:: i_PartyPaSBInd	TINAObjUSM	4.4.2 End a Service Session 4.4.3 End Service Session via Access Session 4.4.5 Suspend a Service Session 4.4.6 Suspend Participation in a Service Session 4.4.9 Invite a User to Join a Session	Ret

Table 3-13. Required interfaces

Server	Interfaces
TINAObjUSM	TINAProviderBasicUsage::i_ProviderBasicReq TINAProviderMultipartyUsage::i_ProviderMultipartyReq TINAProviderVotingUsage::i_ProviderVotingReq TINAProviderControlSRUsage::i_ProviderControlSRReq TINAProviderPaSBUsage::i_ProviderPaSBReq
TINAObjPA	TINAScsPAIntra::i_Access

The ssUAP interacts with the PA and USM. The interactions with the PA mirror those across Ret and allow the ssUAP to request the start, resumption or joining of service sessions. Similarly the PA may invoke operations on i_Init to request that the ssUAP initiate these events.

The interactions between the ssUAP and USM are those specified in Ret usage part.

3.7.1 TINAScsSSUAPIntra::i_AccessInitialise

`i_AccessInitialise` interface is offered to the PA in order that the application can be requested to respond to invitations to service session which the ssUAP supports.

It provides the following operations (which are analogous to `i_ProviderNamedAccess` interface:

- **`startServiceInit`**`()` - allows the PA to initialize the ssUAP to invoke a new service session request for which the ssUAP is compatible.
- **`resumeParticipationInit`**`()` - allows the PA to initialize the ssUAP and instruct it to invoke a resume request for a specific, suspended service session.
- **`joinSessionInit`**`()` - allows the PA to initialize the ssUAP and instruct it to prepare to join a specific, pre-existing service session to which the PA has received an invitation.

3.8 SF

A Service Factory (SF) is a service-specific object, which manages the lifecycle of the service session COs for a service type.

A request to create a service session of a particular service type will result in the creation of one or more object instances[12]. The SF will create and initialize the instances according to rules imposed by their implementation. The SF will return to the client one or more interface references to these components. (The SF is used to create/ manage instances of all the service session related objects defined in this document: USM, SSM, CompUSM and PeerUSM.)

Requests to create a new service session are typically made by UAs, PeerAs. SSM is also client of an SF. It can usually request the SF to create a new USM when a new user is joining an existing session or can request the deletion/suspension of Service Session objects. Another client of the SF is the SLCM, which initializes and manages the SF. The client must have an interface reference to the SF and issue an appropriate request. A SF which supports more than one service type would typically provide separate interfaces for each service type.

A SF supports capabilities to:

12. Typically, the USM and SSM.

- create and initialize objects for one or more service type upon request. (This includes choosing the session models supported by the service session although this may be fixed by the service type.)
- create and initialize an object (typically USM, or CompUSM) to be used in conjunction with other objects (typically a SSM & USMs) created by a different factory instance;
- continue to manage the created objects. It may provide a list of sessions managed by it, and may ‘clean up’ some sessions if requested;
- support suspension/resumption of a service session.

It may optionally include mechanisms to schedule the activation of a session at a specific date and time. (This mechanism includes resource reservation.).

The SF assembles the resources necessary for the existence of a component it creates. Therefore, the SF represents a scope of resource allocation, which is the set of resources available to the SF. A SF may support an interface that enables clients to constrain the scope.

Table 3-14. Interfaces, clients and where to find specs

Interface	Client(s)	Event traces	IDL
TINAScsSF:: i_SSCreate	TINAObjNamedUA TINAObjAnonUA TINAObjPeerA TINAObjSSM	4.4.1 Start a Service Session and Selection Of session model 4.4.10 Join a Service Session with invitation	Annex 2.18
TINAScsSF:: i_SSManage	TINAObjSSM TINAObjNamedUA TINAObjAnonUA	4.4.5 Suspend a Service Session 4.4.6 Suspend Participation in a Service Session 4.4.3 End Service Session via Access Session 4.4.2 End a Service Session	Annex 2.18
TINAScsSF:: i_Resume	TINAObjNamedUA TINAObjAnonUA TINAObjPeerA	4.4.7 Resume a Service Session 4.4.8 Resume Participation in a Service Session	Annex 2.18
TINAScsSF:: i_SSEvents	TINAObjSSM TINAObjMUSM ?		Annex 2.18
TINAScsSF:: i_Init	TINAObjSLCM	4.5.7 Register a new service 4.5.9 Withdraw a service 4.5.8 Modify an existing service	Annex 2.18

Table 3-15. Required interfaces

Server	Interfaces
TINAObjSSM	TINAScsSSMInit::i_Init TINAScsSSMIntra::i_Resume TINAScsSSMIntra::i_AccountingPushMgmt
TINAObjUSM	TINAScsUSMInit::i_Init TINAScsUSMIntra::i_Resume
TINAObjPeerUSM	TINAScsPeerUSMInit::i_Init TINAScsPeerUSMIntra::i_Resume

Table 3-15. Required interfaces

Server	Interfaces
TINAObjCompUSM	TINAScsCompUSMInit::i_Init TINAScsCompUSMIntra::i_Resume

3.8.1 i_SSCreate

`i_SSCreate` interface supports the operations to create Service Session objects (USM, SSM, CompUSM and PeerUsm)

It provides the following operations:

- **`createSSession`**() - allows the AnonUA or NamedUA or PeerA to request the creation of a new service session. This results in the creation of both a USM and SSM (or PeerUSM, CompUSM: ffs).
- **`createUserSSession`**() - allows the SSM to request the addition of a new User to an existing service session. This results in the creation of USM.
- **`createPeerSSession`**() - allows the SSM to request the creation of a PeerUSM to support a federated session.
- **`createCompSSession`**() - allows the SSM to request the creation of a CompUSM to support service composition.

3.8.2 i_SSManage

`i_SSManage` interface supports the operations to manage (end, suspend and get information on) Service Session and related objects (USM, SSM, CompUSM and PeerUsm). A service session is uniquely identified by a GlobalSessionId within a Service Factory. Every USM managed by a SF is uniquely identified by the couple: GlobalSessionId and UsersessionId.

It provides the following operations:

- **`endSSession`**() - allows the SSM to request the deletion of a service session. This results in the deletion of the SSM and all relevant USMs (or PeerUSM and CompUSM). Related resources are released.
- **`endUserSSession`**() - allows the SSM to request the deletion of a Party from an existing service session. This results in the deletion of the relevant USM. Related resources are released.
- **`endPeerSSession`**() - allows the SSM to request the deletion of a PeerUSM. Related resources are released.
- **`endCompSSession`**() - allows the SSM to request the deletion of a CompUSM to support service composition.
- **`listSSession`**() - allows the AnonUA /NamedUA or the SLCM to request for the list of service sessions for this retailer service that match a given property list.
- **`getSsmRef`**() - allows the AnonUA or NamedUA to request for the interface reference of the SSM/USM for a specific session.
- **`suspendSSession`**() - allows the SSM to request the suspension of an existing service session. This results in the creation of a i_Resume interface on the SF, that will be used by UA to resume the service session.

- **`suspendParticipation`**() - allows the SSM to request the suspension of user participation in an existing service session. This might result in the deletion/disabling of interfaces on USM and in the creation of i_Resume interface that can be used by UA to resume participation.

3.8.3 i_Resume

`i_Resume` interface supports the operations to resume a service session and user participation in a service session (USM, SSM, CompUSM and PeerUsm).

It provides the following operations:

- **`resumeSSession`**() - allows the AnonUA or NamedUA to request the resuming of a *suspended* service session. This results in the enabling of interfaces on SSM (identified by the GlobalSessionId) and on the USM (or PeerUSM and CompUSM) identified by userSessionId.
- **`resumeParticipation`**() - allows the AnonUA or NamedUA to request the resuming of a user participation in an *active* service session. This results in the enabling of the USM interfaces of the user identified by userSessionId.

3.8.4 i_Init

This interface allows the SLCM to initialize, configure and manage the SF.

It provides the following operations:

- **`configure`**() - allows the SLCM to configure the SF.
- **`getSessionInfo`**() - allows the SLCM to get Service Session Instances Information.
- **`setSessionInfo`**() - allows the SLCM to set Service Session Instances Information.
- **`deactivate`**() - sets the state of the SF to `deactivating`. Future service session creation requests will be rejected. Once ongoing service sessions are finished, the SF passes to `inactive` state.
- **`activate`**() - sets the state of the SF to `active`. In this state, the SF can be used for service provision.
- **`halt`**() - sets the SF state to `inactive` without a deactivation phase. The SF reports the users using it about the fact and the service sessions are ended immediately.
- **`delete`**() - kills the SF if it is in `inactive` state.

3.8.5 i_SSEvents

`i_SSEvents` interface is used by SSM, USM, PeerUSM and CompUSM to send notification to the SF about changes that occur in the Service Session.

It provides the following operations:

- **`notifySSModification`**() - allows the notification of changes in the Service Session from SS objects (USM, SSM, ...).

3.9 SSM

SSMs support (some or all of) the following capabilities:

- keep track and control the various resources shared by multiple users in a service session. This can be done just by having references to other objects (like a CSM) which really maintain the context of usage for a specific kind of resources;

- hold the state of the service session and support suspension/resumption of the service session;
- support adding/inviting/removing users to/from the service session by interacting with the corresponding UAs;
- support adding/removing/modifying stream bindings and the users' participation in them;
- support the negotiating capabilities among the users interacting with the USMs. SSM will serve as a control center of consensus building (such as voting procedures);
- support management capabilities associated with the service session (e.g., accounting).

Table 3-16. Interfaces, clients and where to find specs

Interface	Client(s)	Event traces	IDL
TINAScsSSMIntra::i_Join	UA (PeerA)	4.4.10 Join a Service Session with invitation	Annex 2.20
TINAScsSSMInit::i_Init TINAScsSSMIntra::i_Resume	SF	4.4.1 Start a Service Session and Selection Of session model 4.4.7 Resume a Service Session 4.4.8 Resume Participation in a Service Session	Annex 2.19 Annex 2.20
TINAScsSSMProviderBasicUsage::i_ProviderGetInterfaces TINAScsSSMProviderBasicUsage::i_ProviderRegisterInterfaces TINAScsSSMProviderBasicUsage::i_ProviderInterfaces TINAScsSSMProviderBasicUsage::i_ProviderBasicReq TINAScsSSMProviderControlSRUsage::i_ProviderControlSRReq TINAScsSSMProviderMultipartyUsage::i_ProviderMultipartyReq TINAScsSSMProviderPaSBUsage::i_ProviderPaSBReq TINAScsSSMProviderVotingUsage::i_ProviderVotingReq	USM (PeerUSM)	Various event traces concerned with usage.	Annex 2.21 Annex 2.22 Annex 2.23 Annex 2.24 Annex 2.25
TINAScsSSMIntra::i_AccountingPush	CSM	4.4.17 Service Session Accounting	Annex 2.20
TINAScsSSMIntra::i_AccountingPushMgmt	SF/USM		Annex 2.20

Table 3-17. Required interfaces

Server	Interfaces
TINAObjnamedUA	TINAScsNamedUAIntra::i_InvitationDelivery TINAScsNamedUAIntra::i_AccountingPush
TINAObjSF	TINAScsSF::i_SSEvent TINAScsSF::i_SSManage TINAScsSF::i_SSCreate

Table 3-17. Required interfaces

Server	Interfaces
TINAObjUSM	TINAScsUSMIntra::i_Resume TINAScsUSMIntra::i_AccountingPush TINAScsUSMPartyBasicExtUsage::i_PartyBasicExtReq TINAScsUSMPartyBasicExtUsage::i_PartyGetInterfaces TINAScsUSMPartyMultipartyIndUsage::i_PartyMultipartyInd TINAScsUSMPartyMultipartyUsage::i_PartyMultipartyExe TINAScsUSMPartyMultipartyUsage::i_PartyMultipartyInfo TINAScsUSMPartyPaSBIndUsage::i_PartyPaSBInd TINAScsUSMPartyPaSBUsage::i_GeneralStreamInfo TINAScsUSMPartyPaSBUsage::i_PartyGeneralStreamInfo TINAScsUSMPartyPaSBUsage::i_PartyPaSBExe TINAScsUSMPartyPaSBUsage::i_PartyPaSBInfo TINAScsUSMPartyVotingUsage::i_PartyVotingInfo TINAScsUSMPartyControlSRUsage::i_PartyControlSRInd TINAScsUSMPartyControlSRUsage::i_PartyControlSRInfo
CSM or CC	i_AccountingPush
TINAObjPeerA	i_InvitationDelivery i_AccountingPush
Other	i_AccountingMgt

3.9.1 i_Join

This interface allows the UA (or PeerA) to forward requests from a consumer to join a session upon invitation or announcement, it also supports replies to invitations.

It provides the following operations:

- **`joinSessionWithInvitation`**( ) - allows a party to join a service session from which an invitation has been issued.
- **`joinSessionWithAnnouncement`**() - allows a party to join an announced service session.
- **`replyToInvitation`**() - allows a party to accept (eventually followed by a join...) or reject an invitation.

3.9.2 i_Init

This interface provides the following operation:

- **`initialise`**( ) - allows the SF to initialize the SSM. Return values are references to the interfaces available to USM(s).
- **`halt`**() - Allows the SF to force a service session to end.
- **`suspend`**() - Allows the SF to force a service session to be suspended.

3.9.3 i_Resume

This interface is created upon suspension of a service session and the interface reference is stored by the SF.

It provides the following operations:

- **`resumeSession`**`()` - allows the SF to resume a service session and the participation of a user.
- **`resumeParticipation`**`()` - allows the SF to resume the participation of a given user in a service session.

3.9.4 i_AccountingPushMgmt

Defined under accounting management. The interface allows management of the accounting event reporting that is pushed to an accounting interface by the SSM as a consequence of session activity. Typical clients are SF and USM.

3.9.5 i_AccountingPush

Defined under accounting management. The interface allows accounting event to be pushed to the SSM from another object, such as CSM, as a consequence of a communication session activity.

3.9.6 Feature set interfaces

The following intradomain interfaces are very similiar to the ones supported by the USM and defined in Ret. They support the same functionalities, by the way some parameters and exceptions have been changed (e.g. `participantSecretID` has been replaced by `partyID`):

```
TINAScsSSMProviderBasicUsage::i_ProviderGetInterfaces,
TINAScsSSMProviderBasicUsage::i_ProviderRegisterInterfaces,
TINAScsSSMProviderBasicUsage::i_ProviderInterfaces,
TINAScsSSMProviderBasicUsage::i_ProviderBasicReq,
TINAScsSSMProviderMultipartyUsage::i_ProviderMultipartyReq,
TINAScsSSMProviderPaSBUsage::i_ProviderPaSBReq,
TINAScsSSMProviderVotingUsage::i_ProviderVotingReq.
```

3.10 USM

Table 3-18. Interfaces, clients and where to find specs

Interface	Client(s)	Event traces	IDL
TINAProviderBasicUsage:: i_providerBasicReq TINAProviderMultipartyUsage:: i_providerMultipartyReq TINAProviderVotingUsage:: i_providerVotingReq TINAProviderControlSRUsage:: i_providerControlReq TINAProviderPaSBUsage:: i_providerPaSBReq	TINAObjssUAP	4.4.2 End a Service Session 4.4.3 End Service Session via Access Session 4.4.5 Suspend a Service Session 4.4.6 Suspend Participation in a Service Session 4.4.9 Invite a User to Join a Session	Ret
TINAScsUSMIntra::i_SessionCtrl TINAScsUSMIntra::i_AccountingPushMgmt	TINAObjUA		Annex 2.27

Table 3-18. Interfaces, clients and where to find specs

Interface	Client(s)	Event traces	IDL
TINAScsUSMInit::i_Init	TINAObjSF	4.4.1 Start a Service Session and Selection Of session model 4.4.7 Resume a Service Session 4.4.8 Resume Participation in a Service Session 4.4.10 Join a Service Session with invitation	Annex 2.26
TINAScsUSMIntra::i_Resume	TINAObjSSM	4.4.7 Resume a Service Session 4.4.8 Resume Participation in a Service Session	Annex 2.27
TINAScsUSMIntra::i_AccountingPush	TINAObjSSM		Annex 2.27
TINAScsUSMPartyBasicExtUsage:: i_PartyBasicExtReq TINAScsUSMPartyBasicExtUsage:: i_PartyGetInterfaces TINAScsUSMPartyMultipartyIndUsage:: i_PartyMultipartyInd TINAScsUSMPartyMultipartyUsage:: i_PartyMultipartyExe TINAScsUSMPartyMultipartyUsage:: i_PartyMultipartyInfo TINAScsUSMPartyPaSBIndUsage:: i_PartyPaSBInd TINAScsUSMPartyPaSBUsage:: i_GeneralStreamInfo TINAScsUSMPartyPaSBUsage:: i_PartyGeneralStreamInfo TINAScsUSMPartyPaS- BUsage::i_PartyPaSBExe TINAScsUSMPartyPaS- BUsage::i_PartyPaSBInfo TINAScsUSMPartyVotin- gUsage::i_PartyVotingInfo TINAScsUSMPartyControlSRUsage:: i_PartyControlSRInd TINAScsUSMPartyControlSRUsage:: i_PartyControlSRInfo	TINAObjSSM	all usage	Annex 2.28 Annex 2.30 Annex 2.31 Annex 2.34 Annex 2.29
TINAScsUSMIntra::i_MgmtCtxt	TINAObjasUAP		Annex 2.27

Table 3-19. Required interfaces

Server	Interfaces
TINAObjSSUAP	TINAPartyBasicExtUsage::i_PartyBasicExt TINAPartyMultipartyUsage::i_PartyMultipartyExe TINAPartyMultipartyUsage::i_PartyMultipartyInfo TINAPartyMultipartyUsage::i_PartyMultipartyInd TINAPartyVotingUsage::i_PartyVotingInfo TINAPartyControlSRUsage::i_PartyControlSRInd TINAPartyControlSRUsage::i_PartyControlSRInfo TINAPartyPaSBUsage::i_PartyPaSBExe TINAPartyPaSBUsage::i_PartyPaSBInfo TINAPartyPaSBUsage:: i_GeneralStreamInfo TINAPartyPaSBUsage:: i_PartyGeneralStreamInfo TINAPartyPaSBIndUsage::i_PartyPaSBInd
TINAObjUA	TINAScsNamedUAIntra::i_AccountingPush TINAScsNamedUAIntra::i_SessionInfo
TINAObjSSM	TINAScsSSMProviderBasicUsage::i_ProviderGetInterfaces TINAScsSSMProviderBasicUsage::i_ProviderRegisterInterfaces TINAScsSSMProviderBasicUsage::i_ProviderInterfaces TINAScsSSMProviderBasicUsage::i_ProviderBasicReq TINAScsSSMProviderControlSRUsage::i_ProviderControlSRReq TINAScsSSMProviderMultipartyReq::i_ProviderMultipartyReq TINAScsSSMProviderPaSBUsage::i_ProviderPaSBReq TINAScsSSMProviderVotingUsage::i_ProviderVotingReq TINAScsSSMIntra::i_AccountingPushMgmt

The USM represents one party in a session and therefore offers to the ssUAP the session control (across Ret) and service specific control (the latter not specified in TINA). The participation of the party in the service session can also be controlled by the access session (across Ret) via the UA or directly by the UA using TINAScsUSMIntra::i_SessionCtrl. The UA also controls the delivery of accounting events sent out by the USM to the UA, by invoking on TINAScsUSMIntra::i_AccountingPushMgmt.

The interactions between the USM and SSM are numerous and mirror those on both sides of Ret, because the USM must invoke session control requests on the SSM and, the SSM must indicate and inform the USM of various changes to the session state. These interfaces are structured in exactly the same way as Ret Feature Sets. In addition, the USM also supports the TINAScsUSMIntra::i_AccountingPush so that the SSM can forward accounting information.

The USM has limited interactions with the SF. When instantiated the USM is initialized with the necessary usage session properties using TINAScsUSMInit::i_Init operations which also return appropriate USM interface references for the interactions with the ssUAP and UA.

3.10.1 TINAScsUSMIntra::i_SessionCtrl

Enables UA to give session control commands that have been requested across an access session (not necessarily the access session that started the service)

- **`endSession`**() - enables the entire session to be ended by the UA (or other permitted object) in the retailer domain. The operation should perform similarly to end Session() invoked by the ssUAP

- **endMyParticipation**() - enables the UA in the retailer domain to terminate the participation of the party represented by the USM.
- **suspendSession**() - enables the UA in the retailer domain to suspend the entire session through the USM of the party represented by the UA.
- **suspendMyParticipation**() - enables the UA in the retailer domain to suspend the participation in the session of the party represented by the UA.

3.10.2 TINAScsUSMIntra::i_AccountingPushMgmt

Defined under accounting management. The interface allows management of the accounting event reporting that is pushed to an accounting interface by the USM as a consequence of session activity.

3.10.3 TINAScsUSMIntra::i_AccountingPush

Defined under accounting management. The interface allows accounting event to be pushed to the USM from another session object, the SSM, a consequence of session activity.

3.10.4 TINAScsUSMInit::i_Init

- **initialise**() - is used by the SF to initialize the service with appropriate properties when the component has been newly instantiated for a new session. The operation is used once and gives access to other interfaces on the USM component.

3.10.5 TINAScsUSMIntra::i_Resume

This interface is given as a fresh reference when the session is suspended or the party represented by the USM is suspended.

- **resumeSession**() - is used by the SSM to inform the USM the session is resuming. The action taken by the USM is specific to the service, but includes informing the namedUA of the party that the session is resumed.
- **resumeMyParticipation**() - is used by the SSM to inform the USM the participant is resuming. The action taken by the USM is specific to the service.

3.10.6 Ret Interfaces

The following interfaces USM-SSM mirror those found in Ret ssUAP-USM and are used by the SSM to inform the USM of changes to the session, either commands or information. They reflect the Ret feature sets. Some of them have been modified to reflect that interactions occur within the same domain:

```
TINAScsUSMPartyMultipartyIndUsage::i_PartyMultipartyInd
TINAScsUSMPartyMultipartyUsage::i_PartyMultipartyExe
TINAScsUSMPartyMultipartyUsage::i_PartyMultipartyInfo
TINAScsUSMPartyPaSBIndUsage::i_PartyPaSBInd
TINAScsUSMPartyPaSBUsage::i_GeneralStreamInfo
TINAScsUSMPartyPaSBUsage::i_PartyGeneralStreamInfo
TINAScsUSMPartyPaSBUsage::i_PartyPaSBExe
TINAScsUSMPartyPaSBUsage::i_PartyPaSBInfo
TINAScsUSMPartyVotingUsage::i_PartyVotingInfo
TINAScsUSMPartyControlSRUsage::i_PartyControlSRInd
TINAScsUSMPartyControlSRUsage::i_PartyControlSRInfo
```

These interfaces are currently empty:

```
TINAScsUSMPartyBasicExtUsage::i_PartyBasicExtReq
TINAScsUSMPartyBasicExtUsage::i_PartyGetInterfaces
```

3.10.7 TINAScsUSMIntra::i_MgmtCtxt

- **`bind`**() - binds a set of management context to the service session in question.
- **`unbind`**() - unbinds a set of management contexts from the service session in question.
- **`rebind`**() - rebinds a set of management contexts to the service session in question.

3.11 SLCM

Editor's note: IDL is currently not available.

The SLCM provides the required functionality to handle the service type and service instance lifecycle. It also allows the configuration of the service network. The SLCM offers the following management functionality (Figure 3-3).

- Service type management: SLCM holds the service descriptions[13] for the relevant service types. Following to Service Deployment, activation/de-activation of service is taken care of. It also offers service status management including suspension/resumption of TINA services, not for service instances. Service version is also managed together with service deployment management.
- Service instance management: SLCM takes care of the deployment (using the deployment management capabilities, explained below), configuration, activation and monitoring of the different service instances. These includes the control and configuration of service factories for each deployed service, the update of traders or locators to make them reachable and the notification to the Sub component, to make the service subscribable. It also performs version control on the deployed service instances (upgrades, modifications, downgrades, etc.). SLCM also controls the deactivation and withdrawal, including the removal of the deployed software, of service instances.
- Service deployment management: this functionality provides with capabilities for downloading service related software and data, and launching, configuring and activating service servers[14]. It will be supported by DPE and node management facilities[15]. This functionality allows the retailer to distribute, over its network composed of a set of service nodes, the service software (SF, USM, SSM and other service components) that will provide a particular service instance. It provides access to DPE Kernel repositories and DPE services databases (for instance, trader and locator) to make the deployed software active and reachable.

13. These descriptions shall be provided by standardization bodies, industrial fora, operators consortia or the retailer itself. From these descriptions, the retailer derives the service templates that describes the service instances (particular implementation of a service type) that it has deployed and provides for use to customers.
14. SFs and other session independent components.
15. These facilities are not specified yet, but they are absolutely needed.

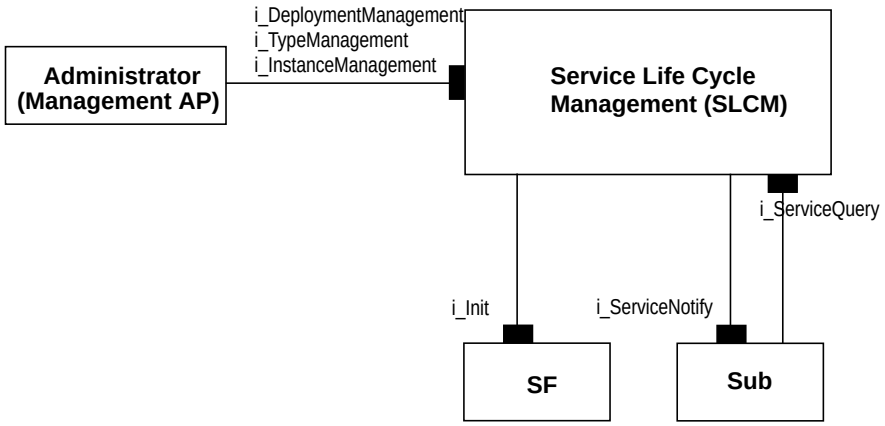


Figure 3-3. Computational Model of Service Life Cycle Management.

It allows the deployment, configuration, activation, deactivation and withdrawal of service instances (interface `i_InstanceMgmt`). It also provides with an interface (`i_TypeMgmt`) to create, modify, query and delete service type descriptions. The clients for these interface are specialized management applications inside the retailer domain.

Table 3-20. Interfaces, clients and where to find specs

Interface	Client(s)	Event traces	IDL
TINAScsServiceLCMgmt:: i_ServiceQuery	TINAObjSub	(not shown)	-
TINAScsServiceLCMgmt:: i_InstanceMgmt	TINAObjSLCM	4.5.7 Register a new service 4.5.8 Modify an existing service 4.5.9 Withdraw a service	-
TINAScsServiceLCMgmt:: i_TypeMgmt	TINAObjSLCM	4.5.7 Register a new service	-
TINAScsServiceLCMgmt:: i_DeploymentMgmt	TINAObjSLCM	4.5.7 Register a new service 4.5.8 Modify an existing service 4.5.9 Withdraw a service	-

Table 3-21. Required interfaces

Server	Interfaces
TINAObjSF	TINAScsSF::i_Init
TINAObjSub	TINAScsSubInitial::i_ServiceNotify

3.11.1 i_ServiceQuery

This interface provides information needed for service subscription to the Sub.

It provides the following operations:

- **`listServices`**() - allows Sub to retrieve list of services that the subscriber will be able to subscribe.
- **`getServiceInfo`**() - allows Sub to retrieve the information regarding the service which could be required to subscription. This information includes the service template.

3.11.2 i_DeploymentMgmt

This interface allows an administrator or a management AP[16] to deploy service elements in the information network. These service elements are service nodes (DPE kernels), DPE services and DPE facilities.

It provides the following operations:

- **`createServiceNode`**() - allows an administrator or a management AP to create and configure a new service node. A reference to the node manager for that node is passed as a parameter. The Node Manager is an abstraction of the computing node, that allows to control and monitor the node resources (CPU, processes, memory and file systems), the node DPE kernel (different repositories and locator) and installed DPE services.
- **`deleteServiceNode`**() - allows an administrator or a management AP to delete a service node.
- **`createServiceDomain`**() - allows an administrator or a management AP to define a service domain. This domain is composed of a set of service nodes or service domains and provides a specific service instance to a number of accessing users. The identity of the users (UAs) making use of this service domain and of each component service node is not determined on deployment time. The UAs are assigned to the service domain dynamically following the particular operator service provision policy (for instance, KTN access point proximity). The DPE platform will be configured in such a way that the service instance is usable and reachable from the required users, terminals or KTN access points served by the new service domain.
- **`addToServiceDomain`**() - allows an administrator or a management AP to assign a service node (or service domain) to a service domain. This implies that the service node will be provided with the required configuration for the provision of a specific service when the corresponding service instance is deployed, or immediately if the service instance is already deployed.
- **`dropFromServiceDomain`**() - allows an administrator or a management AP to remove a service node (or service domain) from a specific service domain. All software and configuration or usage data related to the specific service the domain provides and residing on the service node is withdrawn.
- **`deleteServiceDomain`**() - allows an administrator or a management AP to remove a service domain. All software and configuration or usage data related to the specific service the domain provided and residing on the service nodes composing the domain is withdrawn.

3.11.3 i_InstanceMgmt

This interface allows an administrator or a management AP to control and manage the service itself taking into account of service version, access right and so on.

16. For management AP we mean an offline application that accesses management system (but not through Ret.)

It provides the following operations:

- **`createServiceInstance`**() - allows an administrator or a management AP to create a service instance. If service domains have been defined for that service instance, the deployment will be initiated on those defined service domains. When this operation ends, the service is deployed but inactive (not available for subscription and use). That implies that the location and trader services are not updated with the required information to make the service factories reachable and the subscription component (Sub) does not know about this new service instance.
- **`modifyServiceTemplate`**() - allows an administrator or a management AP to modify the service template for a service instance.
- **`modifyServiceConfiguration`**() - allows an administrator or a management AP to modify the service configuration for a service instance.
- **`activateServiceInstance`**() - allows an administrator or a management AP to activate a service instance. Once executed, the service is available for subscription and use in the defined service domains. That implies that the location and trader services are updated with the required information to make the service factories reachable and the subscription component (Sub) is notified about this new subscribable service instance.
- **`deactivateServiceInstance`**() - allows an administrator or a management AP to deactivate a service instance. Once executed, the service is not available for use.
- **`deleteServiceInstance`**() - allows an administrator or a management AP to remove a service instance. Service contracts for that service instance needs to be cancelled or modified being the contracted service (instance) replaced by a new one.
- **`getState`**() - allows an administrator or a management AP to check the state of a service instance.

3.11.4 i_TypeMgmt

This interface allows an administrator or a management AP to manage the service types lifecycle.

It provides the following operations:

- **`defineServiceType`**() - allows an administrator or a management AP to define a new service type.
- **`modifyServiceType`**() - allows an administrator or a management AP to modify an already existing service type.
- **`deleteServiceType`**() - allows an administrator or a management AP to eliminate an already existing service type.
- **`listServiceTypes`**() - allows an administrator or a management AP to retrieve the list of service types in that retailer domain.
- **`getServiceTypeInfo`**() - allows an administrator or a management AP to retrieve information about a particular service type, namely the service description.

3.12 AmcLadder

Object AmcLadder is a generic object for accounting management, from which other session 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[17].

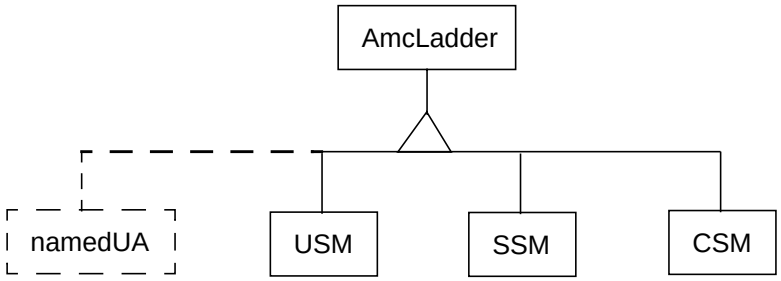


Figure 3-4. Object Inheritance of AmcLadder to Service Components

Figure 3-4 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.

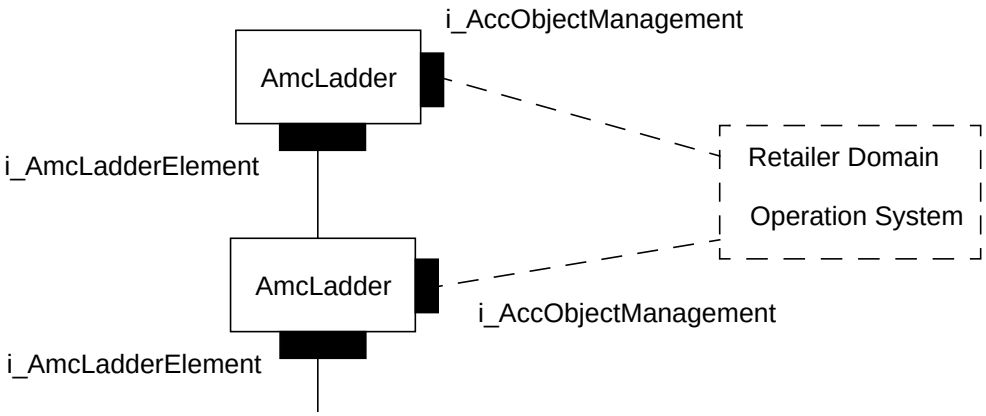


Figure 3-5. Formation of Accounting Management Ladder

Figure 3-5 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,

17. The namedUA could be derived from this generic object or have a special behavior and definition. In current specifications, the namedUA is storing persistently (till its records are processed by a BIlling Center application) the accounting records for the corresponding user, and thus is not considered a pure AmcLadder element.

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 3-22. Supported Interfaces of AmcLadder

Interface	Client(s)	Event traces	IDL
TINAScsAmcObject:: i_AmcLadderElement	TINAObjAmcLadder	Scenario in Accounting Management (sect 5)	Annex 2.4
TINAScsAmcObject:: i_AccObjectManagement	Operation System [AmcLadder] [SF]		Annex 2.4

Table 3-23. Required Interfaces of AmcLadder

Server	Interfaces
TINAObjAmcLadder	TINAScsAmcObject::i_AmcLadderElement

3.12.1 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.

3.12.2 i_AccObjectManagement

This interface provides the following operations:

- **`start`**`()` - starts accounting activities of the object afresh.
- **`stop`**`()` - stops accounting activities of the object.
- **`suspend`**`()` - suspends accounting activities of the object.
- **`resume`**`()` - resumes accounting activities of the suspended object.
- **`set_state`**`()` - sets the state of the object. This operation is used when recovering from a crash or system failure, the state of the object has to be restored from a log record.
- **`set_accounting_cycle`**`()` - sets accounting cycle, when the object reports its accounting events periodically to the upper elements in the ladder.
- **`suspend_notification`**`()` - suspends notification of events to the upper elements. Other accounting activities, however, can continue.
- **`resume_notification`**`()` - resumes notification of events, which is being suspended.
- **`flush_notification`**`()` - flushes events in the notification queue. This operation may be used to flush all the events to the upper elements, before the object is to be deleted, at the conclusion of a service transaction.
- **`set_verbosity_level`**`()` - sets verbosity of accounting events. This operation may be used, in particular, when accounting events go to a log manager.

- **`set_Notification_Destination`**( ) - adds an *i_AmcLadderElement* to its list of notification destination. Note that it is possible that a ladder element may send its events to multiple destinations, e. g. an SSM may send its events to multiple USMs.
- **`reset_Notification_Destination`**( ) - removes an interface from destination list.
- **`reset_all_Notification_Destination`**( ) - resets all the notification destination, and the destination list becomes empty.

3.13 Federation and Composition related components

The components below have not been specified in detail for this version since they rely on the definition of the Retailer-to-Retailer and Third-Party Provider reference points and those reference points are currently unspecified. This section provides some proposals for what the components are expected to contain upon specification.

3.13.1 PeerA - Peer Agent

The PeerA should support the combined external capabilities of a namedUA and a PA. In addition to these capabilities, the PeerA must support inter domain management functionality e.g., transfer of accounting info.

It is possible that the PeerA needs to support mechanisms for exchanging Session Graph information between service session in different domains, since the level of trust between two retailers might affect the level of detail made visible to other domains, yet this level of trust is clearly service independent.

3.13.2 PeerUSM - Peer Usage Session Manager

The PeerUSM allows a service session to interact with another service session in another domain. It differs from the USM in a number of ways, most notably it most support all feature set interactions in both directions, whereas the USM only supports Req's from the PA and Ind's, Info's, and Exe's from the SSM. This also means that the SSM must offer the corresponding interfaces to the PeerUSM, thus the SSM is not yet complete.

3.14 Yet-To-Be-Defined Service Components

3.14.1 Security Manager

The functionality of Security Manager is yet to be defined fully, however it is planned to provide the following security functionalities.

- *Authentication:* authentication server is necessary, which will provide authentication information of the users of Ret reference point interfaces. The authentication information of the user may be provided by a third-party authentication service, which would provide driver's license number, social security number, bank account number, etc., which would provide sufficient level of trust of the user. Although TINA user is living in a cyberspace, those information regarding the user's physical identity is still necessary to obligate the user's financial responsibilities. Along with these physical authentication information, normal security measures for authentication such as ID card, authentication service based on symmetric/asymmetric cryptography will be provided to grant access to TINA services. These cryptographic authentication mechanisms will be provided by DPE security services.
- *Authorization:* once the user is authenticated, he is given authorization to access services (interfaces) given his record from subscription management. In short, s/he is given authorization to services s/he has subscribed to. In this way, security manager has an essential relationship with subscription management regarding authorization.

- *Role Management:* the security management is a complex issue, yet it is still relatively easy when the security management is closed within a single retailer domain. The federation and composition concept, though they are still experimental at this point, can create a very complex security issue. In particular trust management is a difficult issue, as it can be quite involved to know who represents whom and who trusts whom by how much in a complex delegation chain. Ideas have proposed (TINA'97, "Dynamic Role Creation from Role Class Hierarchy" by Takeo Hamada) to manage the complexity of this issue by introducing role based management in TINA service session. Roles are introduced as a consequence of subscription management, from which the user is entitled to have a role within a service session. More details are yet to be defined in the subsequent version of SCS specifications.

4. Dynamic Behavior

This chapter defines a set of interactions between TINA service components. Dynamic interactions are part of the computational viewpoint, and are complementary to the “static” component definitions presented in Chapter 3. These interactions must be supported by TINA components in order for them to interoperate[1]. Interactions are defined in a set of scenarios. Scenarios are based upon a user wishing to perform an action. Scenarios are categorized according to the separation of access and usage.

4.1 Scenario Groupings

The following is a list of scenarios, which identify interactions between TINA components. Only scenarios for which the interactions have been defined have been included. Scenarios are listed below as access and usage related as defined in Service Architecture 5.0. Usage scenarios are further categorized by primary usage and ancillary usage.

Access related scenarios are associated with the establishment and ending of an access session.

Usage related scenarios are associated with the use of service sessions. The primary usage scenarios are based upon the use of generic service session control operations. These operations are applicable to both primary usage and ancillary usage service sessions, but are given in this section for clarity. The ancillary usage scenarios are based on actual ancillary services that a provider may wish to support, e.g. subscription.

Access related scenarios:

1. Contact a provider (Section 4.3.1)
2. Login to a Provider as a Known User (Section 4.3.2)
3. Login to Provider as Anonymous User (Section 4.3.3)
4. Logout from a Provider (Section 4.3.4)
5. Check Accounting Information (Section 4.3.5)
6. List Subscribed Services (Section 4.3.6)

Usage related scenarios:

The following are **primary usage** related scenarios:

1. Start a Service Session and Selection Of session model (Section 4.4.1)
2. End a Service Session (Section 4.4.2)
3. End Service Session via Access Session (Section 4.4.3)
4. End Participation in a Service Session (Section 4.4.4)
5. Suspend a Service Session (Section 4.4.5)
6. Suspend Participation in a Service Session (Section 4.4.6)

1. Conforming to the scenarios defined here is not sufficient to ensure interoperability.

The following are **ancillary usage** related scenarios:

4.2 Scenario Descriptions

Each scenario defines a precondition; a number of sequential steps that take place in the defined order, and a post-condition. The scenario steps are also shown diagrammatically in event traces.

The precondition defines a state that must have been fulfilled before the steps of the scenario can occur. (Usually other scenarios that must have been performed before this scenario can take place.) The post-condition defines the expected state that the various components in the scenario would be in after the step have been performed.

Some scenarios are split into several cases. The cases cover scenarios that can be initiated in a number of ways, or detail options which depend upon the situation.

Event trace diagrams trace the steps defined in each scenario. Comment boxes denote actions performed by the object. (These actions may be entirely internal, or involve interactions with objects which are not prescribed by TINA). Return messages are denoted by [ ], and those which return a value as [return value].

Abbreviations for components are the same as in Section 2. In addition, IR denotes interface reference, and an interface type is in `courier` font.

The following are assumptions which apply to many of the scenarios:

- Components in the user's domain can gain interface references to other components in the user's domain. How references are retrieved is not specified;
- UAPs are categorized into access session related UAPs, and service session related UAPs. When a scenario refers to an access session related UAP, the UAP must provide access session related capabilities. When a scenario refers to a service session related UAP, the UAP must provide service session related capabilities;
- The service session related UAPs may have to be a service specific UAP i.e., a UAP which is specialized to the service type of session;
- The naming service is a DPE service. It provides interface references to provider domain objects, (e.g. initial agents), when given a provider's name and/or user's name. It is directly accessible by all components on the DPE. How the naming service gains interface references is undefined. This may require the naming service to contact the provider in order to gain a reference to the required object;
- The location service is a DPE service. It provides interface references to service factories. (Other location services are also possible.) How the location service gains interface references is undefined. This may require the location service to contact the provider of the factory in order to gain a reference to the required object;
- Both naming service and location service can be replaced with trading service, depending on how interface references of objects are exported to the service. Naming, location and trading service use within the scenarios is for further study;
- Some scenarios indicate that interface references to components become invalid. This means that components which have interface references to these components should not use them, and that the DPE may ensure that such references cannot be used to interact with the components.

4.3 Access related scenarios

4.3.1 Contact a provider

This example shows the user making contact with a provider. This scenario supports user mobility by allowing the user to contact a specific provider from any terminal.

A PA must be present in the user's domain.

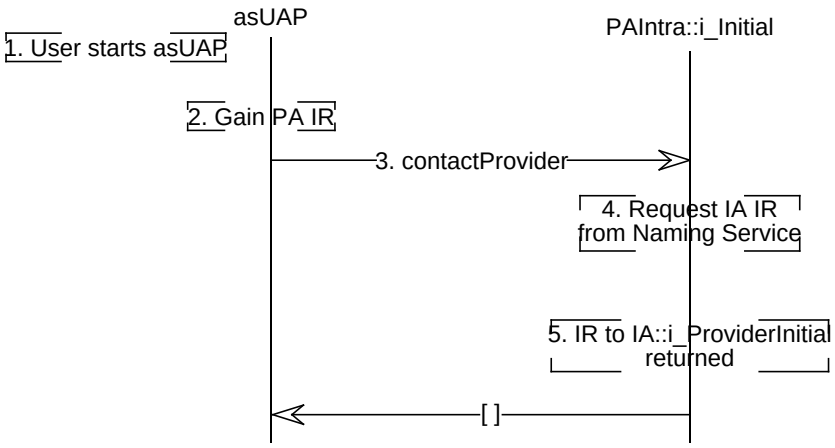


Figure 4-1. Contact a Provider

1. User starts an access session related UAP. He supplies the provider name he wishes to contact. The UAP gains a reference to the PA i_Initial interface.
2. UAP -> PA::i_Initial::contactProvider()
   UAP requests the PA to contact the provider, giving the provider name.
3. PA gains a reference to an `i_ProviderInitial` interface of an initial agent of the provider[2] using a location or naming service.

   PA returns success to UAP.

The PA has an interface reference to the IA. The user has not setup an access session between the PA and IA. The IA does not support invocation of services, only operations to set up an access session as a known user or an anonymous user.

The IA has no knowledge of any interfaces on the PA or in the user's domain.

It is possible for the provider to download a provider specific PA to the user's domain, once an interface reference to the IA has been gained by the PA. This helps to support user mobility. No scenario describing how this is achieved is defined at present.

4.3.2 Login to a Provider as a Known User

This scenario shows how a user can establish an access session with his named UA in the domain of a certain provider, so he can make use of this provider's services to which he has previously subscribed.

It is assumed that the user has already had an initial contact with this provider by means of which the user's PA obtained a reference to the `i_ProviderInitial` interface of an initial agent of the provider.

2. The PA may use a location service to find an interface reference for the IA, or some other means.

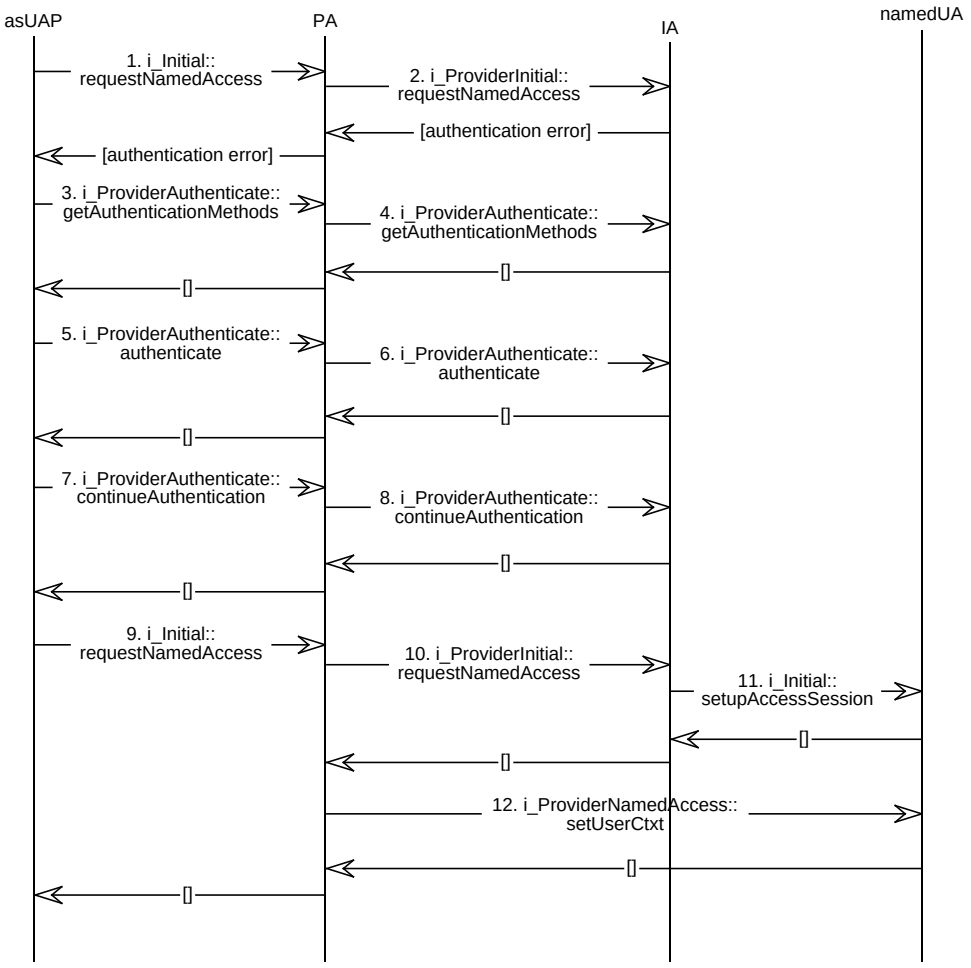


Figure 4-2. Logon as a known user

1. asUAP -> PA::i_Initial::requestNamedAccess()
   The user uses an access session related UAP to forward to the PA the request to log in to the provider as a known user. The access session UAP may request the user's user name and other security related information, like for instance a password, to forward to the PA in the request.

2. PA -> IA::i_ProviderInitial::requestNamedAccess()
   The PA invokes `requestNamedAccess()` on the provider's IA to establish an access session that allows the user to access the services he is subscribed to.

   At this stage, a secure context between user and provider may have already been established through DPE security services. If this is the case, and no further authentication of the user and provider is necessary, then this operation returns a reference to the `i_ProviderNamedAccess` interface, and the scenario continues at step 12.

   If a secure context between user and provider has not been established yet, the provider will not allow an access session to be set-up, so the IA returns an exception `e_AuthenticationError` which contains a reference to an `i_ProviderAuthenticate`

interface. This interface can be used to authenticate the user and provider domains, and establish a secure context.

The invocation on the PA returns an exception as well to the asUAP.

3. asUAP -> PA::i_ProviderAuthenticate::getAuthenticationMethods()
The `i_ProviderAuthenticate` interface provides a generic set of operations that can be used to 'transport' authentication information between the authenticating domains (user and provider). In order to use these methods for authentication, both domains must know the authentication method they will use to generate the authentication data 'transported' in the `authenticate`() and `continueAuthentication`() operations.

   The asUAP request the PA to find out the authentication methods the provider supports

4. PA -> IA::i_ProviderAuthenticate::getAuthenticationMethods()
The PA request a list of the authentication methods the provider supports. The provider should return one or more authentication methods that he wishes the user to use to authenticate the domains. (Only one authenticate method will be used to authenticate).

   The list of authentication methods is forwarded to the asUAP.

5. asUAP -> PA::i_ProviderAuthenticate::authenticate()
The asUAP uses the `authenticate`() operation to select an authentication method and pass authentication data to the PA. The operation also allows the request for specific privileges for the user, that are used as part of the secure context.

6. PA -> IA::i_ProviderAuthenticate::authenticate()
The PA passes the selected authentication method and authentication data to the provider, and the request for specific privileges for the user.

   The provider processes the authentication data. If the authentication method has successfully authenticated the domains, then the operation returns with the authentication status as `SecAuthSuccess`, and the privileges allowed by the user. An access session has been established between the domains, and the scenario continues with step 10. (In order to establish the secure context for DPE interactions, some of the returned privileges, and authentication specific data may need to be passed to the DPE. This is DPE specific and not defined by TINA).

   If the authentication status is SecAuthContinue, the authentication must continue, using the `continueAuthentication`() operation. The provider returns some more authentication (challenge) data from `authenticate`(). It is forwarded to the asUAP.

7. asUAP -> PA::i_ProviderAuthenticate::continueAuthentication()
The asUAP responds to the challenge data, (processing it according to the authentication method, and passing the results to the PA).

8. PA -> IA::i_ProviderAuthenticate::continueAuthentication()
The PA forwards the results to the provider..

   Again if the authentication has been successful, the provider returns the privileges granted to the user, and possibly some authentication specific data. (As in step 5., in order to establish the secure context for DPE interactions, some of the returned privileges, and authentication specific data may need to be passed to the DPE. This is DPE specific and not defined by TINA).

   If the authentication status is SecAuthContinue, then the PA must continue to process the challenge data, returned by this operation, and invoke `continueAuthentication`() with the new results. This loop continues until the authentication status is successful, or a `SecAuthFailure`, or a `SecAuthExpired` is returned. The former means the user has failed

to authenticate, (because he has been sending the wrong responses to the challenge data). The latter means that responses must be generated within a certain timeframe (according to the authentication method), and the PA did not respond quickly enough. They may attempt to authenticate again, restarting the authentication process by calling authenticate().

9. asUAP -> PA::i_Initial::requestNamedAccess()
   Now that a secure context between user and provider has been established, the asUAP again requests that the access session is established. This operation returns the PA::i_Access Interface Reference to the asUAP.

10. PA -> IA::i_ProviderInitial::requestNamedAccess()
    The PA forwards the request to the provider.

11. IA -> UA::i_Initial::setupAccessSession()
    The provider's IA contacts the namedUA of the user and requests the setup of an access session with the authenticated user. The namedUA generates the necessary access session identifiers and returns them together with a reference to the `i_ProviderNamedAccess` interface; this interface will be used for interactions during the access session; the access session identifier will be used in all requests on this interface.

    The access session ids and interface reference are passed to the PA.

12. PA -> UA::i_ProviderNamedAccess::setUserCtxt()
    The PA starts access session interactions invoking the `setUserCtxt()` operation on `i_ProviderNamedAccess`. This gives the user's UA some information about the user's domain, such as interface references, available user applications, or the operating system used.

    The user's UA acknowledges the receipt of this user's domain information. The PA should have this acknowledgment before considering the establishment of the access session complete because some operations may rise an exception if the UA does not have this information.

    The PA informs the asUAP of the successful establishment of the access session. The asUAP in turn will inform the user who can now start using the services he is subscribed to.

The asUAP can now start to register its interfaces to the PA::i_Access.

4.3.3 Login to Provider as Anonymous User

This scenario allows the user to establish an access session with an anonymous user agent of the provider. It is used when the user wishes to make use of the provider's services, but does not wish to disclose their identity, or form a contract with the provider that lasts longer than a single access session. This scenario supports user mobility, by allowing the user to establish an access session with a provider from any terminal.

It is assumed that the user has already had an initial contact with this provider by means of which the user's PA obtained a reference to the i_ProviderInitial interface of an initial agent of the provider.

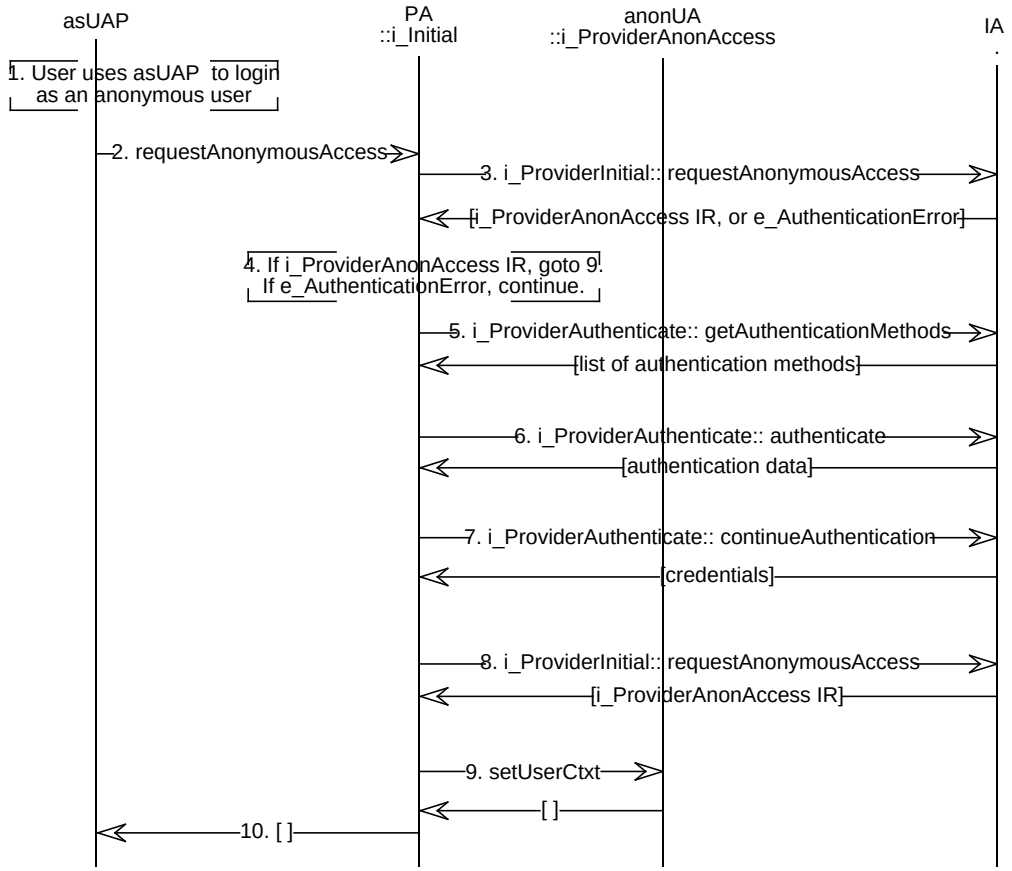


Figure 4-3. Login as an anonymous user

1. User uses an access session related UAP to login to the provider, as an anonymous user.

2. asUAP -> PA::i_Initial::requestAnonymousAccess()
   The user uses an access session related UAP to forward to the PA the request to log in to the provider as an anonymous user.

3. PA -> IA::i_ProviderInitial::requestAnonymousAccess()
   The PA invokes requestAnonymousAccess() on the provider's IA to establish an access session that allows the user to access the provider's services.

4. At this stage, a secure context between user and provider may have already been established through DPE security services. If this is the case, then this operation returns a reference to the i_ProviderAnonymousAccess interface, and the scenario continues at step 12.

   If a secure context between user and provider has not been established yet, the provider may not allow an access session to be established, so the IA returns an exception e_AuthenticationError which contains a reference to an i_ProviderAuthenticate interface. This interface 'can' be used to authenticate the user and provider domains, and establish a secure context. (An anonymous user will not wish to identify themselves to the provider, but they may wish to authenticate the provider, ensuring they have established an access session with the 'actual' provider, and not another spoofing their identity. The i_ProviderAuthenticate interface can be used to authenticate the provider, given that an appropriate authentication method is used).

5. PA -> IA::i_ProviderAuthenticate::getAuthenticationMethods()
   The i_ProviderAuthenticate interface provides a generic set of operations that can be used to 'transport' authentication information between the authenticating domains (user and provider). In order to use these methods for authentication, both domains must know the authentication method they will use to generate the authentication data 'transported' in the authenticate() and continueAuthentication() operations.

   The PA request a list of the authentication methods the provider supports. The Provider should return one or more authentication methods that allows the provider to authenticate itself to the user, without the user providing any authentication information. (Only one authenticate method will be used to authenticate).

6. PA -> IA::i_ProviderAuthenticate::authenticate()
   The PA uses to the authenticate() operation to select an authentication method. (They will also pass some authentication data generated according to authentication protocol, by the user domain). The operation also allows the PA to request specific privileges for the user, that are used as part of the secure context[3].

   If the authentication method is successful, the operation returns with the authentication status as SecAuthSuccess, and the privileges allowed by the user. An access session has been established between the domains, and the scenario continues with step 10. (In order to establish the secure context for DPE interactions, some of the returned privileges, and authentication specific data may need to be passed to the DPE. This is DPE specific and not defined by TINA).

   If the authentication status is SecAuthContinue, the authentication must continue, using the continueAuthentication() operation. The provider returns some more authentication (challenge) data from authenticate().

7. PA -> IA::i_ProviderAuthenticate::continueAuthentication()
   The PA responds to the challenge data, (processing it according to the authentication method, and passing the results to the provider through continueAuthentication()).

   Again if the authentication has been successful, the provider returns the privileges granted to the user, and possibly some authentication specific data. (As in step 5., in order to establish the secure context for DPE interactions, some of the returned privileges, and authentication specific data may need to be passed to the DPE. This is DPE specific and not defined by TINA).

   If the authentication status is SecAuthContinue, then the PA must continue to process the challenge data, returned by this operation, and invoke continueAuthentication() with the new results. This loop continues until the authentication status is successful, or a SecAuthFailure, or a SecAuthExpired is returned. The former means the user has failed to authenticate, (because they have been sending the wrong responses to the challenge data). The latter means that responses must be generated within a certain timeframe (according to the authentication method), and the PA did not respond quickly enough. They may attempt to authenticate again, restarting the authentication process by calling authenticate().

8. PA -> IA::i_ProviderInitial::requestAnonymousAccess()
   Now that a secure context between user and provider has been established, the PA again requests that the access session is established.

3. While the user may not be authenticated for anonymous usage of the provider, there are third-party authentication schemes that allow a user's true identity to be kept from the provider. The user may be known to the provider by a temporary alias. In either circumstance there is still a need to agree the secure context between user and provider.

If the provider accepts this request, then a reference to the i_ProviderAnonAccess interface of the anonUA[4] is returned. This interface will be used for interactions during the access session. Included in the return is an access session identifier that will be used in all requests on that interface.

9. PA -> anonUA::i_ProviderAnonAccess::setUserCtxt()
   The PA starts access session interactions invoking the setUserCtxt() operation on i_ProviderAnonAccess. This gives the user's UA some information about the user's domain, such as interface references, available user applications, or the operating system used.

   The user's UA acknowledges the receipt of this user's domain information. The PA should have this acknowledgment before considering the establishment of the access session complete because some operations may rise an exception if the UA does not have this information.

10. The PA informs the asUAP of the successful establishment of the access session and returns the PA::i_Access interface reference, to be used by the asUAP for further request during the access sesion. The asUAP in turn will inform the user who can now start 'subscribing' to services that they will use during the access session.

The asUAP can now register its interfaces to the PA::i_Access.

4.3.4 Logout from a Provider

This event trace shows how a user can logout from a provider. Logging out means ending all the access sessions the user has with this provider. In the example followed here the user has only one access session with the provider, and no service sessions.

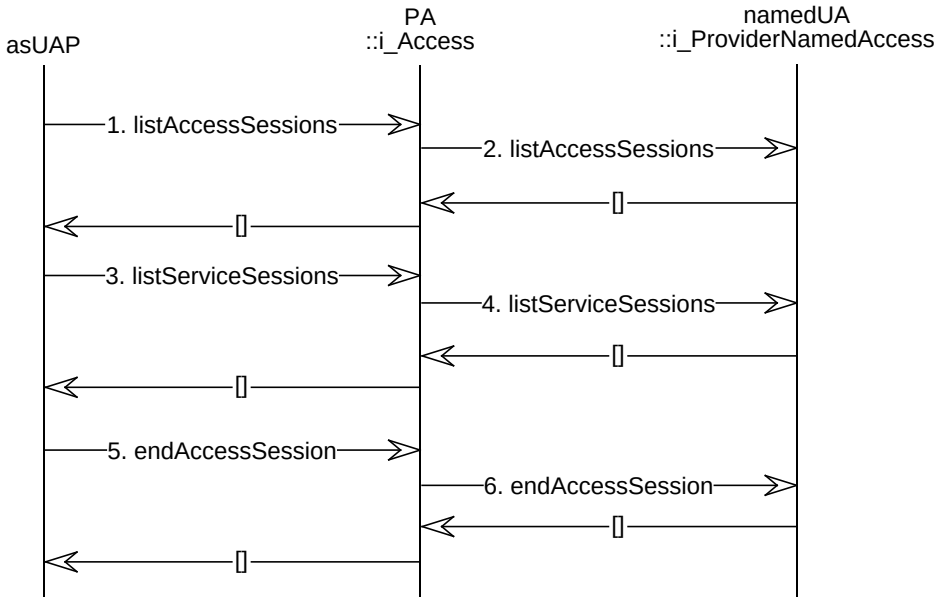


Figure 4-4. Logout from a provider

4. The anonymous User Agent (anonUA) has no knowledge of the users identity, nor personal information. Besides usually an anonymousUA instance can support more than one anonymous user.

1. asUAP -> PA::i_Access::listAccessSessions()
   The user uses an access session related UAP to logout from the provider, starting with checking whether or not (s)he has other access sessions going on with the same provider.

2. PA -> UA::i_ProviderNamedAccess::listAccessSessions()
   The PA forwards this request to the user's UA, which stores updated information about the access sessions the user is involved in.

   The UA returns the list of access sessions the user has with the provider; in this case the list will only contain one access session, which is the one within which these interactions are taking place.

   The PA forwards the list to the asUAP which in turn forwards it to the user.

3. asUAP -> PA::i_Access::listServiceSessions()
   The user does not want to leave the access session before finishing any service session (s)he may have with the provider, so (s)he uses the asUAP to request a list of service sessions, scoped to contain only service sessions in the current access session.

4. PA -> UA::i_ProviderNamedAccess::listServiceSessions()
   The PA forwards this request to the user's UA, which stores updated information about the service sessions the user is involved in.

   The UA returns the list of service sessions the user has in this access session; in this case the list is empty.

   The PA forwards the empty list to the asUAP which in turn forwards it to the user.

   If the list were not empty the user would start the process of finishing the service sessions, or decide against ending the access session.

5. asUAP -> PA::i_Access::endAccessSession()
   The user requests through the asUAP the end of the current access session. In case (s)he had not made sure there no service sessions running in this access session, (s)he could use this request to specify some actions to be taken if there are. It would also be possible to request ending all access session with the provider in this invocation if there were any more.

6. PA -> UA::i_ProviderNamedAccess::endAccessSession()
   The PA forwards the request to end the access session; instructions to follow if any service sessions were active in the access session would be forwarded too.

   The UA considers the access session finished (deletes associated interfaces and user context) and returns an acknowledgment to the PA.

   The PA deletes the interfaces associated to the access session and forwards the acknowledgment to the UAP, which in turn informs the user of the completion of the logout process.

4.3.5 Check Accounting Information

This event trace shows how User (asUAP) can check its accounting information using namedUA. As preconditions of the scenario, namedUA has billing records stored by previous and on-going service sessions (see Section 4.4.17), which are made accessible from asUAP or PA, through the interface (`namedUA::i_AccountingPull`). As postconditions, retrieved billing information can be consumed at asUAP and PA, or cached into PA, in which case later references to the billing information. can be obtained from PA, which resides in the user domain, not necessarily through namedUA across Ret reference point. The matter, however, is an implementation option.

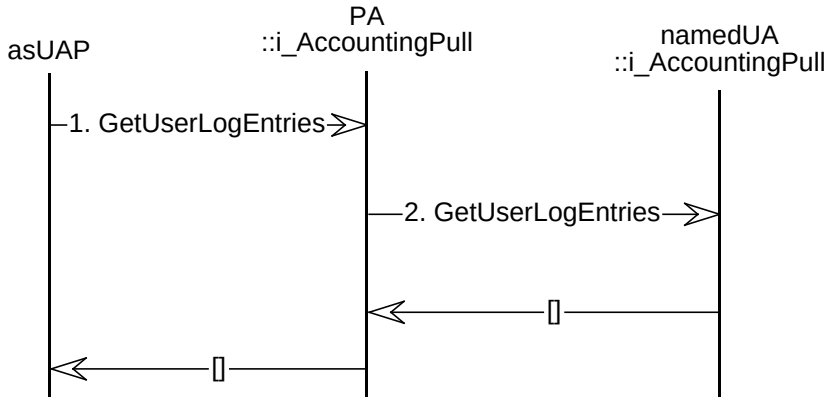


Figure 4-5. Check Accounting Information

1. asUAP -> PA::i_AccountingPull::GetUserLogEntries()
   `GetUserLogEntries` request is sent from asUAP to PA. A list of billing records is returned to asUAP as its reply. asUAPs (Access Session UAP) are capable of querying billing status of the user (pull-accounting events).
2. PA -> namedUA::i_AccountingPull::GetUserLogEntries()
   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. `GetUserLogEntries` request is sent from PA to named UA. A list of billing records is returned to PA as its reply.

4.3.6 List Subscribed Services

This scenario allows the user to request a list of their subscribed services. These are services to which the user has subscribed to, or been subscribed to by a third party./

The scenario is applicable to both named and anonymous users. The interactions between the user domain and the provider domain are identical for both types of user. For this reason, the User Agent (UA) component and i_ProviderAccess interfaces are used in the scenario and event traces. These are abstract components/interfaces, and so direct instances of these types are created in the provider domain. However, sub-types of these for named users, (namedUA, i_ProviderNamedAccess), and anonymous users (anonUA, i_ProviderAnonAccess) are created by the provider domain.

This scenario does not describe how the user subscribes to, or becomes subscribed to these services. This is covered by traces in Section 4.5.1 and Section 4.5.3.

It is assumed that the user has established an access session with this provider, and the user's PA has a reference to the i_ProviderAccess interface of a user agent of the provider.

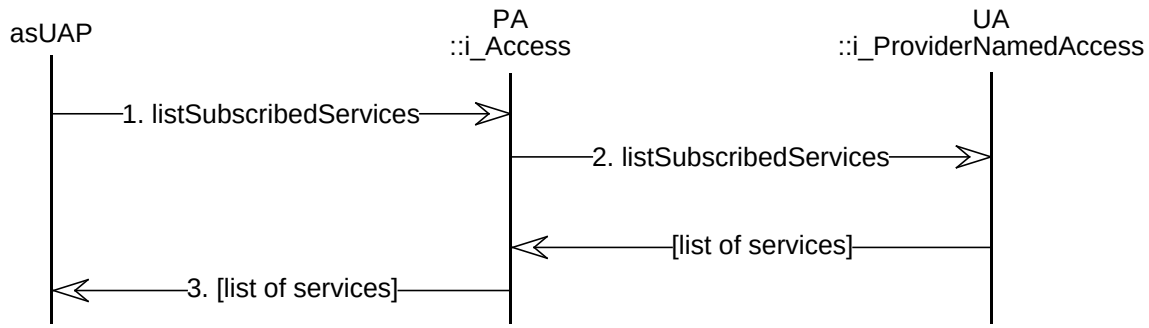


Figure 4-6. List subscribed services

1. asUAP -> PA::i_Access::listSubscribedServices()
   The user uses the as-UAP to obtain a list of their subscribed services. The as-UAP invokes this operation on the PA.

   The PA may have previously retrieved a list of the user's subscribed services, and so may not need to invoke the next operation. (The PA is informed of new subscribed services through the newSubscribedServicesInfo() operation on the i_UserAccessSessionInfo interface.)

   This scenario assumes the PA does not have up-to-date information on the subscribed services, and continues to the next step.

2. PA -> UA::i_ProviderNamedAccess::listSubscribedServices()
   The PA retrieves the list of services subscribed to by the user. This information is held by the UA. (The UA may retrieve this information from the SubscriptionAgent when it initializes, or at any other time.)

   The UA returns a list of the subscribed services, including properties of each of the services.

4.4 Usage related Scenarios

4.4.1 Start a Service Session and Selection Of session model

This scenario allows a user involved in an access session with a named or anonymous UA to start a service session that supports a specified session model. The UA is used as the generalization of both the anonUA and namedUA. There are two cases which are supported:

- Case 1: User starts a service session using an ssUAP. This assumes that the ssUAP is available in the user's domain and can be launched by the user and can obtain references to the access session PA.
- Case 2: User starts a service session using an asUAP. This does not require the ssUAP to be present in the user's domain provided there is a deployment/download capability and assumes the component can be launched by the PA.

4.4.1.1 Case 1: User starts a service session using an ssUAP

The ssUAP is present in the users domain and is launched by the user. The ssUAP holds references to the PA supporting the access session.

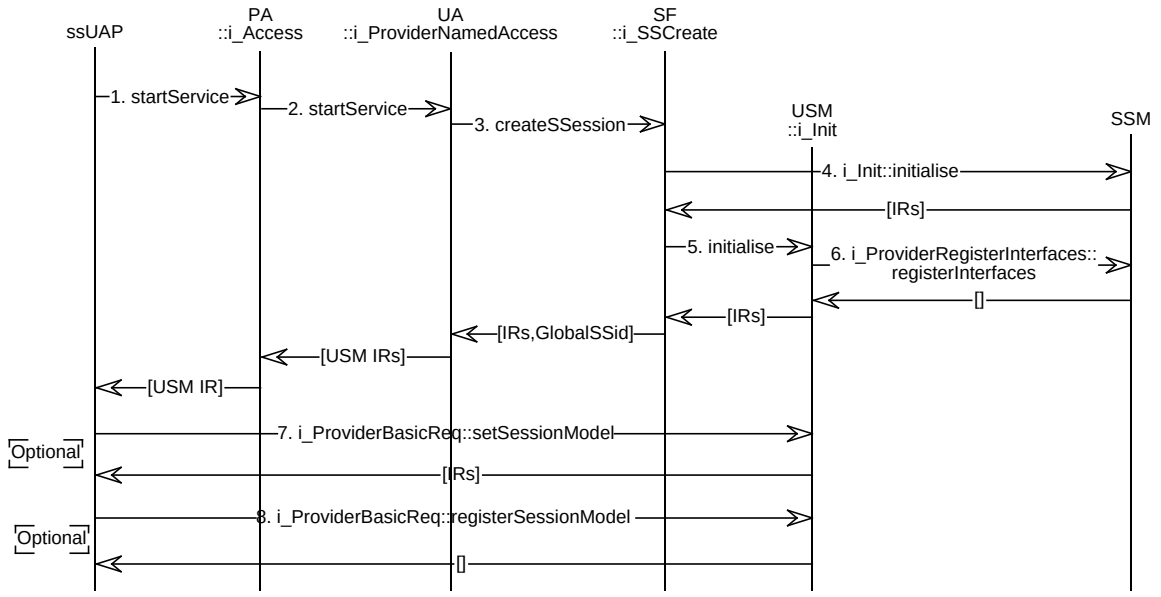


Figure 4-7. Case 1: User instructs an ssUAP to start a service session

1. ssUAP -> PA::i_Access::startService()
   requests from the PA a new service session of a particular service type which supports a specified session model (e.g. TINA service session graph). Other properties for the service can be specified. The ssUAP may also give the interface types and references it will support in the session.
2. PA -> UA::i_ProviderNamedAccess::startService()
   requests from the UA a new service session of a particular service type which supports a specified session model (e.g. TINA service session graph). Other properties for the service can be specified. The PA may also give the interface types and references that the ssUAP will support in the session. The UA may perform (various non-prescriptive) authorization and personalization actions before continuing. The UA may return unsuccessful and raise an exception to the PA if the service request is declined. The UA must obtain a reference to a service factory to create the session components. This may be predefined or the result of a scoped search on a trader.
3. UA -> SF::i_SSCreate::createSSession()
   requests both the USM and SSM session components to be instantiated and initialized with optional properties specified by the UA
4. SF -> SSM::i_Init::initialise()
   the SSM is initialized and interface references are returned
5. SF -> USM::i_Init::initialise()
   the USM is initialized with SSM references and USM interface references are returned.
6. USM -> SSM::i_ProviderRegisterInterfaces::registerInterfaces()
   SSM is given USM interface references by the USM.

References to the USM are returned to UA and ssUAP respectively.

ssUAP has references to interfaces on the USM. If the references are not to i_ProviderBasicReq interfaces then they have bespoke behavior and are undefined in TINA, this scenario has completed. The UAP is able to interact with the USM using the references to the Ret or service-specific interfaces.

The following optional scenario is supported by the ProviderBasic feature set defined in Ret. The exchange of interface types, references and session models across Ret is more capable if the ssUAP supports the ProviderBasicExt feature set. If ssUAP has references to USM::i_ProviderBasicReq interfaces, then the scenario may continue to allow the ssUAP and USM to select alternative interfaces for their interactions:

7. Optional: ssUAP -> USM::i_ProviderBasicReq::i_SessionModel::setSessionModel()
   requests the USM to return other possible session models supported by the service. This may return a list of models and optionally feature sets and interface references. If references are not returned they can be obtained using i_ProviderBasicReq::getInterface().

8. Optional: ssUAP -> USM::i_ProviderBasicReq::i_SessionModel::registerSessionModel()
   enables the ssUAP to inform the USM that it supports a specified session model and optionally feature sets and interface references. the return confirms that configuration is accepted.

4.4.1.2 Case 2: User starts a service session using an asUAP

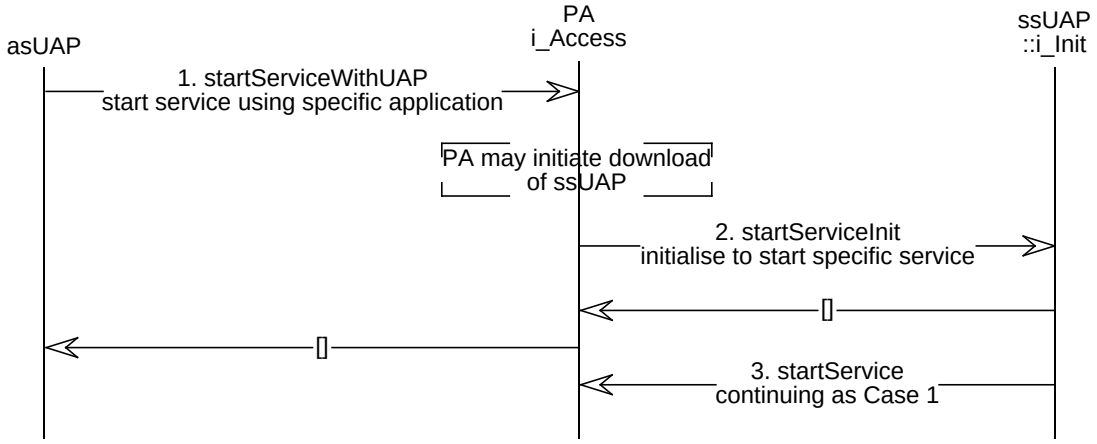


Figure 4-8. Case 2: User instructs asUAP to start a service session

This case allows service sessions to be started where an appropriate service-specific UAP does not exist on the terminal.

1. asUAP -> PA::i_Access::startServiceWithUAP()
   requests the a service session to be started using an optionally specified UAP. If necessary the PA will trigger the download/deployment of the ssUAP or launch a pre-existing ssUAP. See also ‘‘Update terminal with new service session components’ scenario. The PA starts the ssUap when it becomes available on the terminal and obtains the ssUAP::i_Init reference.

2. PA -> ssUAP::i_Init::startServiceInit()
   initializes the application to start a service session of a specified type; the PA interface reference is passed to the ssUAP.

3. ssUAP -> PA::i_Access::startService()
   This is equivalent as step 1 in Case 1 ‘ ssUAP to start service’ except the ssUAP responds to the initialization from the PA not the user.

The ssUAP uses the PA to start the session of the service type, as defined in 4.4.1.1.

4.4.2 End a Service Session

This scenario covers a user ending a service session. This ends the session for all the users of the session.

An access session exists between the PA and a UA. A service session exists between an ssUAP, USM and SSM.

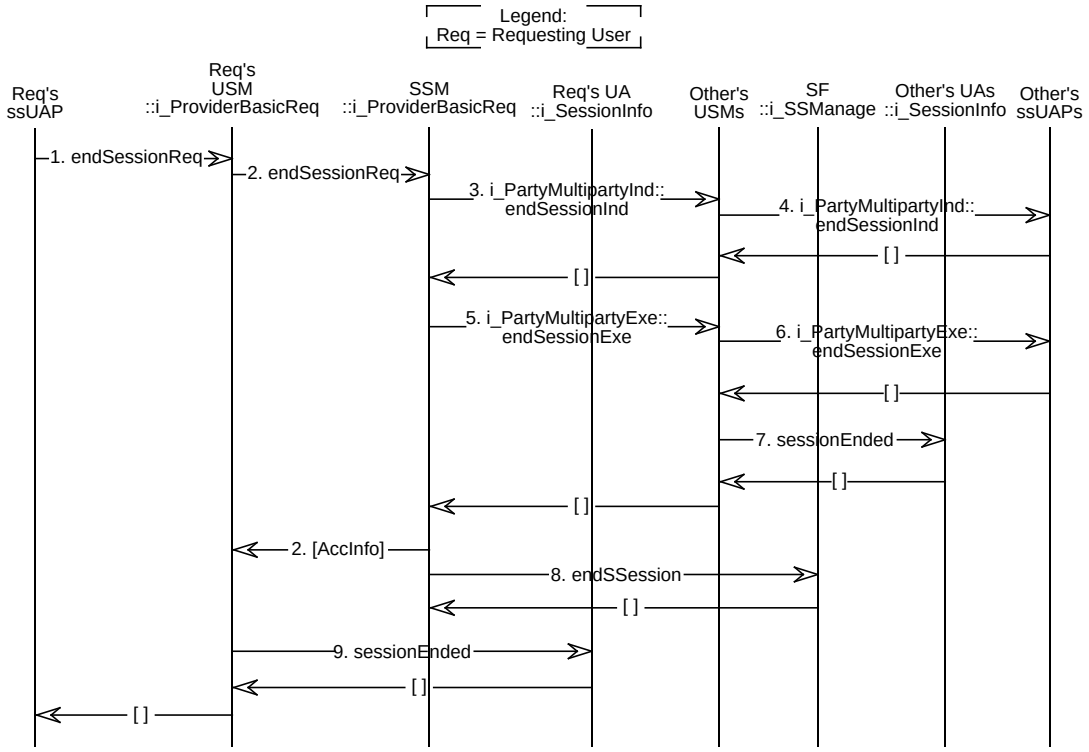


Figure 4-9. End a Service Session through ssUAP

1. ssUAP -> USM::i_ProviderBasicReq::endSessionReq()
   The user requests (through the ssUAP) the session to be ended.
2. USM -> SSM::i_ProviderBasicReq::endSessionReq()
   Request is passed on to the SSM. The return value of this invocation is the accounting information for the requesting user.
3. SSM -> USM::i_PartyMultipartyInd::endSessionInd()
   The SSM sends an indication about the request to end the session to all other user's USMs
4. USM -> ssUAP::i_PartyMultipartyInd::endSessionInd()
   The indication is passed on to the respective ssUAP's. Involved parties may vote on the decision to end the session, parties with owner rights might refuse, but assuming it is accepted to end the session:
5. SSM -> USM::i_PartyMultipartyExe::endSessionExe()
   The SSM sends an Exe to all other USMs instructing them to end the session. Accounting information is passed to the USMs with this invocation. This is considered the 'point of no return'; if a provider so chooses, the initial invocation can return here (similar to the

asynchronous version of e.g. Section 4.4.11). All accounting has stopped and the service session specific interfaces are no longer available to the user.

6. USM -> ssUAP::i_PartyMultipartyExe::endSessionExe()
   The instruction is passed to the respective ssUAPs.

7. USM -> UA::i_SessionInfo::sessionEnded()
   All other participants UA's are informed about the session being ended and accounting information is transferred. If the i_ProviderSessionInfo was passed as part of the setUsrCtxt, the UA sends an endSessionInfo to the PA (not shown in the diagram, applies to step 9. as well).

8. SSM -> SF::i_SSManage::endSSession()
   The SF is informed that the session is about to end and eventually releases the resources.

9. USM -> UA::i_SessionInfo::sessionEnded()
   The USM informs the requesting users UA about the session ending and transfers accounting information.

The UAP may still be running. Any references to the USM/SSM held by the UAP are invalid i.e., use of them will not cause operations to occur on any USM/SSM or other CO. Their use should raise an exception from the DPE.

4.4.3 End Service Session via Access Session

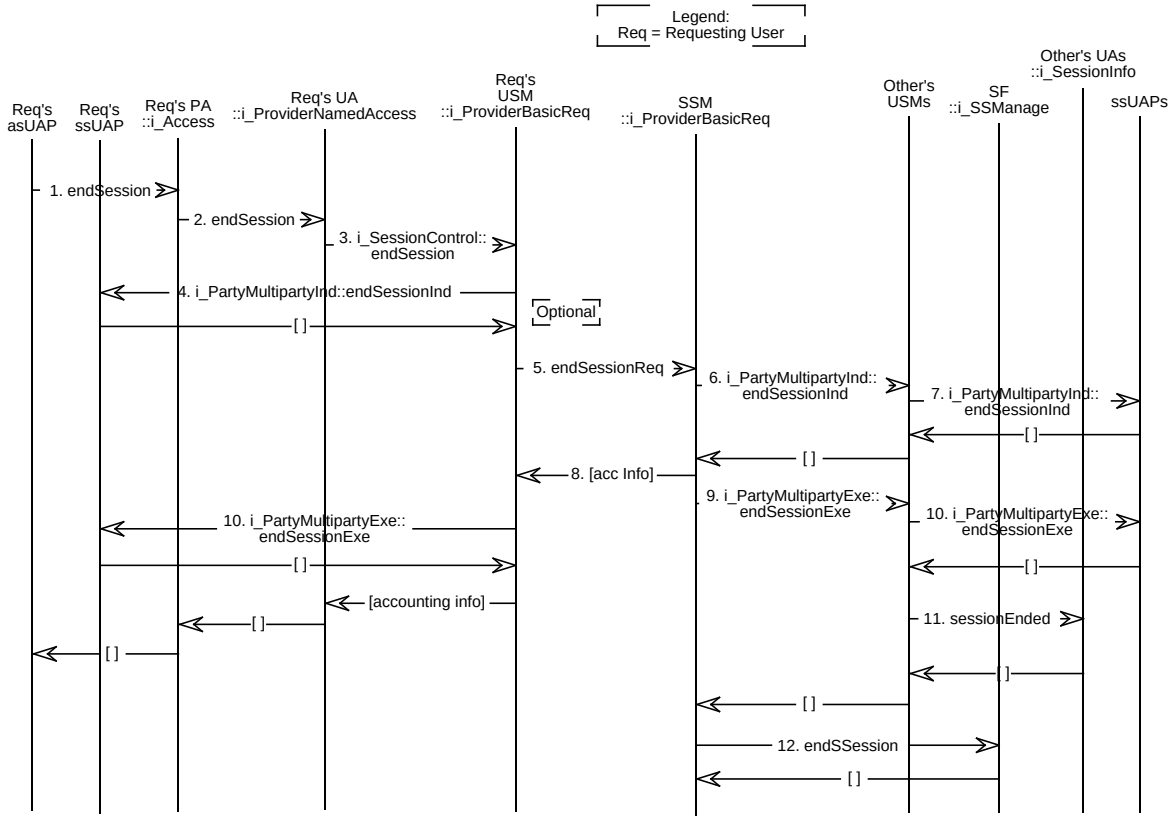


Figure 4-10. End service session (via access)

The access session used to end a particular service session may not be the access session responsible for starting the service session originally. In the general case the asUAP used to end the service session and the ssUAP participating in the service may not be collocated and therefore not ordinarily they are not likely to hold interface references to each other.

1. asUAP -> PA::i_Access::endSession()
   The asUAP instructs the PA to end the specified session. The session identifier can have previously been obtained using PA::i_Access::listServiceSessions()
2. PA -> Req's namedUA::i_ProviderNamedAccess::endSession()
   The PA instructs the namedUA to end the specified service session.
3. namedUA -> Req's USM::i_SessionCtrl::endSession()
   The namedUA which holds a reference to the USM instructs USM to end the session. If this is permitted the next stages occur.
4. USM -> ssUAP::i_PartyMultipartyInd::endSessionInd()
   Optionally the USM may give the participant's ssUAP indication that the endSession request has been received. This may assist the ssUAP to clear down the session gracefully in advance of the final 'Exe' message.
5. USM -> SSM::i_ProviderBasicReq::endSessionReq()
   The USM issues the endSession request to the SSM. The stages to 10 follow the same format as the end session scenario initiated from the participating ssUAP.
6. SSM -> Other's USM::i_PartyMultipartyInd::endSessionInd()
   The SSM issues indications to the other USMs.
7. USM -> ssUAP::i_PartyMultipartyInd::endSessionInd()
   The other USM's may issue indications to their respective ssUAP.
8. The SSM confirms to the requesters USM that the session is ending. Accounting information are passed to the USM (accounting info will be then passed back by the USM to the UA).
9. SSM -> other's USM::i_PartyMultipartyExe::endSessionExe()
   The SSM instructs all USMs but the requester's to end. Accounting information are sent to USMs.
10. USM -> ssUAP::i_PartyMultipartyExe::endSessionExe()
    All USMs forward the Exe to their respective ssUAPs.
11. USM -> namedUA::i_SessionInfo::sessionEnded()
    Other participants UAs are informed that the session has ended. Accounting information is passed to the UAs by the USMs
12. SSM -> SF::i_SSManage:::endSSession()
    The SSM informs the service factory that the session has ended and resources have been released.

4.4.4 End Participation in a Service Session

This scenario covers a user ending participation in a service session. This leaves other participants continuing in the session. The event sequence bears a close relationship to end service session.

An access session exists between the PA and a UA. A service session exists between a UAP, USM and SSM.)

In this scenario, we assume that requesting user differs from the user whose end of participation is requested. It is possible for a user to use the same operations for ending the participation of that user as well, but then the scenario follows the one given below in Figure 4-12.

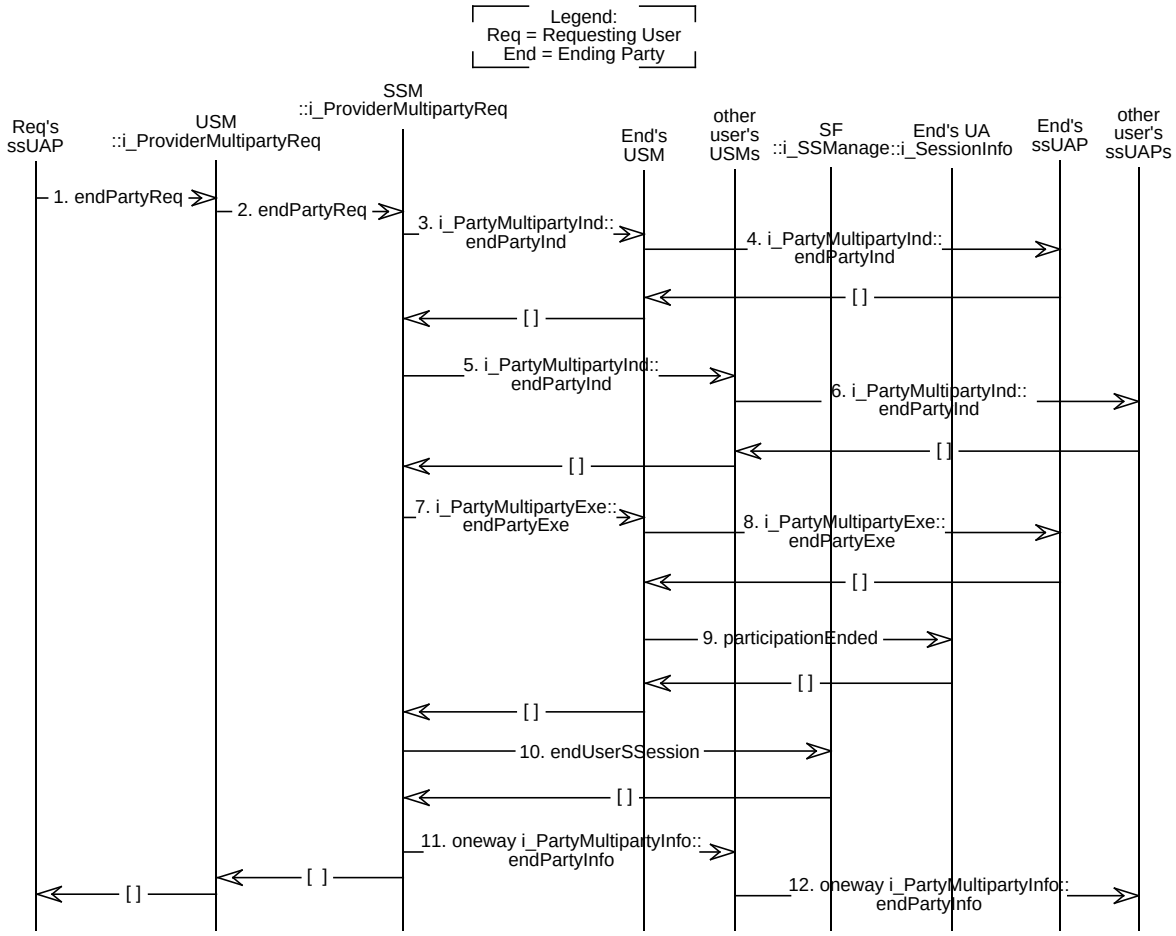


Figure 4-11. End Participation in a Service Session

1. ssUAP -> USM::i_ProviderMultipartyReq::endPartyReq()
   A user requests the USM to end the participation for a given user.
2. USM -> SSM::i_ProviderMultipartyReq::endPartyReq()
   USM forwards the request to SSM. This operation will return, if successful, accounting info.
3. SSM -> Ending user's USM::i_PartyMultipartyInd::endPartyInd()
   If the user being requested to end participation has the right to reject the request, an indication is sent to ending user's USM
4. USM -> Other users ssUAP::i_PartyMultipartyInd::endPartyInd()
   The indication is forwarded to ending user's ssUAP.
5. SSM -> Other users USM::i_PartyMultipartyInd::endPartyInd()
   All other users (with voting, write or ownership rights for the session) receive indications that the participant is about to end participation.
6. USM -> Other users ssUAP::i_PartyMultipartyInd::endPartyInd()
   Indication is forwarded to corresponding ssUAPs. If the request is rejected by sessions owner(s) or by votes, an exception is raises and the scenario stops, otherwise:

7. SSM -> Ending user's USM::i_PartyMultipartyExe::endPartyExe()
   SSM instructs USM of the ending user to end the participation. Accounting info is passed to the USM. Point of no return (see Section 4.4.2 step 5.)
8. USM -> Ending user's ssUAP::i_PartyMultipartyInd::endPartyInd()
   The instruction is passed on to the user ssUAP, upon return, the ssUAP can no longer access interfaces on the USM
9. SSM -> Ending user's UA::i_SessionInfo::participationEnded()
   The ending user's UA is informed that the participation in the service session is ended, accounting info is passed to UA
10. SSM -> SF::i_SSManage::endUserSSession()
    The SF is instructed to release resources previously reserved for the ending user. In this scenario the SF can kill the ending user's USM immediately.
11. SSM -> Other users USM::oneway i_PartyMultipartyInfo::endPartyInfo()
    All other USMs are informed that the ending user is no longer participating in the service session.
12. USM -> ssUAP::i_PartyMultipartyInfo::endPartyInfo()
    Information passed to ssUAPs

A special case is shown in Figure 4-12. where a user directly ends her participation.

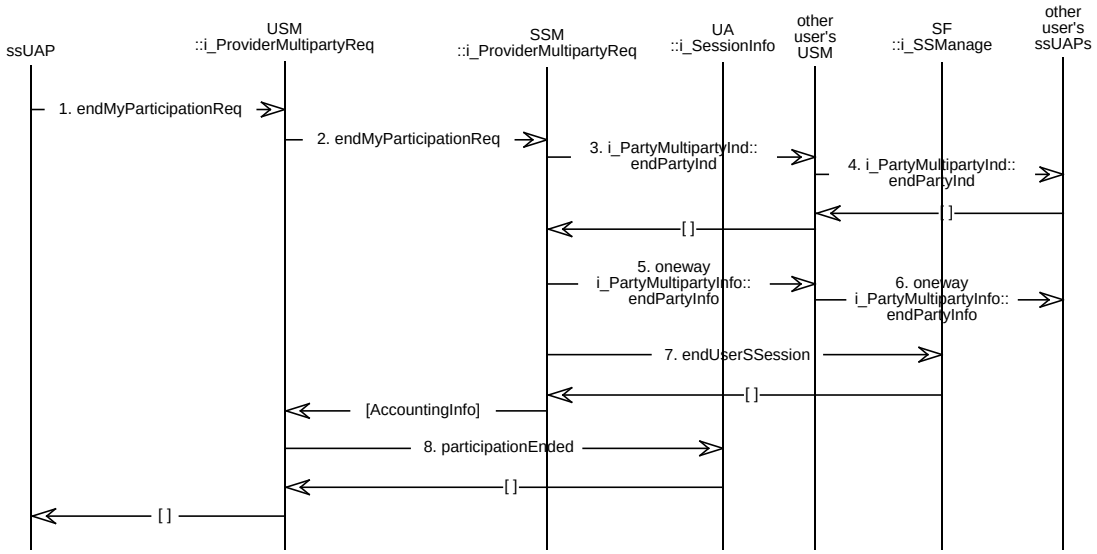


Figure 4-12. End My Participation in a Service Session

It follows the same principle as the general case, main differences are that the initial request is a different operation (endMyParticipation) and that no indications or Exe's are sent to the ending user.

Furthermore, the SF can not kill the USM immediately, but must wait till the USM has returned (see Section 4.4.2 for a possible solution). The accounting information is here passed as a return parameter to the call from USM to SSM and the call to the UA occurs after that return.

The ssUAP may still be running. Any references to the USM/SSM held by the ssUAP (of the ending user) are invalid i.e., use of them will not cause operations to occur on any USM/SSM or other CO. Their use should raise an exception from the DPE.

4.4.5 Suspend a Service Session

The user and provider have an access session. The user is involved in a service session with active ssUAP, USM and SSM.

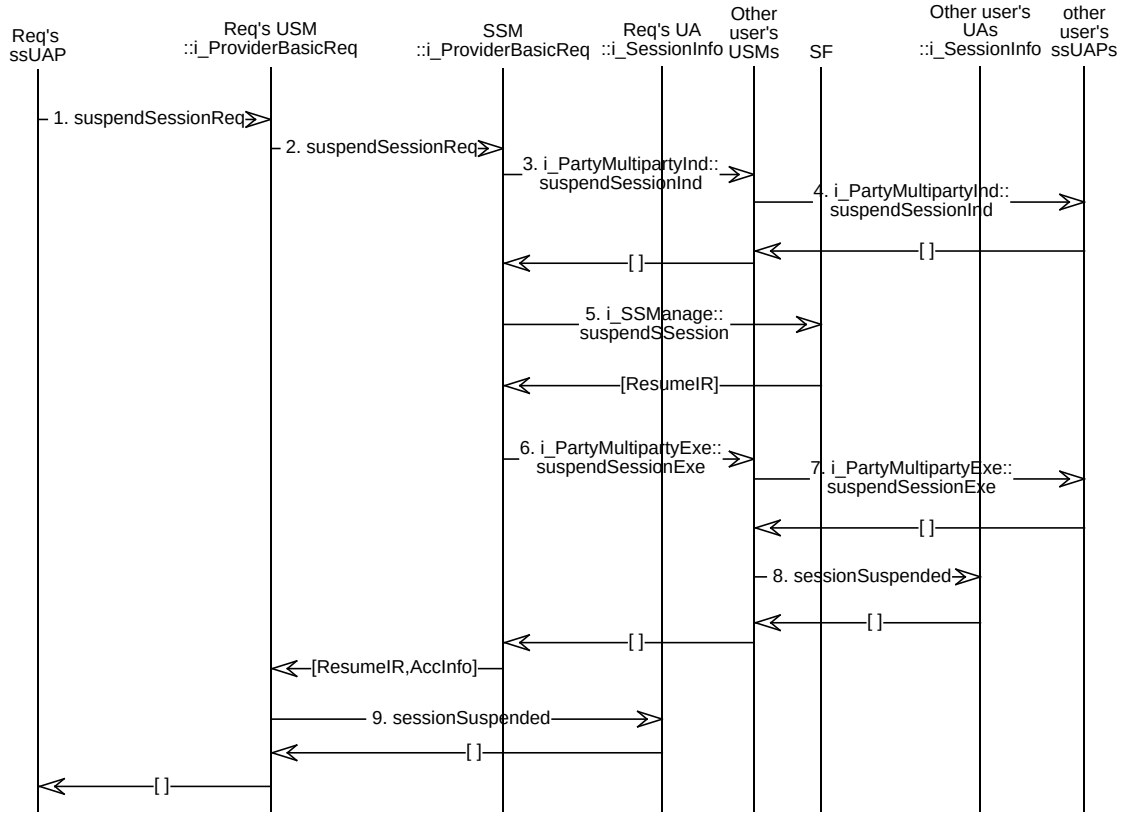


Figure 4-13. Suspend a Service Session

1. ssUAP -> USM::i_ProviderBasicReq::suspendSessionReq()
   UAP requests USM to suspend the service session.
2. USM -> SSM::i_ProviderBasicReq::suspendSessionReq()
   Request is passed to SSM. This operatio returns Resume interface reference and Accounting information.
3. SSM -> Other user’s USM::i_PartyMultipartyInd::suspendSessionInd()
   USM’s of users with decision rights receive indications, that the session is about to be suspended
4. USM -> Other user’s ssUAP::i_PartyMultipartyInd::suspendSessionInd()
   Indication is passed to ssUAPs. If the suspension is rejected, the thread stops with an exception, otherwise:
5. SSM -> SF::i_SSManage::suspendSSession()
   SSM informs SF that the session will be suspended. The return value is an interface reference on the SF that can be used to resume the session.

6. SSM -> Other user's USM::i_PartyMultipartyExe::suspendSessionExe()
   Other USMs are instructed to suspend the session. The resume IR and relevant accounting information is passed to each USM. Point of no return (see Section 4.4.2 step 5.)
7. USM -> ssUAP::i_PartyMultipartyExe::suspendSessionExe()
   The instruction is passed on to each ssUAP
8. USM -> Other user's UA::i_SessionInfo::sessionSuspended()
   Each UA is informed about the suspension. The resume IR and accounting info is stored in the UAs.
9. USM -> Requesting user's UA::i_SessionInfo::sessionSuspended()
   Similar action for the requesting user.

The session descriptor can be used later by the UAP or be retrieved by the UA so that resumption of participation can be from a different terminal. The service session ceases to become interactive for all users of the session. Depending on the semantics of the service, the session state when resumed may not correspond to the moment of suspension. Possible communication sessions are ended.

4.4.6 Suspend Participation in a Service Session

'Suspend participation in a service session' scenario may be used by users who simply wish temporarily not to be involved in a service session. It may be resumed from within the same access session, but it can also be used to allow session mobility and resumed within a different access session, possibly from a different location. The 'Resume Participation in a Service Session' scenario is used to resume the session, whether session mobility occurs or not.

As was the case for 'End Participation in a Service Session', two cases are considered, a general case, where a user requests a given users participation to be suspended, and a specialized case where the user whose participation is requested to be suspended is actually the requesting user.

The user and provider have an access session. The user to be suspended is involved in a multiparty service session with active ssUAP, USM and SSM.

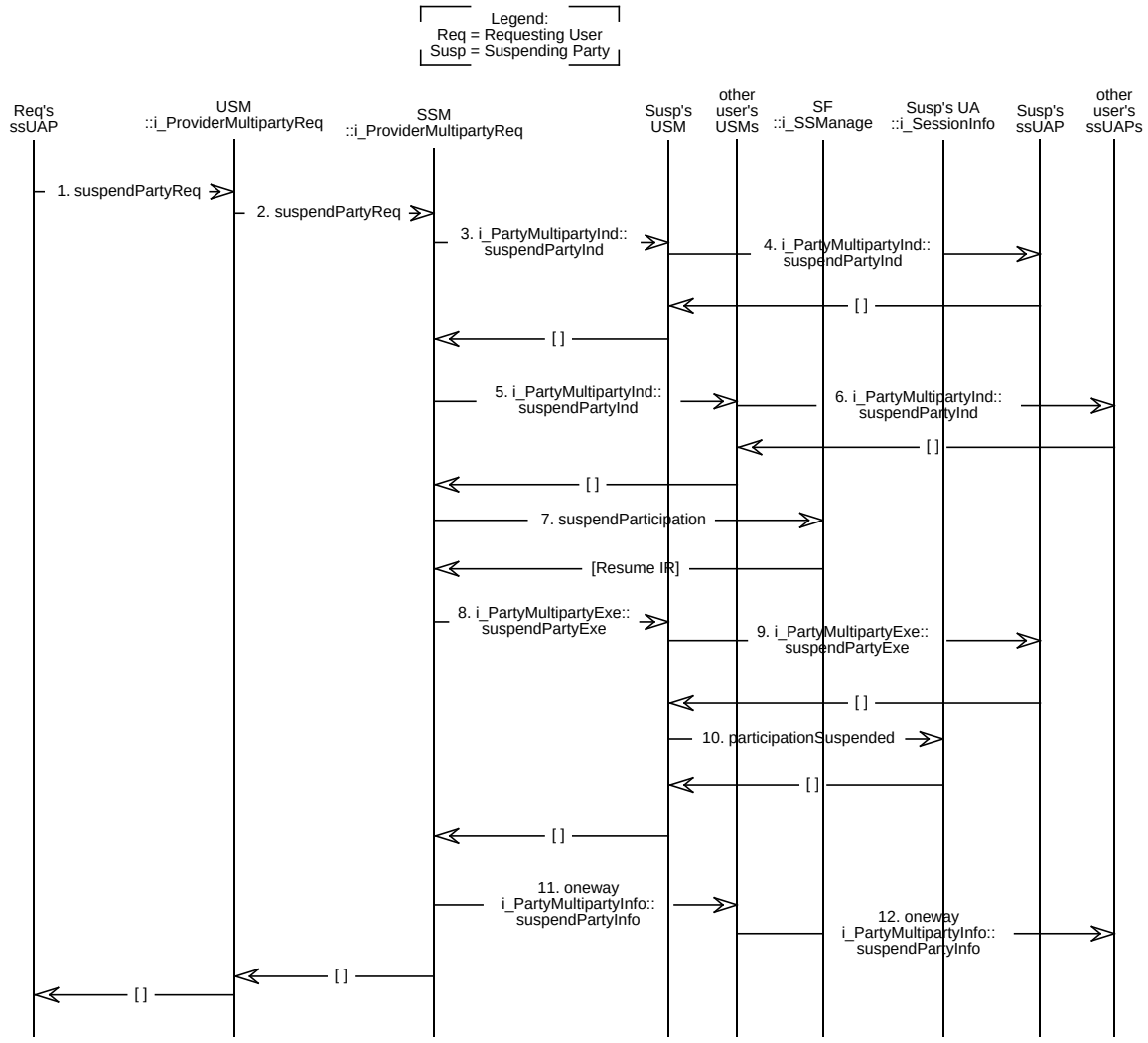


Figure 4-14. Suspend Participation in a Service Session

This scenario is very similar to ‘End Participation in a Service Session’. A few exceptions are given below:

1. ssUAP -> USM::i_ProviderMultipartyReq::suspendPartyReq()
2. USM -> SSM::i_ProviderMultipartyReq::suspendPartyReq()
3. SSM -> Suspending user’s USM::i_PartyMultipartyInd::suspendPartyInd()
4. USM -> Suspending user’s ssUAP::i_PartyMultipartyInd::suspendPartyInd()
5. SSM -> Other user’s USM::i_PartyMultipartyInd::suspendPartyInd()
6. USM -> Other user’s ssUAP::i_PartyMultipartyInd::suspendPartyInd()
7. SSM -> SF::i_SSManage::suspendParticipation()
   The SF returns a resume interface reference to be used to resume participation.

8. SSM -> Suspending user's USM::i_PartyMultipartyExe::suspendPartyExe()
   An interface reference for resuming the SSM is passed with this call. Point of no return (see Section 4.4.2 step 5.)
9. USM -> Suspending user's ssUAP::i_PartyMultipartyExe::suspendPartyExe()
   Interface references for resuming SSM and suspending users USM are passed to the ssUAP
10. USM -> Suspending user's UA::i_SessionInfo::participationSuspended()
    Interface references for resuming SSM and suspending users USM are passed to the UA together with the accounting info.
11. SSM -> Other user's USM::i_PartyMultipartyInfo::suspendPartyInfo()
12. USM -> Other user's ssUAP::i_PartyMultipartyInfo::suspendPartyInfo()

Figure 4-15. shows a special case of suspend participation, where Req and Susp coincide.

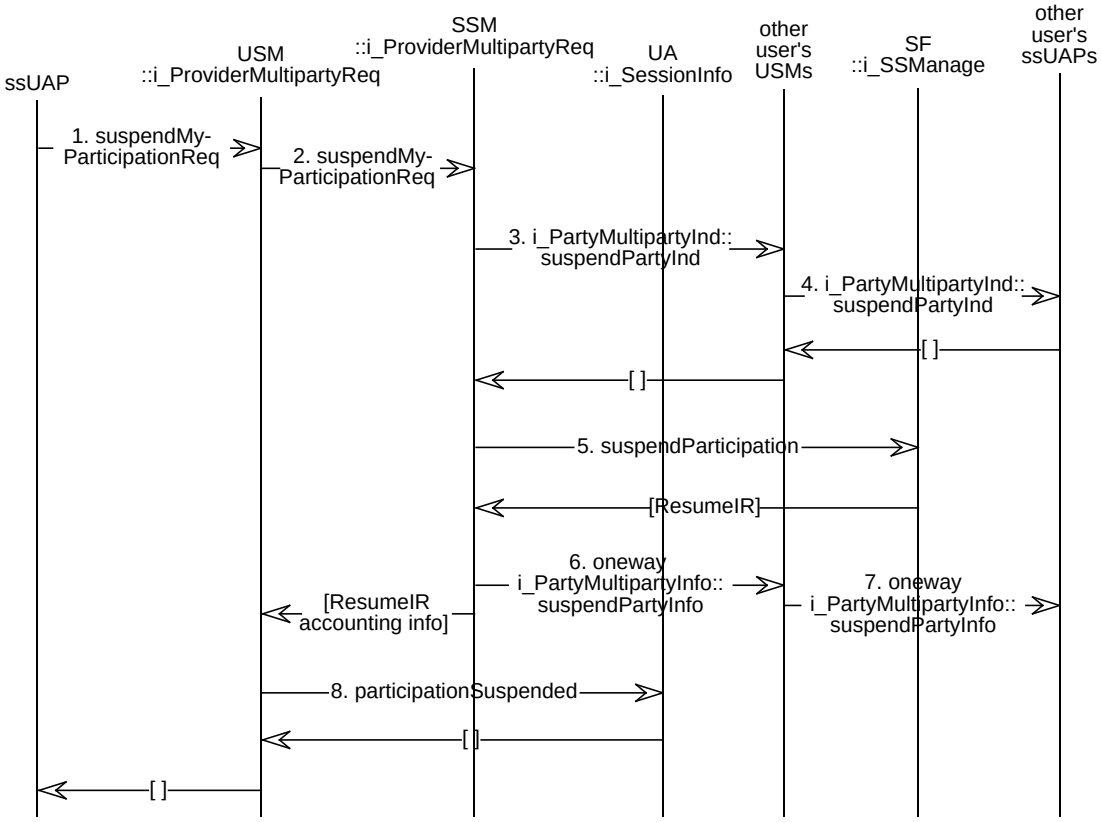


Figure 4-15. Suspend My Participation in a Service Session

'Suspend My Participation in a Service Session' differs in the same way from 'Suspend Participation in a Service Session' as the two cases of End Participation differ.

The resume interfaces can be used later by the ssUAP or be retrieved by the UA so that resumption of participation can be from a different terminal. The service session ceases to be interactive for the user who suspended participation. Depending on the semantics of the service the session state when resumed may not correspond to the moment of suspension. Possible communication sessions are ended if all participants are suspended.

4.4.7 Resume a Service Session

This event trace shows how a user can resume a service session that (s)he previously suspended. It is assumed that the user is allowed to do so

As when resuming participation in a service session (see 4.4.8) the user can start requesting from the provider a list of suspended service sessions; since these steps are exactly the same as in 4.4.8 they will be omitted here.

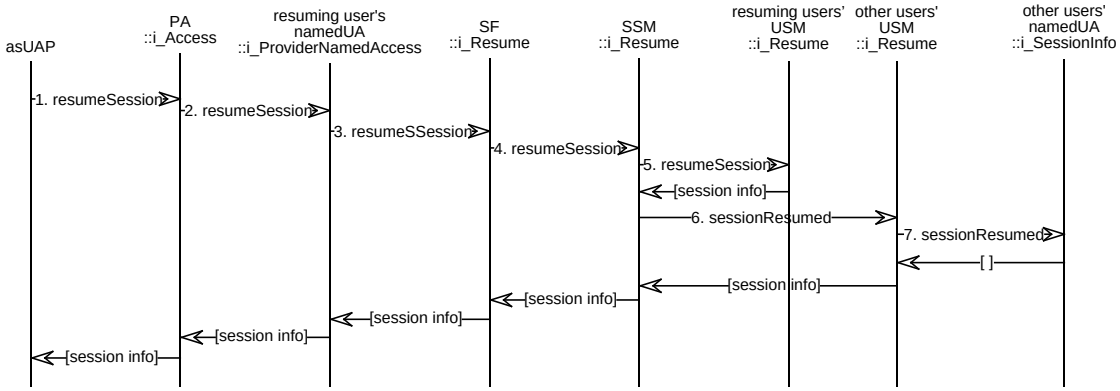


Figure 4-16. Resume a Service Session

1. asUAP -> PA::i_Access::resumeSession()
   The user uses an access session UAP to forward the PA the request to resume the suspended service session.

2. PA -> namedUA::i_ProviderNamedAccess::resumeSession()
   The PA forwards the request to the user's namedUA.

3. namedUA -> SF::i_Resume::resumeSSession()
   The user's namedUA uses the resume interface reference obtained in the suspension interactions to request from a service factory the resumption of the service session.

   Notice that it is not necessary that the resuming user is the one that suspended the service session. It is only necessary that the user has the necessary rights, but all service session participants get the resume interface at the end of the service session suspension interactions so they have the means to resume the session.

4. SF -> SSM::i_Resume::resumeSession()
   The SF forwards the resume request to the service session SSM. The SSM checks that particular user is allowed to request

5. SSM -> resuming user's USM::i_Resume::resumeSession()
   The SSM informs the USM of the user that started the resume interactions, that the session has resumed.

   The USM returns information about the service session, including the interface references necessary to interact with it.

6. SSM -> other participants' USM::i_Resume::sessionResumed()
   The service session's SSM informs then the USMs of the user sessions of all the rest of the session participants that the session has resumed.

7. USM -> namedUA::i_SessionInfo::sessionResumed()
   Each USM in turn informs its corresponding namedUA that the session has resumed, so

namedUAs can have an updated list of service sessions the corresponding user is participating in, together with the session status.

After receiving returns from the UAs, the USMs return the service session information to the SSM.

Returns notifying of the success of the resume operation go back step by step to the user, carrying the service session information to the asUAP; the asUAP will communicate with the ssUAP to forward to it this service session information, but these interactions are out of the scope of these component specifications.

4.4.8 Resume Participation in a Service Session

This event trace shows how a user can resume his/her participation in a service session after having suspended it. It is assumed that the service session is still going on, otherwise the user would have been notified of its termination. It is also assumed that the service session has not changed during the time the user's participation was suspended; see [1] for a study of the possibilities in case the service session does change.

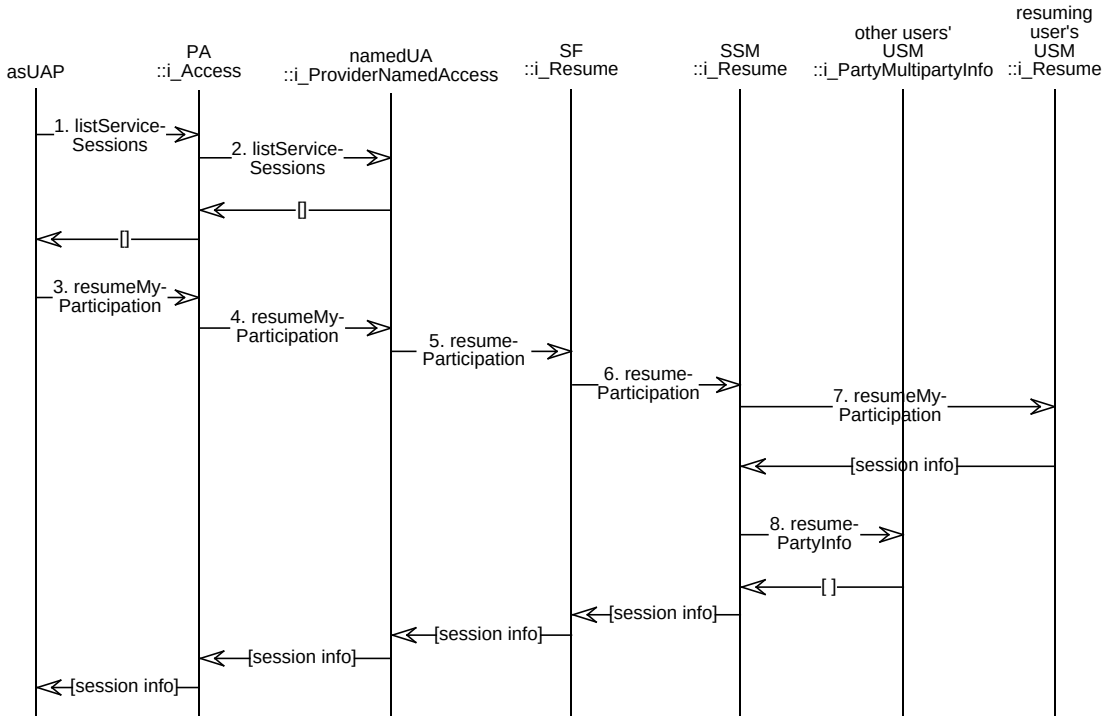


Figure 4-17. Resume Participation in a Service Session

1. asUAP -> PA::i_Access::listServiceSessions()
   The user uses an access related UAP to forward the PA the request for a list of service sessions; the request will be scoped to include only service sessions that the user was taking part of and where (s)he suspended his/her participation.

2. PA -> namedUA::i_ProviderNamedAccess::listServiceSessions()
   The PA forwards the request to the user's namedUA; it is assumed here that in this access session the user logged in as a known user, but the anonymous case is formally identical.

The namedUA returns the list to the PA, that forwards it to the asUAP, which in turn forwards it to the user, who chooses among them the service session where (s)he wants to resume his/her participation.

These two first steps may not be necessary, but they are shown here to make this example more complete.

3. asUAP -> PA::i_Access::resumeMyParticipation()
   The user uses the access related UAP to forward the PA the request of resuming his/her participation in a certain service session, identified by a service session id.
4. PA -> namedUA::i_ProviderNamedAccess::resumeMyParticipation()
   The PA forwards the request to the user's namedUA
5. namedUA -> SF::i_Resume::resumeParticipation()
   The user's namedUA uses the resume interface reference obtained in the previous suspension interactions to request from a service factory the resumption of his/her participation.
6. SF -> SSM::i_Resume::resumeParticipation()
   The SF informs the service session's SSM of the user's participation resumption.
7. SSM -> Resuming user's USM::i_Resume::resumeMyParticipation()
   The service session's SSM informs the user session's USM of the user's participation resumption; the USM returns asome information about the service session, including the interface references necessary to interact with it.
8. SSM -> Other user's USM::i_PartyMultipartyInfo::resumePartyInfo()
   The same notification is sent to the USMs corresponding to the user sessions of the other users in this service session.

   Returns to indicate the successful completion of the process go back from the SSM to the SF, then to the namedUA, to the PA, and to the asUAP that informs the user. The service session information is returned until it reaches the ssUAP.

4.4.9 Invite a User to Join a Session

This scenario defines how an existing service session participant can invite a potential participant to join in the service session. Although the invitee can respond immediately to the invitation the session is joined when the invitee invokes the join session scenario. The scenario does not show voting defined in Ret.

The user who issues the invitation (inviter) and other existing participants in the service session have active access sessions and service session components that have multiparty capabilities. The participant receiving the invitation (invitee) may have an access session, or has registered a PA invitation interface with the UA or, the UA cannot forward the invitation to the PA. Invitations require immediate responses to confirm their delivery, but subsequent responses are also possible

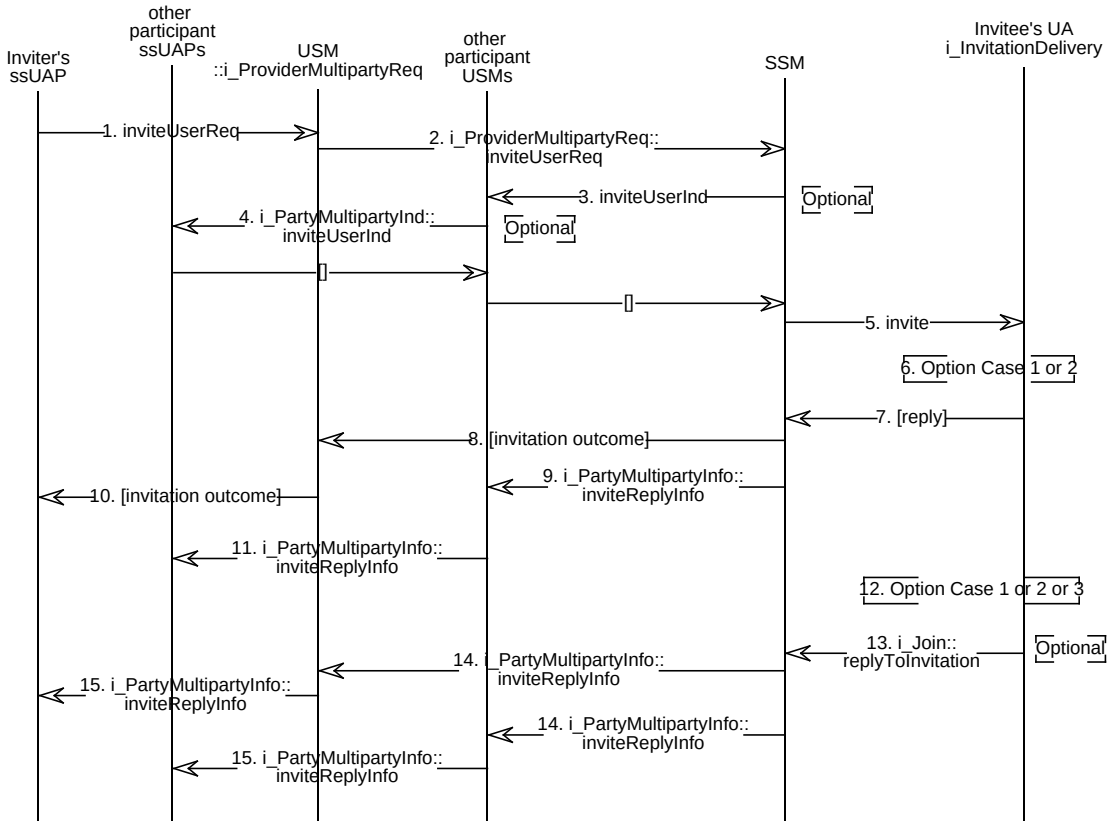


Figure 4-18. Case 1: Participant instructs ssUAP to invite a potential participant to join service session

1. UAP -> USM::i_ProviderMultipartyReq::inviteUserReq()
   instructs the USM to invite a user to join the session. An existing participant (inviter) has instructed the ssUAP to issue an invitation to a potential participant (invitee) to join a session. The inviter supplied a resolvable name/address of the invitee.

2. USM -> SSM::i_ProviderMultipartyReq::inviteUserReq()
   instructs the SSM to invite a user to join the session.

3. Optional SSM -> USM::i_PartyMultipartyInd::inviteUserInd()
   optionally informs other USMs in the existing session that a new user is about to be invited to join. This allow the service to support USM voting or invitation blocking at this point not considered further here.

4. Optional USM -> ssUAP::i_PartyMultipartyInd::inviteUserInd()
   optionally informs other USMs in the existing session that a new user is about to be invited to join. This allow the service to support participant voting or invitation blocking at this point not considered further here. See Section 4.4.15, "Example of Voting Procedure".

5. SSM -> UA::i_InvitationDelivery::invite()
   delivers the invitation to the invitee's UA.
   The SSM uses a trading service to resolve the invitee's name/address to a uaInvite interface reference which can be used to issue the invitation. The naming service will be federated amongst retailer and third party domains. The methods by which the location trading service is established or maintained is outside this event trace. The naming service returns a reference to

the invitee's UA::i_InvitationDelivery. It is assumed that the domain federation (including DPE security) permits this when the invitee and inviter have UAs residing in different retail domains. The UA may perform actions which are not prescribed, before continuing. For example, the UA may check its user profile for a policy on invitations to this specific service and conditions. The policy will determine the UA actions. A number of options are possible.

6. Optional Case 1. The UA can decide to forward the invitation to the invitee's terminal before replying (see Figure 4-19).
7. UA -> SSM returns
   the invitation outcome. A number of outcomes are defined.
8. SSM -> USM return
   to invited user giving invitation outcome.
9. SSM -> USM::i_PartyMultipartyInfo::inviteReplyInfo()
   informs other USMs of a invitation reply
10. USM -> ssUAP return
    to invited user giving invitation reply outcome
11. USM -> ssUAP::i_PartyMultipartyInfo::inviteReplyInfo()
    of other participant UAPs informing of invitation outcome.
12. Optional Case 1 or 2 or Case 3.
13. Optional UA -> SSM::i_Join::replyToInvitation()
    a subsequent invitation reply can be made according to the outcome of Case 1 or 2 or 3 or due to a decision of the UA (e.g. the UA may time-out an invitation if the invitee cannot be contacted).
14. Optional SSM -> USM::i_PartyMultipartyInfo::inviteReplyInfo()
    the subsequent invitation reply is forwarded to USMs
15. Optional USM -> ssUAP::i_PartyMultipartyInfo::inviteReplyInfo()
    the subsequent invitation reply is forwarded to UAPs

Optional Case 1 and 2 - UA forwards the invitation to the invitee's terminal

This option can be used as part of the synchronous response to the deliver invitation invocations. However, this part of the event trace may occur at a significantly later time than step 9. The main feature is that the synchronous return of the SSM->UA invocation can decoupled from any subsequent interaction between the invitee's UA<->PA<->asUAP. Case 1 and 2 only differ by which interface and operation is used to deliver the invitation to the PA in step 1.

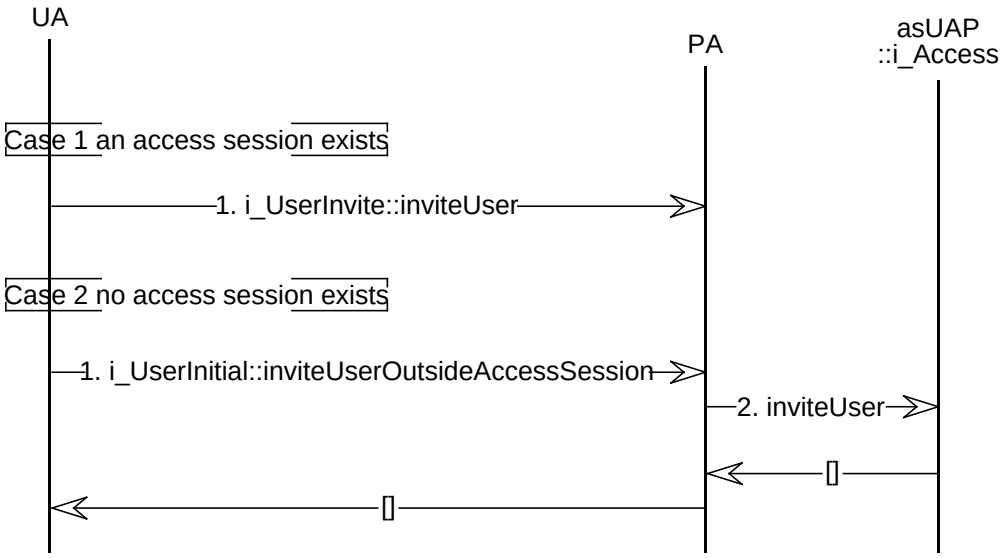


Figure 4-19. Case1+2 the invitee is/is not involved in an active access session

1. Case 1 UA -> PA::i_UserInvite::inviteUser()
   is used to deliver the invitation to a terminal If the UA holds an active PA invite interface inside the access session.

1. Case 2 UA -> PA::i_UserInitial::inviteUserOutsideAccessSession()
   delivers the invitation to a terminal if the UA holds a current PA reference for use outside an access session.

2. PA -> asUAP::i_Access::inviteUser()
   forwards this to the asUAP which can respond appropriately to the invitation on the invitee's behalf or in by interaction.

PA returns the invitation outcome to the UA across Ret. There are a number of possible responses to the invitation with semantics such as declined, ringing, accepted, and party unknown.

Optional Case 3

This option demonstrates how an invitation held by a UA that has already received a synchronous reply can be responded to later by the invitee. For example the UA may have returned to the SSM the status `trying` to contact the invitee. At a later time the invitee starts an access session and explicitly browses invitations received.

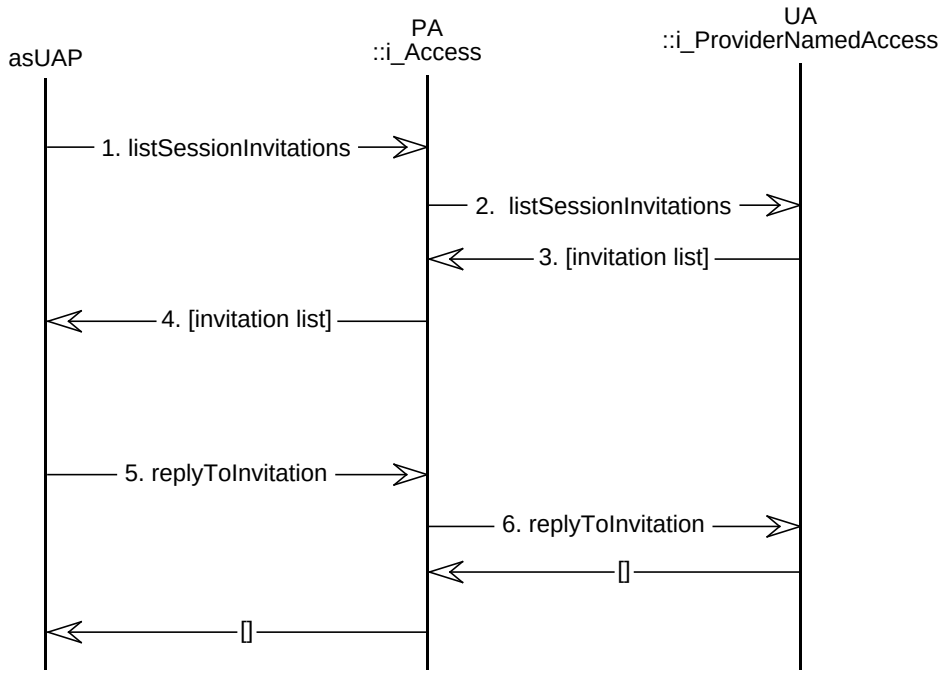


Figure 4-20. Case 3 the invitee is involved in an access session and requests invitations

1. asUAP -> PA::i_Access::listSessionInvitations()
   requests a list of invitations received by the PA.
2. PA -> UA::i_ProviderNamedAccess::listSessionInvitations()
   requests a list of invitations received by the UA.
3. UA -> PA returns
   list of invitations
4. PA -> asUAP returns
   list of invitations
5. asUAP -> PA::i_Access::replyToInvitation()
   responds to the invitation now held by asUAP
   The return gives the success of the invocation.
6. PA -> UA::i_ProviderNamedAccess::replyToInvitation()
   responds to the invitation. The return gives the success of the invocation.

4.4.10 Join a Service Session with invitation

The user and provider have an access session. The user has started a service session related UAP which holds a descriptor of the service session which is active . The session descriptor, the invitation ID and a Service Info[5] are assumed to have been made available to the ssUAP by an invitation scenario through the PA. During the invitation scenario

5. Service info can carry the SSM::i_join interface reference. This interface reference is then used by the UA to contact the SSM.

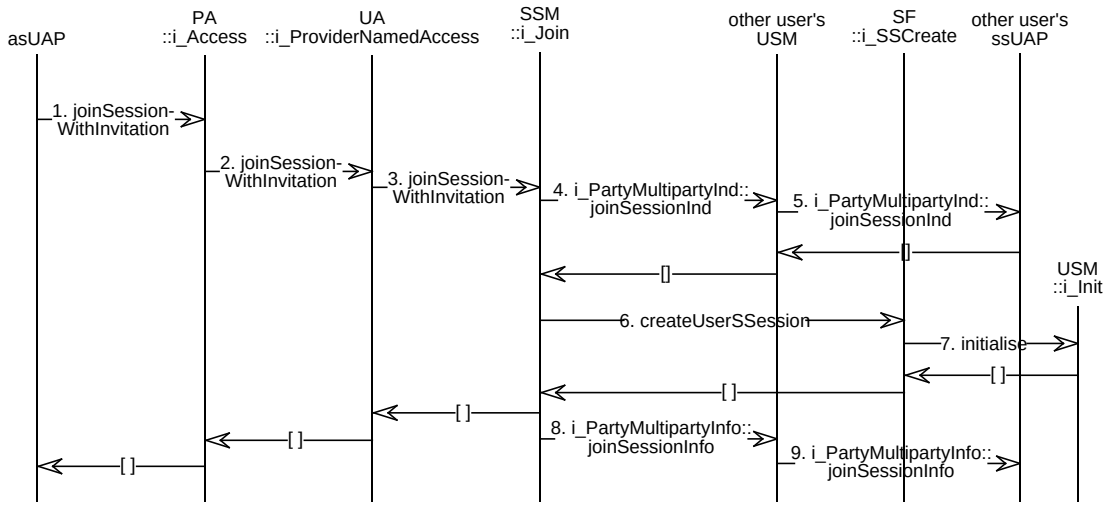


Figure 4-21. Join with Invitation Scenario

1. UAP -> PA::i_Access::joinSessionWithInvitation()
   User starts a service session related UAP. UAP gains a reference to the PA. User instructs UAP to join an identified session. UAP requests to join a session, identified by a session descriptor. This request is sent to the PA.

2. PA -> UA::i_ProviderNamedAccess::joinSessionWithInvitation()
   PA requests to join a session, identified by a session invitation ID, to the `i_ProviderNamedAccess` interface of the UA.
   UA performs actions before continuing. For example authorization decisions may be taken. UA may return unsuccessful, and raise an exception to the PA, if the UA wishes to refuse the join request.

3. UA -> SSM::i_Join::joinSessionWithInvitation()
   UA requests to join with invitation to the `i_Join` interface `of the SSM.`

4. SSM -> USM::i_PartyMultipartyInd::joinSessionInd()
   The SSM checks whether the invitation is still valid and then it forwards all USMs that a user has requested to join the session.

5. USM -> ssUAP::i_PartyMultipartyInd::joinSessionInd()
   USMs forwards their respective UAPs that a user has requested to join the session.

6. SSM -> SF::i_SSCreate::createUserSSession()
   The SSM processes the answers from the USMs and according to service specific policies decides whether to accept or not the join request. In case of acceptance SSM requests to create a User Service Session to the SF.

7. SF -> USM::i_Init::initialise()
   SF allocates resources for a new USM, creates an USM, and initialize it (in the initialization phase the needed references of UA, SSM and UAP are passed to the USM). The USM interface references and the needed information on the session (FSs and Session model) are passed back to the SSM, UA and the UAP.

8. SSM -> others' USMs::i_PartyMultipartyInfo::joinSessionInfo()
   SSM notifies all USMs that the user has joined the session

9. USMs -> UAPs::i_PartyMultipartyInfo::joinSessionInfo()
   The notification of step 8. is forwarded to UAPs

The invited user has joined the service session he had been invited to, the user remains involved in an access session.

4.4.11 Add Participant Oriented Stream Binding to a Service Session

This scenario covers the procedures to add a Participant Oriented Stream Binding to a service session. Both the synchronous and asynchronous procedures are covered.

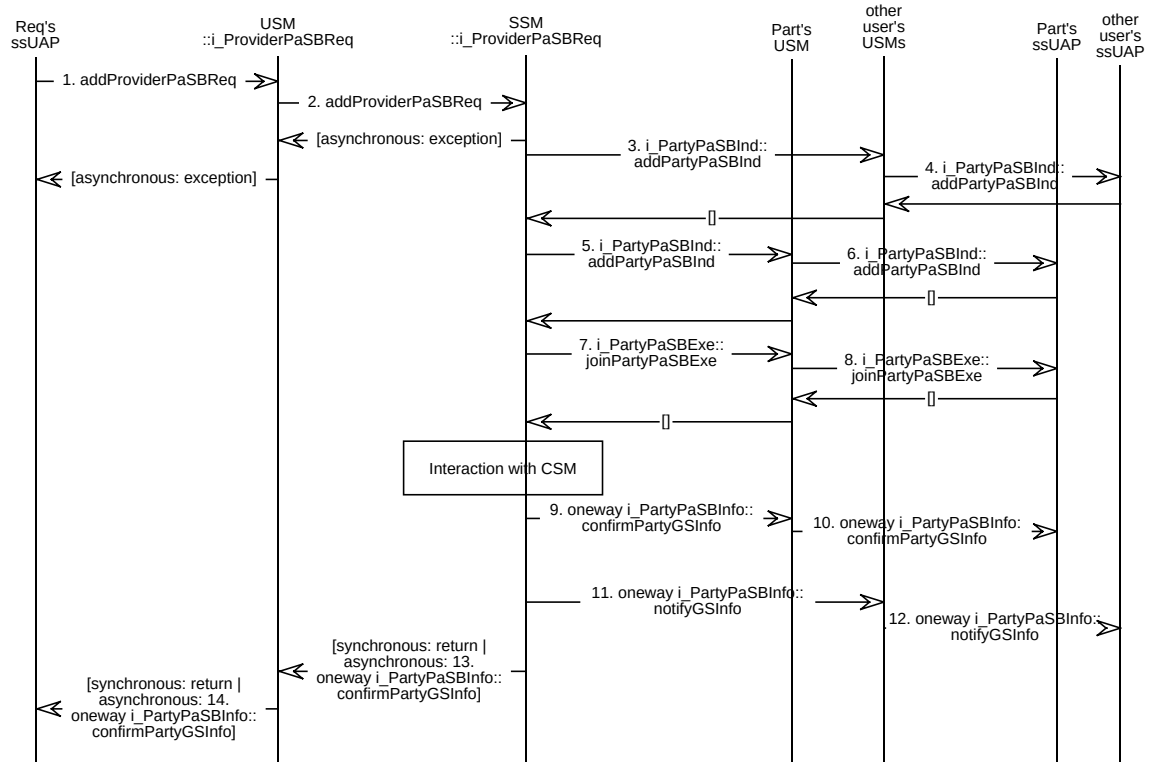


Figure 4-22. Add Participant Oriented Stream Binding to a Service Session

1. ssUAP -> USM::i_ProviderPaSBReq::addProviderPaSBReq()
   A party in the service session initiates the addition of the streambinding. The requesting party can already include a list of SFEPs he wants to bind. The request operation also includes a description of the media associated to the streambinding and a list of parties which need to be invited to join the streambinding.
2. USM -> SSM::i_ProviderPaSBReq::addProviderPaSBReq()
   The request is forwarded to the SSM. If the request is processed asynchronously, an e_NoSynchronousReqResp exception is thrown, which is forwarded to the invoking ssUAP.
3. SSM -> USM::i_PartyPaSBInd::addPartyPaSBInd()
   An addPartyPaSBInd is invoked on the USM of selected parties in the service session. These parties can be selected according to service-specific semantics or as a result of control session relationship settings in the TINA service session model.

4. USM -> ssUAP::i_PartyPaSBInd::addPartyPaSBInd()
   The indication is forwarded to the related ssUAPs.
5. SSM -> USM::i_PartyPaSBInd::addPartyPaSBInd()
   An addPartyPaSBInd is invoked on the USM of those parties which are being invited to join the streambinding.
6. USM -> ssUAP::i_PartyPaSBInd::addPartyPaSBInd()
   The indication is forwarded to the related ssUAPs.
   The optional voting procedure takes place. If the voting results in acceptance, the scenario continues:
7. SSM -> USM::i_PartyPaSBExe::joinPartyPaSBExe()
   A joinPartyPaSBExe is invoked on the USM of those parties which are being invited to join the streambinding.
8. USM -> ssUAP::i_PartyPaSBExe::joinPartyPaSBExe()
   The operation is forwarded to the related ssUAPs. The invited parties return a list of the SFEPs they want to bind as an out parameter of the operation. This list is further returned to the SSM as an out parameter of 7.
   When all invited parties have returned their SFEPs for binding, the SSM determines the related stream flow connections and interacts with the CSM to establish these connections. When the connections are established successfully, the scenario continues:
9. oneway SSM -> USM::i_PartyPaSBInfo::confirmPartyGSInfo()
   A confirmPartyGSInfo is invoked on the USM of those parties which have been invited in order to inform them that the procedures to add the streambinding have finished successfully.
10. oneway USM -> ssUAP::i_PartyPaSBInfo::confirmPartyGSInfo()
    The info is forwarded to the related ssUAPs.
11. oneway SSM -> USM::i_PartyPaSBInfo::notifyGSInfo()
    A notifyGSInfo is invoked on the USM of selected parties (as in 3.) to inform them that the procedures to add the streambinding have finished successfully.
12. oneway USM -> ssUAP::i_PartyPaSBInfo::notifyGSInfo()
    The info is forwarded to the related ssUAPs.
13. oneway SSM -> USM::i_PartyPaSBInfo::confirmPartyGSInfo()
    When the addition of the streambinding has been processed asynchronously, the requester is informed of the successful outcome by means of a confirmPartyGSInfo.
14. oneway USM -> ssUAP::i_PartyPaSBInfo::confirmPartyGSInfo()
    The info is forwarded to the requesting party's ssUAP.

4.4.12 Add Participants to a Participant Oriented Stream Binding

This scenario covers the procedures to add new Participants to an already established Participant Oriented Stream Binding.

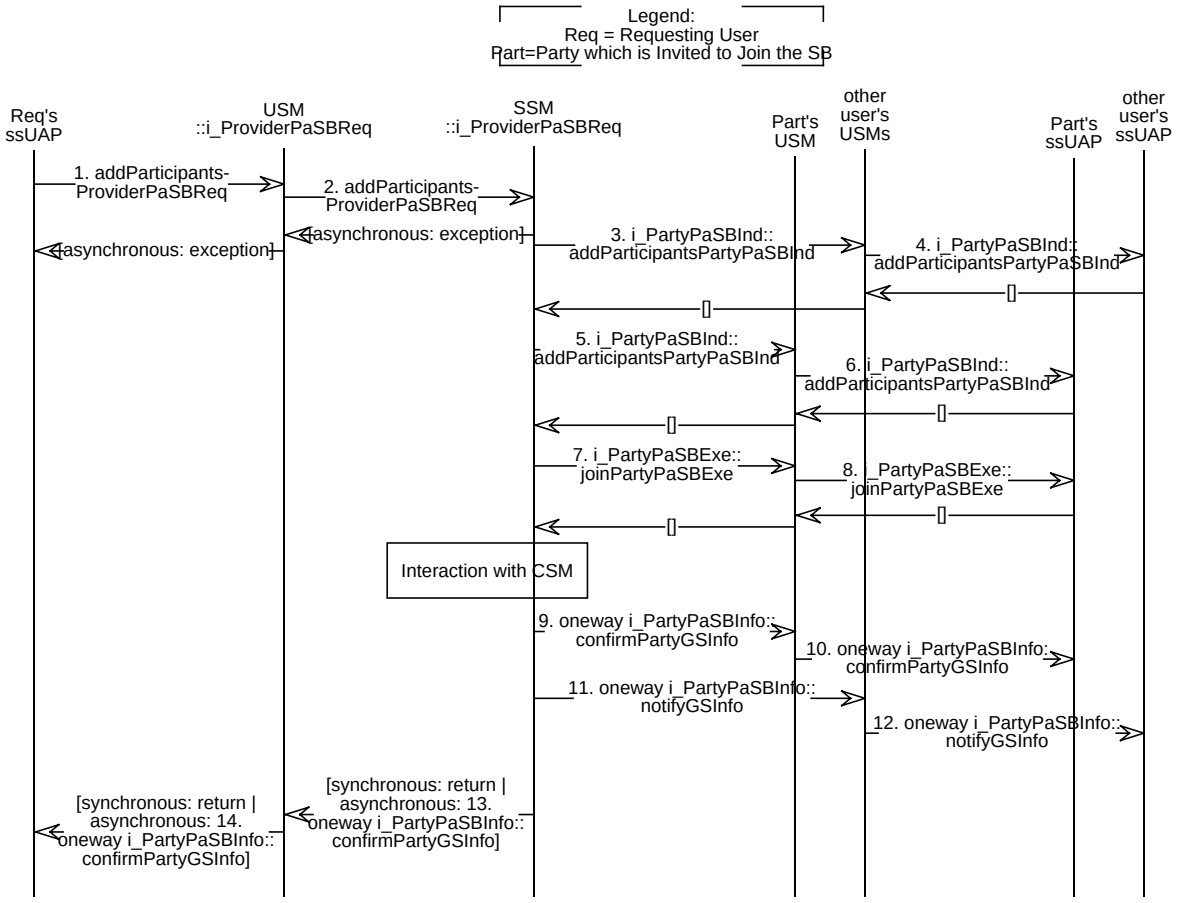


Figure 4-23. Add Participants to a Participant Oriented Stream Binding

1. ssUAP -> USM::i_ProviderPaSBReq::addParticipantsProviderPaSBReq()
   A party in the service session initiates the addition of new parties to the streambinding. The request operation includes a list of parties which need to be invited to join the streambinding. This list can include the requester himself, in which case a list of SFEPs for binding can be included in the operation.
2. USM -> SSM::i_ProviderPaSBReq::addParticipantsProviderPaSBReq()
   The request is forwarded to the SSM. If the request is processed asynchronously, an e_NoSynchronousReqResp exception is thrown, which is forwarded to the invoking ssUAP.
3. SSM -> USM::i_PartyPaSBInd::addParticipantsPartyPaSBInd()
   An addPartyPaSBInd is invoked on the USM of selected parties in the service session. These parties can be selected according to service-specific semantics or as a result of control session relationship settings in the TINA service session model.
4. USM -> ssUAP::i_PartyPaSBInd::addParticipantsPartyPaSBInd()
   The indication is forwarded to the related ssUAPs.
5. SSM -> USM::i_PartyPaSBInd::addParticipantsPartyPaSBInd()
   An addPartyPaSBInd is invoked on the USM of those parties which are being invited to join the streambinding.

6. USM -> ssUAP::i_PartyPaSBInd::addParticipantsPartyPaSBInd()
   The indication is forwarded to the related ssUAPs.
   The optional voting procedure takes place. If the voting results in acceptance, the scenario continues completely identical as in the scenario "Add Participant Oriented Stream Binding to a Service Session".

4.4.13 Delete Participants from a Participant Oriented Stream Binding

This scenario covers the procedures to delete Participants from a Participant Oriented Stream Binding. Both the synchronous and asynchronous procedures are covered.

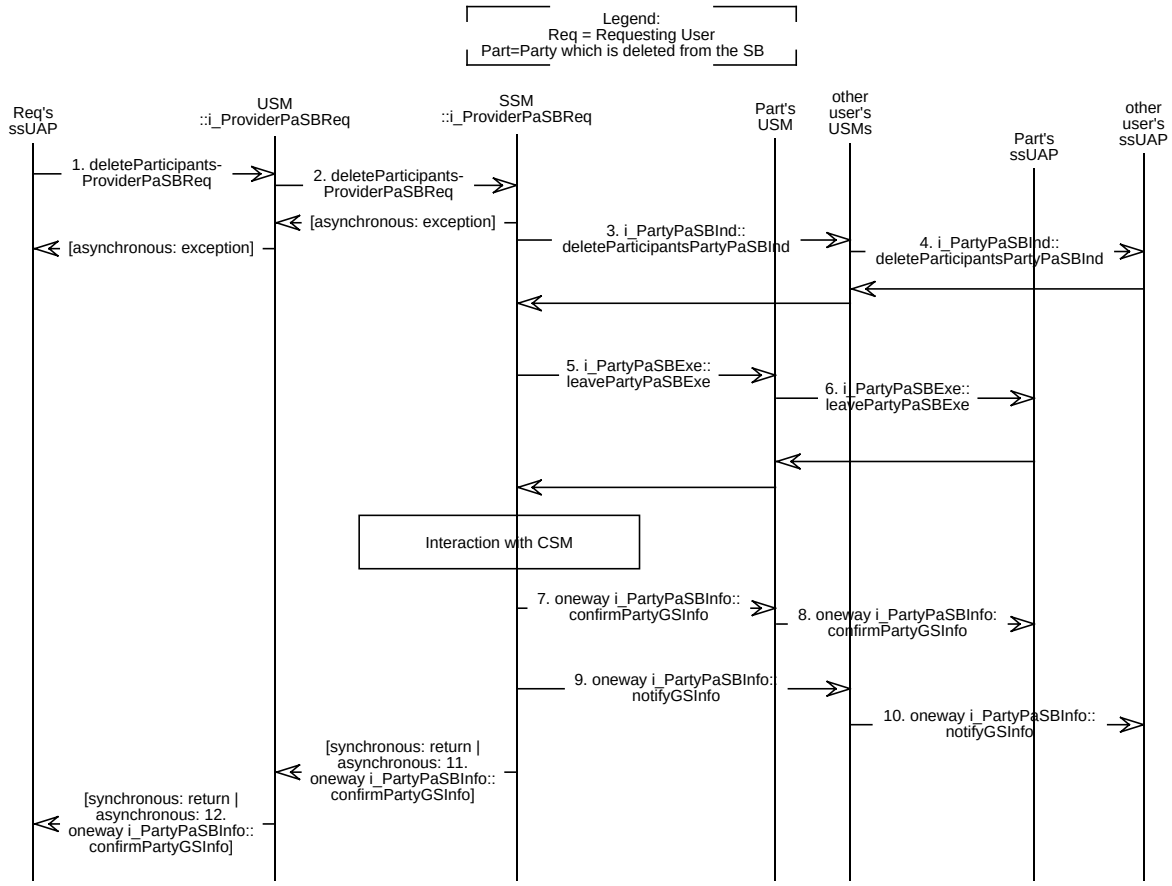


Figure 4-24. Delete Participants from a Participant Oriented Stream Binding

1. ssUAP -> USM::i_ProviderPaSBReq::deleteParticipantsProviderPaSBReq()
   A party in the service session initiates the deletion of parties from the streambinding. The list of parties that are to be removed can include the requester himself.
2. USM -> SSM::i_ProviderPaSBReq::deleteParticipantsProviderPaSBReq()
   The request is forwarded to the SSM. If the request is processed asynchronously, an e_NoSynchronousReqResp exception is thrown, which is forwarded to the invoking ssUAP.
3. SSM -> USM::i_PartyPaSBInd::deleteParticipantsPartyPaSBInd()
   A deleteParticipantsPartyPaSBInd is invoked on the USM of selected parties in the service session. These parties can be selected according to service-specific semantics or as a result of

control session relationship settings in the TINA service session model. The parties that are to be deleted can be among these selected parties, but this is not required.

4. USM -> ssUAP::i_PartyPaSBInd::deleteParticipantsPartyPaSBInd()
   The indication is forwarded to the related ssUAPs.
   The optional voting procedure takes place. If the voting results in acceptance, the SSM determines the effect of removing parties from the streambinding and interacts with the CSM to request the relevant modifications to the communication session. When the modifications are successful, the scenario continues:
5. SSM -> USM::i_PartyPaSBExe::leavePartyPaSBExe()
   A leavePartyPaSBExe is invoked on the USM of those parties which are being deleted from the streambinding.
6. USM -> ssUAP::i_PartyPaSBExe::leavePartyPaSBExe()
   The operation is forwarded to the related ssUAPs.
7. SSM -> USM::i_PartyPaSBInfo::confirmPartyGSInfo()
   A confirmPartyGSInfo is invoked on the USM of those parties which have been deleted from the streambinding,.
8. oneway USM -> ssUAP::i_PartyPaSBInfo::confirmPartyGSInfo()
   The info is forwarded to the related ssUAPs.
9. oneway SSM -> USM::i_PartyPaSBInfo::notifyGSInfo()
   A notifyGSInfo is invoked on the USM of selected parties (as in 3.) to inform them that the procedures to delete parties from the streambinding have finished successfully.
10. oneway USM -> ssUAP::i_PartyPaSBInfo::confirmPartyGSInfo()
    The info is forwarded to the related ssUAPs.
11. oneway SSM -> USM::i_PartyPaSBInfo::confirmPartyGSInfo()
    When the operation has been processed asynchronously, the requester is informed of the successful outcome by means of a confirmPartyGSInfo.
12. oneway USM -> ssUAP::i_PartyPaSBInfo::confirmPartyGSInfo()
    The info is forwarded to the requesting party's ssUAP.

4.4.14 Delete a Participant Oriented Stream Binding from the Service Session

This scenario covers the procedures to delete a Participant Oriented Stream Binding from the Service Session. Both the synchronous and asynchronous procedures are covered.

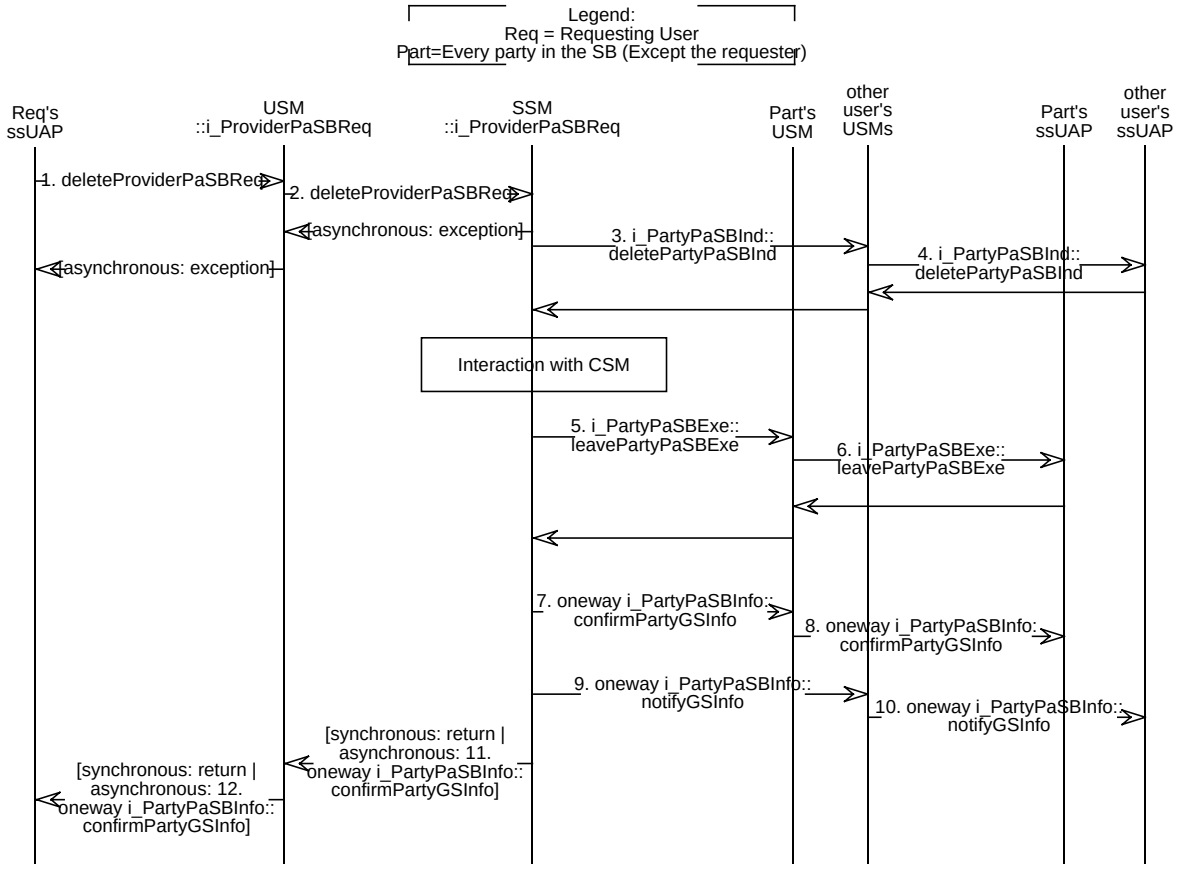


Figure 4-25. Delete a Participant Oriented Stream Binding

1. ssUAP -> USM::i_ProviderPaSBReq::deleteProviderPaSBReq()
   A party in the service session initiates the deletion of the streambinding.

2. USM -> SSM::i_ProviderPaSBReq::deleteProviderPaSBReq()
   The request is forwarded to the SSM. If the request is processed asynchronously, an e_NoSynchronousReqResp exception is thrown, which is forwarded to the invoking ssUAP.

3. SSM -> USM::i_PartyPaSBInd::deletePartyPaSBInd()
   A deletePartyPaSBInd is invoked on the USM of selected parties in the service session. These parties can be selected according to service-specific semantics or as a result of control session relationship settings in the TINA service session model. The parties that have previously joined the streambinding can be among these selected parties, but this is not required.

4. USM -> ssUAP::i_PartyPaSBInd::deletePartyPaSBInd()
   The indication is forwarded to the related ssUAPs. The optional voting procedure takes place. If the voting results in acceptance, the SSM interacts with the CSM to request the relevant modifications to the communication session. When the modifications are successful, the scenario continues exactly as in scenario 4.4.13.

4.4.15 Example of Voting Procedure

This scenario covers the processing of the voting procedure. Whether or not voting is required is defined by either service-specific semantics or as a result of control session relationship settings in the TINA service session model. (Here, “XXX” means Basic, ControlSR, and so on)

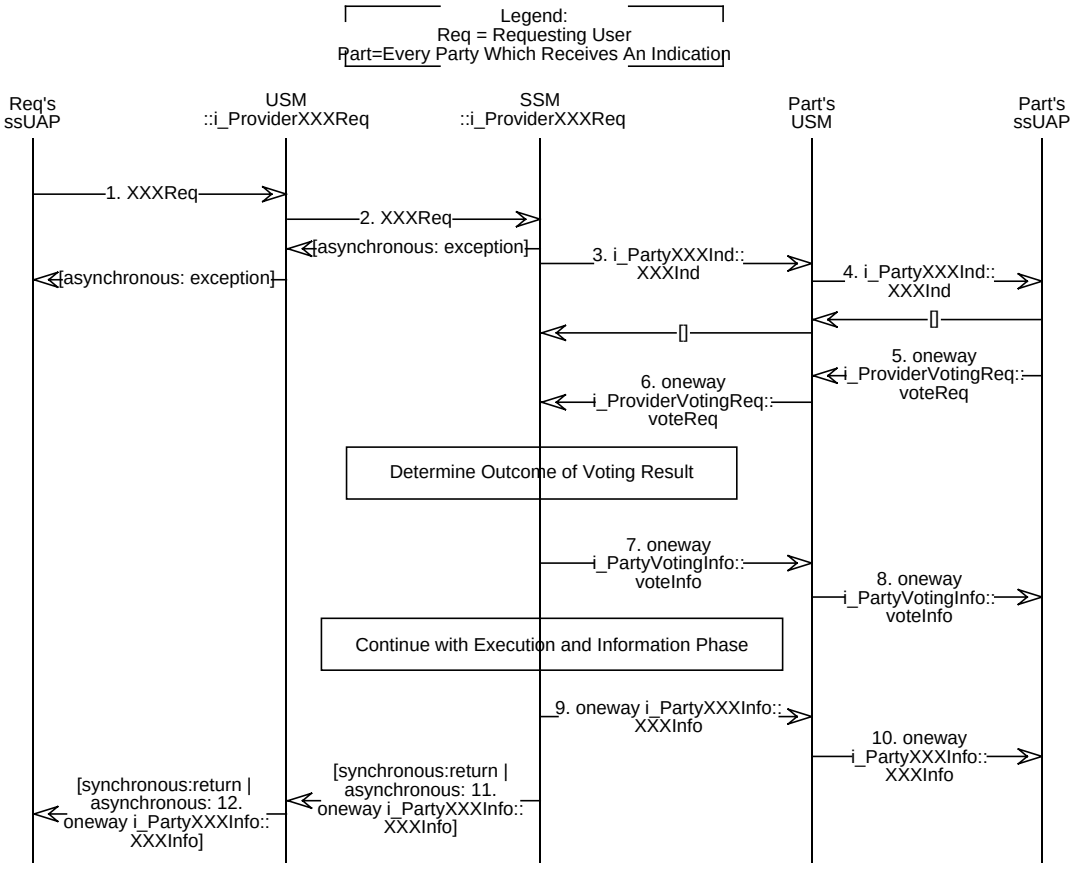


Figure 4-26. Usage of Voting Feature Set

1. ssUAP -> USM::i_ProviderXXXReq::XXXReq()
   A party in the service session initiates a request.
2. USM -> SSM::i_ProviderXXXReq::XXXReq()
   The request is forwarded to the SSM.
3. SSM -> USM::i_PartyXXXInd::XXXInd()
   An XXXInd is invoked on the USM of selected parties in the service session. The indication is identified by an "IndId". All parties which have voting right for the requested action have to reveive an indication. However, indication can also be invoked on additional parties. The parties which receive an indication can be selected according to service-specific semantics or as a result of control session relationship settings in the TINA service session model.
4. USM -> ssUAP::i_PartyXXXInd::XXXInd()
   The indication is forwarded to the related ssUAPs.
5. ssUAP -> USM::i_ProviderVotingReq::voteReq()
   The voting user issues a vote by invoking a voteReq.

6. USM -> SSM::i_ProviderVotingReq::voteReq()
   ... which is forwarded to the SSM. The voteReq contains the IndId to identify the indication it refers to. It is service-specific to determine until what time the SSM is ready to accept votes. It might be that it is required that the voteReq is invoked BEFORE the indication returns, but this requires multithreading in the SSM within the processing context of the request. In most cases, a timer will be started to determine the timeframe for receiving votes. It is again service-specific to determine the “default” vote result, i.e. to treat votes which are awaited but which are not received as either acceptance or failure.
   Determination of the outcome of the voting procedures is either service-specific or results from control session relationship settings in the TINA service session model.
7. oneway SSM -> USM::i_PartyVotingInfo::voteInfo()
   The result of the voting procedures can optionally be communicated to selected parties (as in 3.)
8. oneway USM -> ssUAP::i_PartyVotingInfo::voteInfo()
   The voteInfo is forwarded to the ssUAP.
   If the voting procedures result in acceptance, the processing of the action continues as if no voting has taken place.

4.4.16 Example of Control FS usage

This scenario covers the procedures to set a control session relationship.

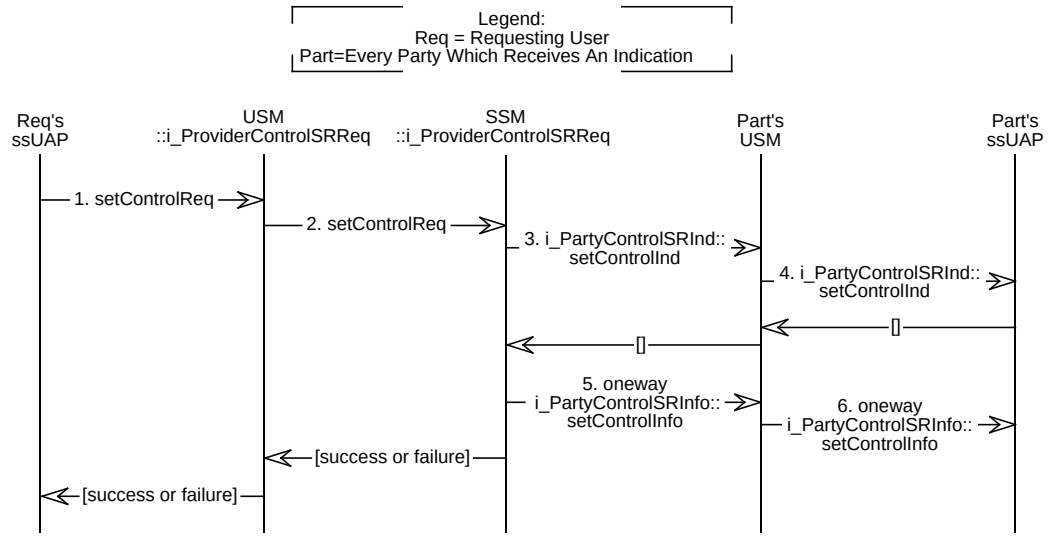


Figure 4-27. Set a Control Session Relationship

1. ssUAP -> USM::i_ProviderControlSRReq::setControlReq()
   A party in the service session initiates the setting of a control session relationship. If no explicit session relationship is created between a controlling party and a controlled object, a default is assumed to exist, whose value is determined either according to service-specific semantics or TINA service session defined default.
2. USM -> SSM::i_ProviderControlSRReq::setControlReq()
   The request is forwarded to the SSM.

3. SSM -> USM::i_PartyControlSRInd::setControlInd()
   A setControlInd is invoked on the USM of selected parties in the service session. These parties can be selected according to service-specific semantics or as a result of control session relationship settings in the TINA service session model.
4. USM -> ssUAP::i_PartyControlSRInd::setControlInd()
   The indication is forwarded to the related ssUAPs.
   The optional voting procedure takes place. If the voting results in acceptance, the scenario continues:
5. SSM -> USM::i_PartyControlSRInfo::setControlInfo()
   A setControlInfo is invoked on the USM of selected parties (as in 3.) to inform them that the procedures to add a control session relationship have finished successfully.
6. oneway USM -> ssUAP::i_PartyControlSRInfo::setControlInfo()
   The info is forwarded to the related ssUAPs.

4.4.17 Service Session Accounting

This scenario shows how the accounting events are pushed from lower levels up to the user agent that finally stores the information. Figure 4-28 provides a graphical representation of this scenario.

In this scenario, we assume a transparent billing context model, where the provider acts as a billing agent for the connectivity provider (CP). Although the flow of accounting events may differ, overall accounting management architecture and necessary component specifications are almost the same for the opaque billing context model.

Bare transport level traffic is measured, as they are specified in NRIM [6], which corresponds to the NFC under measurement. The accounting events are recorded, or collected using an event management ladder [7], 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 when their instances are created.

In WYSWYP (What You See is What You Pay) [13] 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 networks, where internet service provider (CP) is not concerned with per connection QoS or 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 user domain.

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 provider’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, both of which should be agreed at the beginning of the service transaction, as part of management context negotiation.

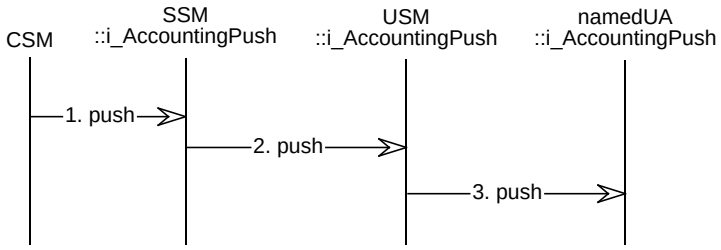


Figure 4-28. An example scenario of accounting management

1. CSM -> SSM::i_AccountingPush::push()
   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.
2. SSM -> USM::i_AccountingPush::push()
   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.
3. USM -> namedUA::i_AccountingPush::push()
   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.

After step 3, the accounting information is stored in the namedUA. When on-line billing is used, the PA can request accounting information from the UA (see Section 4.3.5). Provider's billing systems can also access the UA to get the user's accounting information to produce the user's bill.

4.5 Ancillary Usage Related Scenarios

In the scenarios for subscription management only some interactions are prescriptive:

- Interactions through Ret (between user application and USM/SSM_{ols}).
- Interactions with the User Agent.
- Interactions with the SLCM.

Interactions between Sub and USM/SSM_{ols} are not prescribed and are shown just for completeness.

These scenarios do not include the access phase and the start of a subscription management service session. This has been described in detail in the previous sections.

4.5.1 Subscribe a New Customer

This event trace describes how a new customer is subscribed to the provider, contracting a number of services, and how the service contract and subscriber information (associated end users and end user groups) are defined.

The customer is not previously subscribed to the provider. The scenario “Contract a New Service” on page 118 describes how to contract new services for a customer that is already subscribed to the provider.

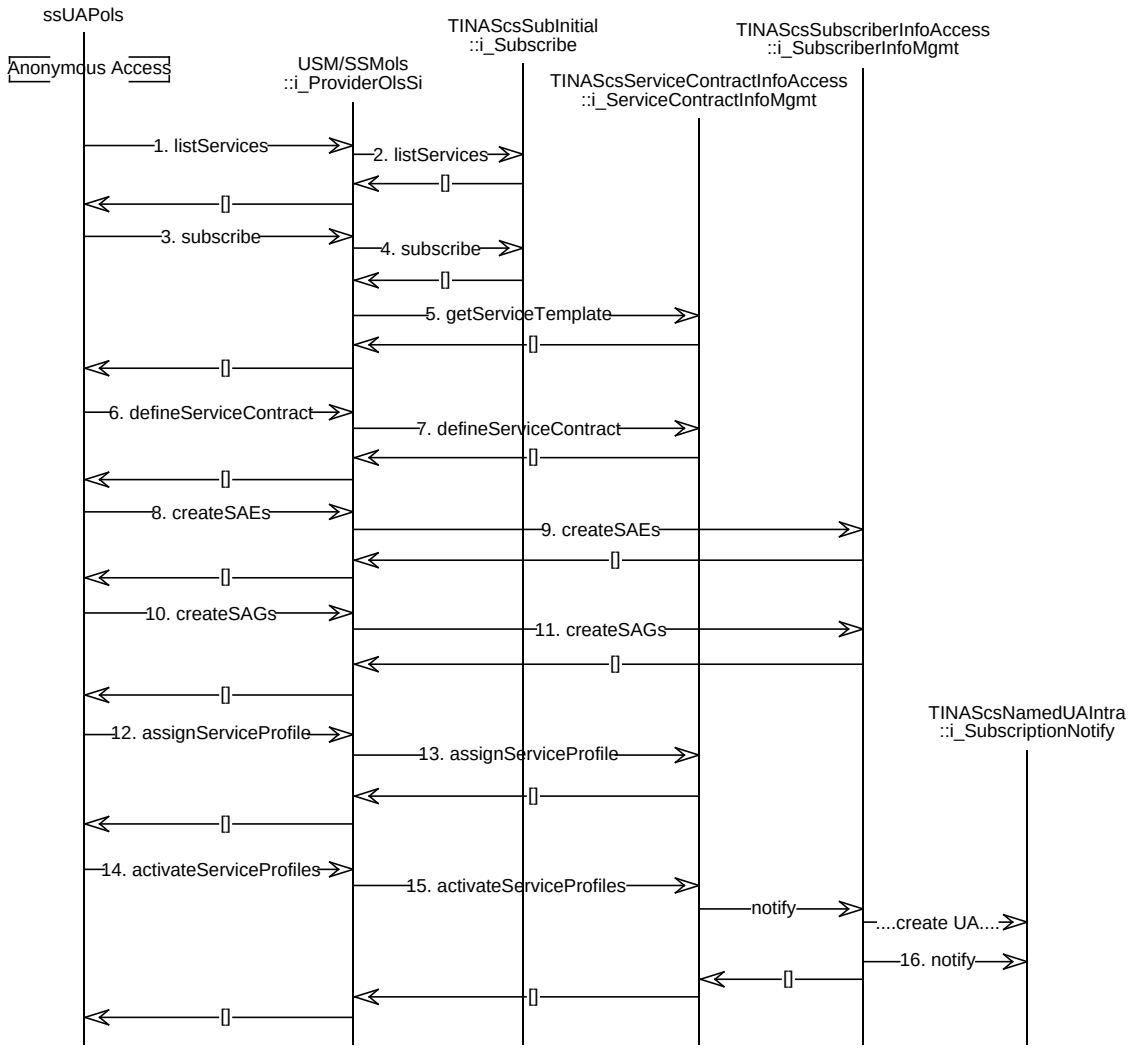


Figure 4-29. Subscribe a new customer.

1. $ssUAP_{ols}$ -> USM/SSM_{ols}::i_ProviderOlsSi::listServices()
   The $ssUAP_{ols}$ requests the list of services to the session manager through Ret-RP.
2. USM/SSM_{ols} -> Sub::i_Subscribe::listServices()
   The USM/SSM_{ols} requests the list of services. Sub returns the list of services provided by the provider. USM/SSM_{ols} returns it on its turn to the user application.
3. $ssUAP_{ols}$ -> USM/SSM_{ols}::i_ProviderOlsSi::subscribe()
   The $ssUAP_{ols}$ requests the creation of a new subscriber. The required subscriber information (for instance, identification and billing information) and the list of services the new customer wants to subscribe to are the main parameters in this operation. The subscriber identifier is returned as a result. The list of service templates for each of the requested services is returned as well.

4. USM/SSM$_{ols}$ -> Sub::i_Subscribe::subscribe()
   The USM/SSM$_{ols}$ requests to Sub the creation of a new subscriber. The subscriber information and the list of services received from the user application are passed as parameters. The new subscriber identifier is returned as a result. Some interfaces references are also returned, the subscriber information management interface (`i_SubscriberInfoMgmt`) and a set of interfaces for service contract definition (`i_ServiceContractInfoMgmt`), one for each requested service. Optionally[6], a service contract identifier can be returned for every requested service.

5. USM/SSM$_{ols}$ -> Sub::i_ServiceContractInfoMgmt::getServiceTemplate()
   For every service, the USM/SSM$_{ols}$ requests a service template using the corresponding service contract management interface.

   At this moment, the reply to the subscriber request is sent to the user application. The subscriber identifier and the list of service templates for each of the requested services are returned in the reply. The ssUAP$_{ols}$ will use these service templates to derive the service profiles that will define the service contracts.

6. ssUAP$_{ols}$ -> USM/SSM$_{ols}$::i_ProviderOlsSi::defineServiceContract()
   The `defineServiceContract` operation is used by the ssUAP$_{ols}$ to specify the service contract. The Subscription Profile and a list of SAG Service Profiles are the main components of the service contract.

7. USM/SSM$_{ols}$ -> Sub::i_ServiceContractInfoMgmt::defineServiceContract()
   The USM/SSM$_{ols}$ uses an equivalent operation on Sub to pass it the contract information. A list of SAG Service Profile identifiers is got as a return value. The list of identifiers is passed back to the ssUAP$_{ols}$.

8. ssUAP$_{ols}$ -> USM/SSM$_{ols}$::i_ProviderOlsSi::createSAEs()
   All the subscription assignment entities (users, terminals or NAPs) in the subscriber domain that could make use of the contracted services are defined using the `createSAEs` operation on the online subscription service specific interface. The user application can propose identifiers for the users it is defining.

9. USM/SSM$_{ols}$ -> Sub::i_SubscriberInfoMgmt::createSAEs()
   These entities are passed to Sub using the equivalent `createSAEs` operation on the subscriber information management interface. The subscriber Identifier can be required if the `i_SubscriberInfoMgmt` is not unique per subscriber. Sub might create at this moment the Access Agent for that entity[7], a namedUA in case of a user.

   The list of assigned entity identifiers is the returned value. These identifiers are unique in the provider domain[8].

   The list is passed back to the user application.

10. ssUAP$_{ols}$ -> USM/SSM$_{ols::}$i_ProviderOlsSi::createSAGs()
    Then, the subscriber, through the user application, creates a number of SAGs. The input values for the creation operation is a list of pairs <list of SAEs, proposed SAG identifier>.

---

6. In implementations that provides only one service contract management interface per service. This interface is shared by all the service contracts.
7. The creation can be deferred to the moment in which this SAE is assigned to a service profile and, thus, has access to some service.
8. The identifier might be a composition of user identifier (the one passed as an input parameter) and the subscriber identifier. For instance, the invocation of `createSAEs(..Juan...)` from subscriber `TinaC` could receive as a return value `Juan_TinaC`. Any other way of constructing identifiers is valid as well, as long as it guarantees the uniqueness of the user identifier in the providler domain.

11. USM/SSM$_{ols}$ -> Sub::i_SubscriberInfoMgmt::createSAGs()
    The creation operation is forwarded to the Sub. This invocation may require the Subscriber Id (in implementations with just one shared subscription information management interface).

    The list of SAG identifiers is returned. These identifiers are unique in the provider domain.

12. ssUAP$_{ols}$ -> USM/SSM$_{ols::}$i_ProviderOlsSi::assignServiceProfile()
    The subscriber, through the user application, can assign a service profile to a number of SAEs and SAGs. The USM/SSM$_{ols}$ may check whether the service profile is part of the subscriber's service contract or not and whether the SAEs and SAGs belong to the subscriber or not, before proceeding with the next step.

13. USM/SSM$_{ols}$ -> Sub::i_ServiceContractInfoMgmt::assignServiceProfile()
    The assignment operation is forwarded to the Sub. When this operation returns, the SAEs (the ones explicitly stated and the ones composing the SAGs) will have a service profile describing the characteristics of the service they will receive from the provider, but they will not be able to use the service unless the service profile is made active.

14. ssUAP$_{ols}$ -> USM/SSM$_{ols::}$i_ProviderOlsSi::activateServiceProfiles()
    This is the final step in the subscription process. The ssUAPols requests the activation of a number of service profiles. The USM/SSM$_{ols}$ may check whether the service profile is part of the subscriber's service contract or not, before proceeding with the next step.

15. USM/SSM$_{ols}$ -> Sub::i_ServiceContractInfoMgmt::activateServiceProfiles()
    The activation operation is forwarded to the Sub. Internally, Sub can notify the subscriber manager component about this activation, so that they can notify the namedUA about the new available subscribed service. Depending on the implementor choice, this might trigger the creation of a namedUA in case this is the first service (profile) assigned to the user.

16. USM/SSM$_{ols}$ -> namedUA::i_SubscriptionNotify::notify()
    Sub notifies the assigned users' namedUA of the new available service. The service identifier and the service profile for the group (SAG) the user has been assigned to are the relevant parameters in the notification.

    The activation operation returns at this moment.

The customer is subscribed to the provider. From then on, the defined end-users can make use of the services they have been granted using the service profile that has been assigned for them or for the group (SAG) they belong to. An interface through which these users' namedUAs can retrieve this information (service profiles and subscribed services) is created in Sub.

4.5.2 Modify Subscriber Information

This scenario describes how the subscriber information is modified. The main aspects that can be modified are SAGs, SAEs and assignment of SAEs to SAGs. This event trace shows first how new SAEs are defined and assigned to existing SAGs, then how existing SAEs are removed from a SAG and finally how some are deleted.

The subscriber is already subscribed to the provider and has contracted some services. A number of SAGs and SAEs have been previously defined for the subscriber.

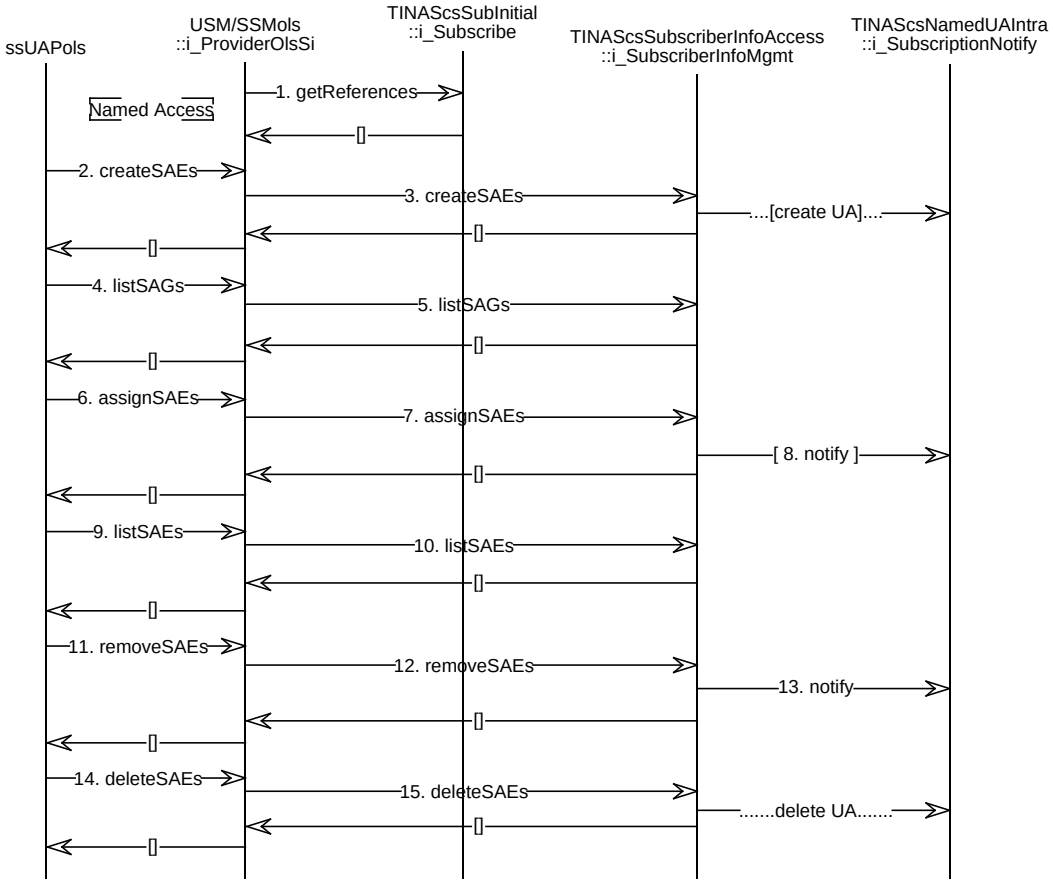


Figure 4-30. Modify Subscriber Information.

Definition of new SAEs:

1. USM/SSM_{ols} -> Sub::i_Subscribe::getReferences()
   The USM/SSM_{ols} gets the interface references for subscription information management (`i_SubscriberInfoMgmt`). This operation is optional and needs not to be performed in case the USM/SSM_{ols} holds the interface references from previous interactions with Sub. It is shown just for completeness. The subscriber identifier will be required in the invocation if the particular Sub implementation offers a different management interface for every subscriber.

2. $ssUAP_{ols}$ -> USM/SSM_{ols}::i_ProviderOlsSi::createSAEs()
   The subscriber defines some new entities (SAEs) using the online subscription service specific interface (`i_ProviderOlsSi`). An identifier for each of them may be proposed.

3. USM/SSM_{ols} -> Sub::i_SubscriberInfoMgmt::createSAEs()

   This request is forwarded to Sub. This could result in the creation of an access agent (namedUA) for these entities (this is an implementor's choice). Another option is to delay the namedUA creation until a service is made available to the user.

   An identifier for each SAE is got as a result and passed back to the user application. These identifiers are unique in the provider domain.

Now the new SAEs are assigned to an existing SAG:

4. $ssUAP_{ols}$ -> USM/SSM_{ols}::i_ProviderOlsSi::listSAGs()
   The $ssUAP_{ols}$ asks for the list of SAGs defined for the subscriber.

5. USM/SSM_{ols} -> Sub::i_SubscriberInfoMgmt::listSAGs()
   USM/SSM_{ols} passes this query to the Sub. The Subscriber identifier may be required in the invocation to determine the subscriber, in case of Sub implementations in which the subscriber information management interface is shared by more than one subscriber. A list of SAG identifiers is returned and passed back to the user application.

6. $ssUAP_{ols}$ -> USM/SSM_{ols}::i_ProviderOlsSi::assignSAEs()
   A list of SAEs is assigned to an existing SAG.

7. USM/SSM_{ols} -> Sub::i_SubscriberInfoMgmt::assignSAEs()
   The request is passed to Sub. Again, the subscriber id may be required if the interface is shared by more than one subscriber.

8. Sub -> namedUA::i_SubscriptionNotify::notify()
   This operation is not always required. If the SAG has a service profile associated, this will be available for the assigned entities and, thus, a notification should be done to the corresponding agents. Sub notifies each assigned entity's namedUA about the new available service, indicating the service identifier and the corresponding service profile.

Removal of SAEs from a SAG:

9. $ssUAP_{ols}$ -> USM/SSM_{ols}::i_ProviderOlsSi::listSAEs()
   $ssUAP_{ols}$ asks for the list of SAEs assigned to a specific SAG. The SAG identifier is one of the input parameters.

10. USM/SSM_{ols} -> Sub::i_SubscriberInfoMgmt::listSAEs()
    USM/SSM_{ols} asks the Sub for this list. A list of entity identifiers is returned and passed back to the $ssUAP_{ols}$.

11. $ssUAP_{ols}$ -> USM/SSM_{ols}::i_ProviderOlsSi::removeSAEs()
    The subscriber, through the user application, requests the removal of a list of entities from a group (SAG).

12. USM/SSM_{ols} -> Sub::i_SubscriberInfoMgmt::removeSAEs()
    USM/SSM_{ols} passes this request to Sub. Sub removes the entities from the SAG, disabling the usage of the service profiles assigned to the SAG for the removed entities.

13. Sub -> namedUA::i_SubscriptionNotify::notify()
    Sub notifies the corresponding namedUAs about the service withdrawn. In case the user has no other service assigned, the namedUA may be deleted (this is an implementors choice).

Deletion of SAEs:

14. $ssUAP_{ols}$ -> USM/SSM_{ols}::i_ProviderOlsSi::deleteSAEs()
    With this operation, the $ssUAP_{ols}$ requests the deletion of a number of entities.

15. USM/SSM_{ols} -> Sub::i_SubscriberInfoMgmt::deleteSAEs()
    The USM/SSM_{ols} forwards this operation to the Sub. This derives in the removal of the entities from any SAG they could be assigned to and in the deletion of the corresponding namedUA.

The subscriber information is updated in the subscriber database and is available for any external client (with the required authorization). The namedUAs have been notified of all the changes in the subscriber information that affects its represented user. The namedUA has performed the required actions to keep the consistency between modified service profiles and possible customisations performed by the user. Some namedUAs may have been deleted as a result of these modifications, either because of the deletion of an entity or because the corresponding entity is no longer assigned to any service.

4.5.3 Contract a New Service

This scenario shows how a new service contract for a subscriber can be defined. The scenario “Subscribe a New Customer” on page 112 describes how to contract services for a customer that is not subscribed to the provider yet.

The subscriber is already subscribed to the provider. Some entities (users, terminals or NAPs) have been already defined for that subscriber.

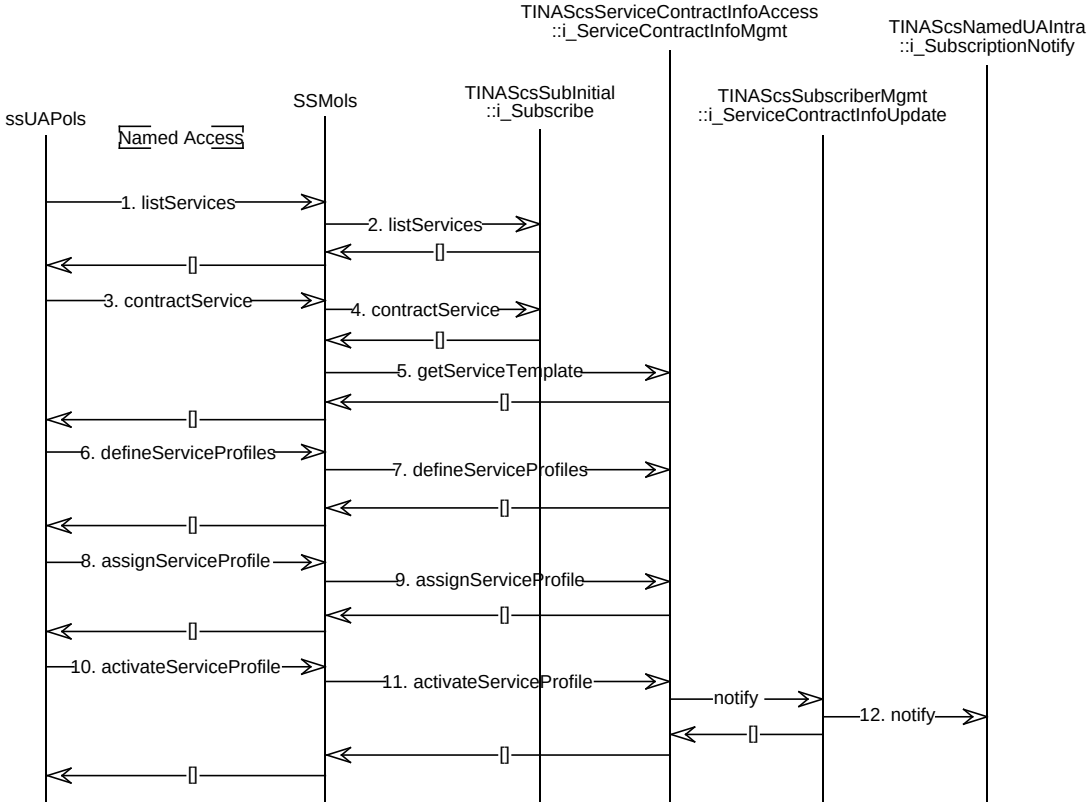


Figure 4-31. Contract a new service.

1. $ssUAP_{ols}$ -> USM/SSM_{ols}::i_ProviderOlsSi::listServices()
   The subscriber, via the $ssUAP_{ols}$, asks for the list of services he is subscribed to. This operation is optional and need not to be performed in case the $ssUAP_{ols}$ keeps this information. It is shown just for completeness.
2. USM/SSM_{ols} -> Sub::i_Subscribe::listServices()
   The USM/SSM_{ols} asks for this list of services to the Sub. This operation is optional for the same reason above mentioned.
3. $ssUAP_{ols}$ -> USM/SSM_{ols}::i_ProviderOlsSi::contractService()
   The $ssUAP_{ols}$ requests the contract of a list of services for the subscriber.
4. USM/SSM_{ols} -> Sub::i_Subscribe::contractService()
   The USM/SSM_{ols} requests the contract of a list of services for a specific subscriber. A list of service contract management interfaces is returned, one per requested service.

5. USM/SSM_{ols} -> Sub::i_ServiceContractInfoMgmt::getServiceTemplate()
   The USM/SSM_{ols} uses the interfaces received in the previous step to query for the corresponding service templates and returns them back to the user application as a reply to the contract request.. These templates are used by the $ssUAP_{ols}$ to derive the corresponding subscription and SAG service profiles.

6. $ssUAP_{ols}$ -> USM/SSM_{ols}::i_ProviderOlsSi::defineServiceProfiles()
   The user application requests the creation of the service profiles for that subscriber. The set of profiles includes a generic profile for the subscriber (applicable to all users) and a number of SAG service profiles, that must be consistent with the previous one.

7. USM/SSM_{ols} -> Sub::i_ServiceContractInfoMgmt::defineServiceProfiles()
   USM/SSM_{ols} requests the creation of these service profiles to the Sub. The subscriber identifier may be required in the invocation if the service contract management interface is shared by more than one service contract.

   A list of SAG service profiles is returned as a reply and passed back to the $ssUAP_{ols}$.

8. $ssUAP_{ols}$ -> USM/SSM_{ols}::i_ProviderOlsSi::assignServiceProfile()
   Then, the service profiles can be assigned to a number of SAGs and SAEs. The input values for this operation are the list of SAGs and SAEs and the identifier of the SAG service profile that is assigned to them.

9. USM/SSM_{ols} -> Sub::i_ServiceContractInfoMgmt::assignServiceProfile()
   USM/SSM_{ols} passes this request to the Sub that performs the assignment.

10. $ssUAP_{ols}$ -> USM/SSM_{ols}::i_ProviderOlsSi::activateServiceProfile()
    The last step is the service profile activation. The user application requests this activation through the subscription service specific interface.

11. USM/SSM_{ols} -> Sub::i_ServiceContractInfoMgmt::activateServiceProfile (Service Profile id)
    The USM/SSM_{ols} passes this activation request to the Sub that performs the required actions. The service profile is then ready for use. The end-users can make use of it.

12. Sub -> namedUA::i_SubscriptionNotify::notify()
    Sub notifies the assigned users' naemdUA of the new available service. The service identifier and the service profile for the group (SAG) the user has been assigned to are the relevant parameters in the notification.

The new service is available for the assigned users, terminals or NAPs, using the service profile that has been assigned to them. An interface through which these users' namedUAs can retrieve this information (service profiles and subscribed services) may be created in Sub at this point.

4.5.4 Modify Service Contract

This trace shows the modification of a service contract. The service profile for an already contracted service is modified and a new SAG (and its corresponding service profile) is defined.

The subscriber is subscribed to the provider. A service contract has been already signed.

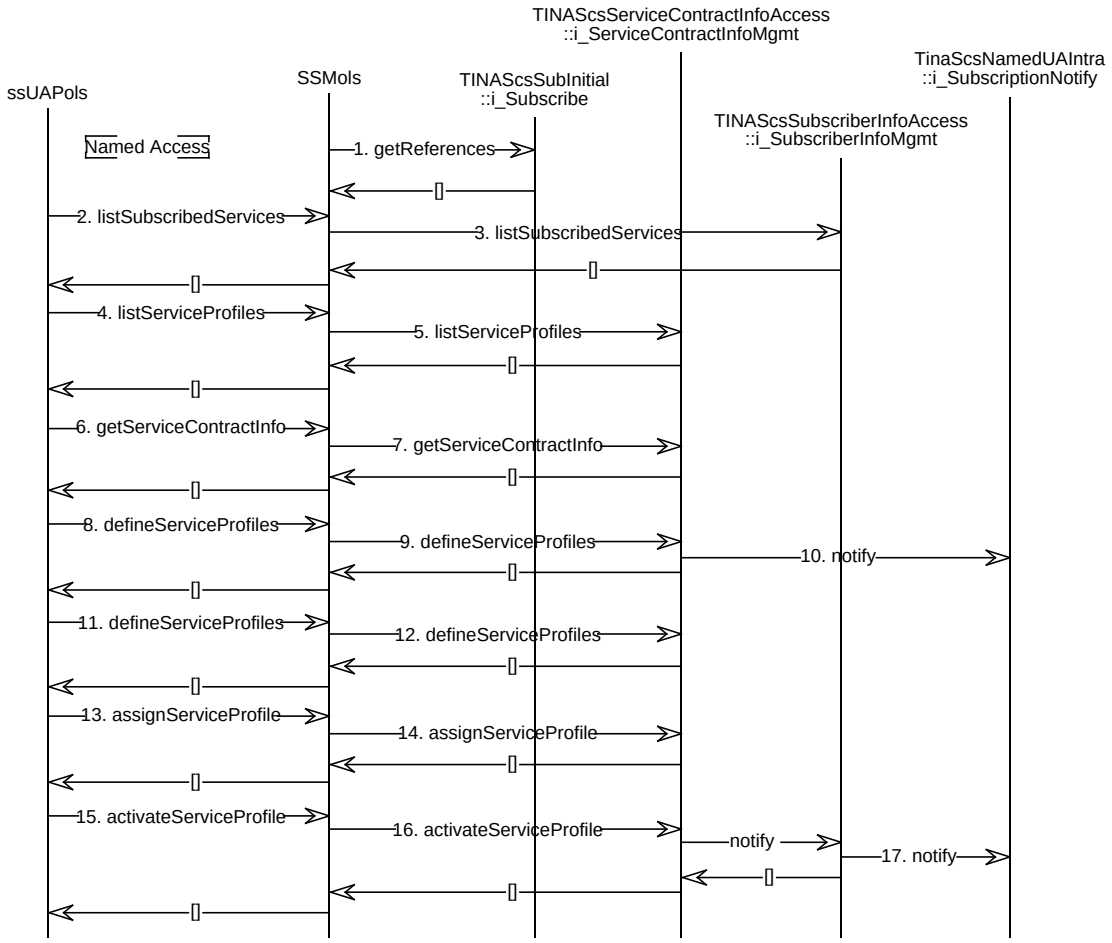


Figure 4-32. Modify a service contract.

1. USM/SSM_{ols} -> Sub::i_Subscribe::getReferences()
   The USM/SSM_{ols} gets the reference of the subscriber information management interface. If a service is specified, this operation returns the service contract information management interface for that service (and that subscriber, in case there is one interface per subscriber -service contract-). This operation and the third one are optional and need not to be performed in case the USM/SSM_{ols} keeps this information. It is shown just for completeness.

2. $ssUAP_{ols}$ -> USM/SSM_{ols}::i_ProviderOlsSi::listSubscribedServices()
   The subscriber, via the $ssUAP_{ols}$, asks for the list of services he(she) has contracted.

3. USM/SSM_{ols} -> Sub::i_SubscriberInfoMgmt::listSubscribedServices()
   The USM/SSM_{ols} gets the list of services a subscriber is subscribed to and passes it back to the $ssUAP_{ols}$. The subscriber identifier may be required in implementations in which the subscriber information management interface is shared by more than one subscriber. This

remark is applicable to most of the operations on the subscriber and service contract information management interfaces.

4. $ssUAP_{ols}$ -> USM/SSM_{ols}::i_ProviderOlsSi::listServiceProfiles()
   The $ssUAP_{ols}$ asks for the list of service profiles defined for a specific service.

5. USM/SSM_{ols} -> Sub::i_ServiceContractInfoMgmt::listServiceProfiles()
   USM/SSM_{ols} forwards the request to Sub. USM/SSM_{ols} receives a list of service profile identifiers that is passed back to the user application.

6. $ssUAP_{ols}$ -> USM/SSM_{ols}::i_ProviderOlsSi::getServiceContractInfo()
   These $ssUAP_{ols}$ requests a number of profiles, specifying their identifiers.

7. USM/SSM_{ols} -> Sub::i_ServiceContractInfoMgmt::getServiceContractInfo()
   USM/SSM_{ols} retrieves the service profiles corresponding to the specified identifiers. The profiles, and the rest of service contract information, are returned to the $ssUAP_{ols}$.

8. $ssUAP_{ols}$ -> USM/SSM_{ols}::i_ProviderOlsSi::defineServiceProfiles()
   The $ssUAP_{ols}$ requests the modification of the service profile.

9. USM/SSM_{ols} -> Sub::i_ServiceContractInfoMgmt::defineServiceProfiles()
   The USM/SSM_{ols} passes the request to Sub.

10. Sub -> namedUA::i_SubscriptionNotify::notify()
    Sub notifies the namedUA of each entity assigned to the modified service profile about the modification. The namedUA checks the consistency between the modified service profile and the possible customisations performed by the user and performs the required actions. For instance, it may reject those customisations not allowed in the new profile and apply the rest to it.

11. $ssUAP_{ols}$ -> USM/SSM_{ols}::i_ProviderOlsSi::defineServiceProfiles()
    The $ssUAP_{ols}$ defines a new service profile.

12. USM/SSM_{ols} -> Sub::i_ServiceContractInfoMgmt::defineServiceProfiles()
    The USM/SSM_{ols} passes the request to Sub. A new service profile identifiers is received and passed back to the user application.

Steps 13 to 17 are described in Section 4.5.3 (steps 8 to 12).

The service contract has been modified. From then on, the users that have been assigned to the new profile can make use of the service and the users assigned to the modified profile will use the service according to the new definition.

4.5.5 Unsubscribe

This scenario describes the procedure for cancelling a service contract or the whole subscription to a provider.

The subscriber is subscribed to the provider and has contracted a number of services. This trace covers the partial and total withdrawal of a subscription.

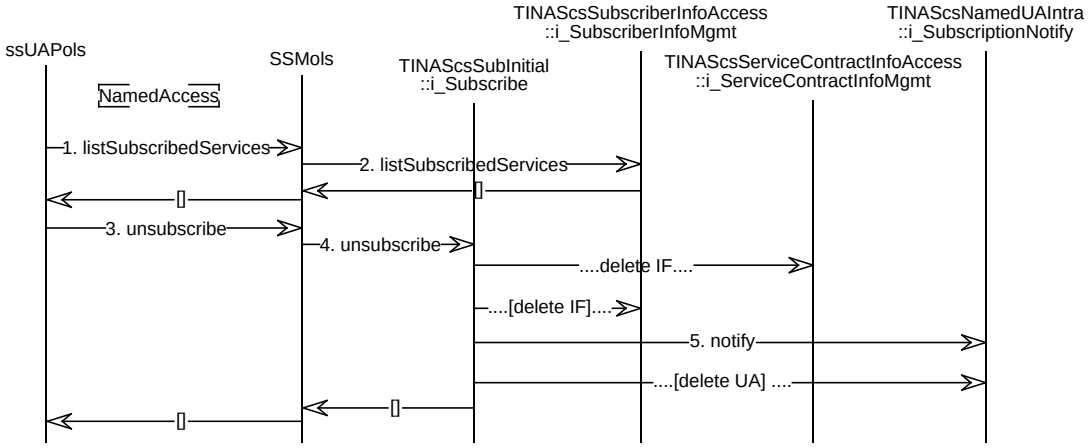


Figure 4-33. Total or partial cancellation of a subscription.

1. $ssUAP_{ols}$ -> USM/SSM_{ols}::i_ProviderOlsSi::listSubscribedServices()
   The $ssUAP_{ols}$ asks for the list of services the subscriber is subscribed to. This operation needs to be performed only in case the $ssUAP_{ols}$ does not keep track of this information.
2. USM/SSM_{ols} -> Sub::i_SubscriberInfoMgmt::listSubscribedServices()
   The USM/SSM_{ols} forwards the operation to Sub. This operation needs to be performed only in case the USM/SSM_{ols} does not keep track of this information and may require a previous retrieval of the subscriber information management interface (see `i_Subscriber::getReferences` operation usage in Section 4.5.2).
3. $ssUAP_{ols}$ -> USM/SSM_{ols}::i_ProviderOlsSi::unsubscribe()
   The $ssUAP_{ols}$ requests the cancellation of the contracts for a list of services.
4. USM/SSM_{ols} -> Sub::i_Subscribe::unsubscribe()
   The USM/SSM_{ols} requests the cancellation of the list of service contracts. All the specified contracts are removed, together with the interface to access them. In case it requests the withdrawal of all the contracted services (this can be indicated as an empty list or as a list including all the services), the subscriber, its management interface and all the corresponding namedUAs are removed.
5. Sub -> namedUA::i_SubscriptionNotify::notify()
   In case the cancellation affects only to a part of the contracted services, Sub notifies the affected users (namedUAs) of those services withdrawal.

No user associated to that subscriber can make use of the service(s) whose service contract has(ve) been cancelled. The service contract management interfaces related to the cancelled contracts are no longer available.

Some namedUAs associated to the subscriber may be deleted in case no service is available for the represented user after the `unsubscribe` operation. The query interfaces in Sub (`i_SubscriberInfoQuery`) are also removed.

If all the subscribed services are cancelled, the customer is removed from the subscribers list in the provider domain, and the subscriber management interfaces and all the associated namedUAs are deleted.

4.5.6 Register to receive invitations outside of an access session

This scenario allows a user to register with a provider for service invitations to be sent to a particular user context. The registration will result in invitations to join a service to arrive at a specific invitation interface when there is no access session for the user. This is an important part of personal mobility.

It is assumed that the user has established an access session, within which to make this request.

The user requests that invitations are sent to a particular user context. A user context represents a specific Provider Agent, which usually equates to a particular terminal. When a user establishes an access session, they can request that the user context for this access session is ‘saved’ by the UA. (The user chooses the name of the user context, so they can recognize and select them as part of this scenario.)

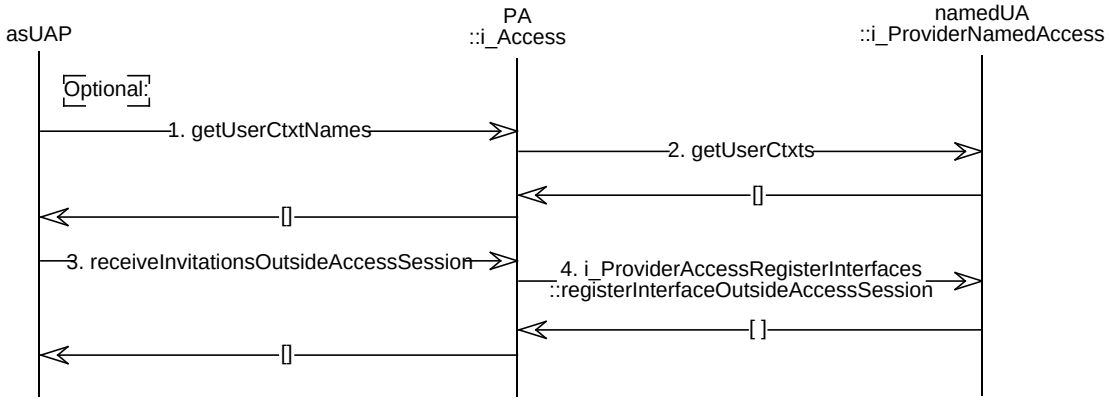


Figure 4-34. Register to receive invitations outside of an access session.

1. asUAP -> PA::i_Access::getUserCtxtNames().
   The user uses the as-UAP to get a list of the user contexts that are ‘saved’ within the UA, in order to select one to receive invitations outside of the access session.

2. PA -> namedUA::i_ProviderNamedAccess::getUserCtxts().
   The PA requests a list of the user contexts known by the UA. This list will include those user contexts ‘saved’ by the UA, as well as any other user contexts from current access sessions with the user.

   (Other operations could also be invoked to retrieve this information for a specific user context: getUserCtxt(), or based on the user’s access sessions: getUserCtxtsAccessSessions()).

   The UA returns the list of user contexts. Each context contains information about the terminal capabilities and user domain interfaces, including an invitation interface.

   The PA returns a list of the user context names to the as-UAP.

3. asUAP -> PA::i_Access::receiveInvitationsOutsideAccessSession().
   The user uses the as-UAP to select a user context at which they wish to receive invitations outside of this access session. The as-UAP invokes this operation to identify the name of the user context.

4. PA -> namedUA::i_ProviderAccessRegisterInterfaces::registerInterfaceOutsideAccessSession().
   The PA retrieves the i_UserInitial interface from the chosen user context. It uses this operation to register this interface with the namedUA. The namedUA will send invitations to this interface after this access session ends.

   This operation returns an interface index number that can be used to unregister the interface (unregisterInterface()). The interface can be unregistered within any access session.

4.5.7 Register a new service

This scenario describes the procedure for deploying a new service in the service network.

The service instance is not previously deployed. Steps 1 to 3 are optional. They are shown for completeness and represent the case in which the service type the service instance corresponds to is not already defined and the service network over which the service instance is to be deployed is not configured.

A set of different management applications may make use of the SLCM. In the trace, these are represented as one management application (managementAP).

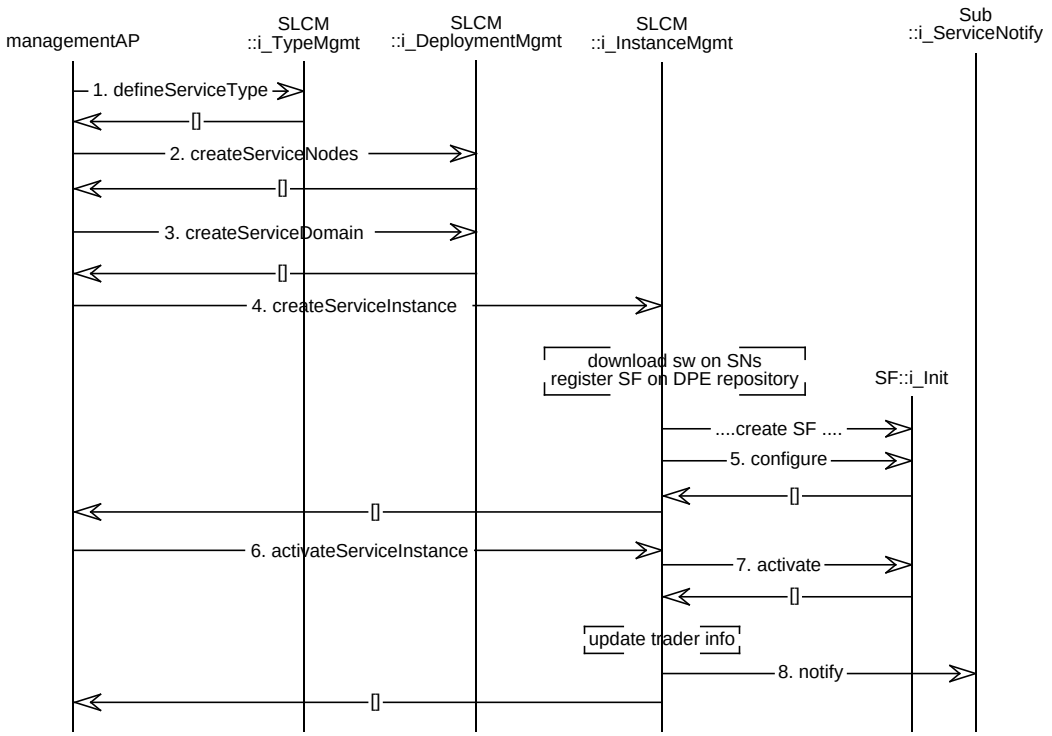


Figure 4-35. Register a new service.

1. managementAP -> SLCM::i_TypeMgmt::defineServiceType()
   The management application or a client external to the provider domain (for instance, an

information distribution application from an industrial forum or provider consortium) may define the service description that a provider's service instance must be consistent with in order to be considered as corresponding to a particular service type.

2. managementAP -> SLCM::i_DeploymentMgmt::createServiceNodes()
   The service network administrator, through a management application that can be different from the previous one, describes the service nodes composing the service network. A set of Node Manager[9] interface references, one for each node, are passed as a parameter. A list of service node identifiers is returned. These identifiers may be generated by the SLCM, in case they have not been specified in the call.

3. managementAP -> SLCM::i_DeploymentMgmt::createServiceDomain()
   The service network administration application defines a service domain for a specific service. In this operation the set of service nodes composing the domain, the service provided and the deployment and configuration policies are defined.

4. managementAP -> SLCM::i_InstanceMgmt::createServiceInstance()
   The management application (maybe different from the one we referred to in previous steps) requests the deployment of a new service instance. The identifier of the new instance and a set of service instance information parameters are specified in the call. These parameters will allow the SLCM to determine how the service instance needs to be deployed over the service network.

   Based on this information, the SLCM will transfer the service software (SF, SSM, USM and other specific service components) to the appropriate service nodes and update each node DPE repositories accordingly.

   The Service Factories are started.

5. SLCM -> SF::i_Init::configure()
   SLCM transfers the required configuration information to the SF.

6. managementAP -> SLCM::i_InstanceMgmt::activateServiceInstance()
   Once the service instance is deployed, the management application (maybe different from the one we referred to in previous steps) requests its activation. The identifier of the new instance is the main input parameter.

7. SLCM -> SF::i_Init::activate()
   SLCM changes the SF state to `active`, so that it can accept requests.

   Then, it updates the information in the trader or location DPE services used to retrieve to the service factory interfaces. This action will make the SFs reachable by their clients.

8. SLCM -> Sub::i_ServiceNotify::notify()
   Finally, the Subscription component is notified about the new service instance. The instance identifier and its service template are passed as input parameters in this operation.

The new service instance is deployed, active and ready for subscription and use. The service nodes providing the service, composing the service domain(s) defined for that service instance, have the required software configuration (new executables, DPE repositories updated) and the involved DPE services are configured to support the service instance.

9. These Node Managers provide the SLCM with the interfaces required to handle the deployment and configuration of service software in a service node and to control and monitor the node resources. They are not shown for simplicity.

4.5.8 Modify an existing service

This scenario describes the procedure for modifying a service instance already deployed in the service network. Two types of updates are considered: changes in the service template (additional parameters or new values in already existing ones) and changes in the service software[10]. The first type of modifications affects the subscription component mainly and does not necessarily require any new software deployment. The second requires new service software deployment, configuration and activation.

The service instance is already deployed. The service instance is active and giving service, the service factories are handling some (active or suspended) service sessions.

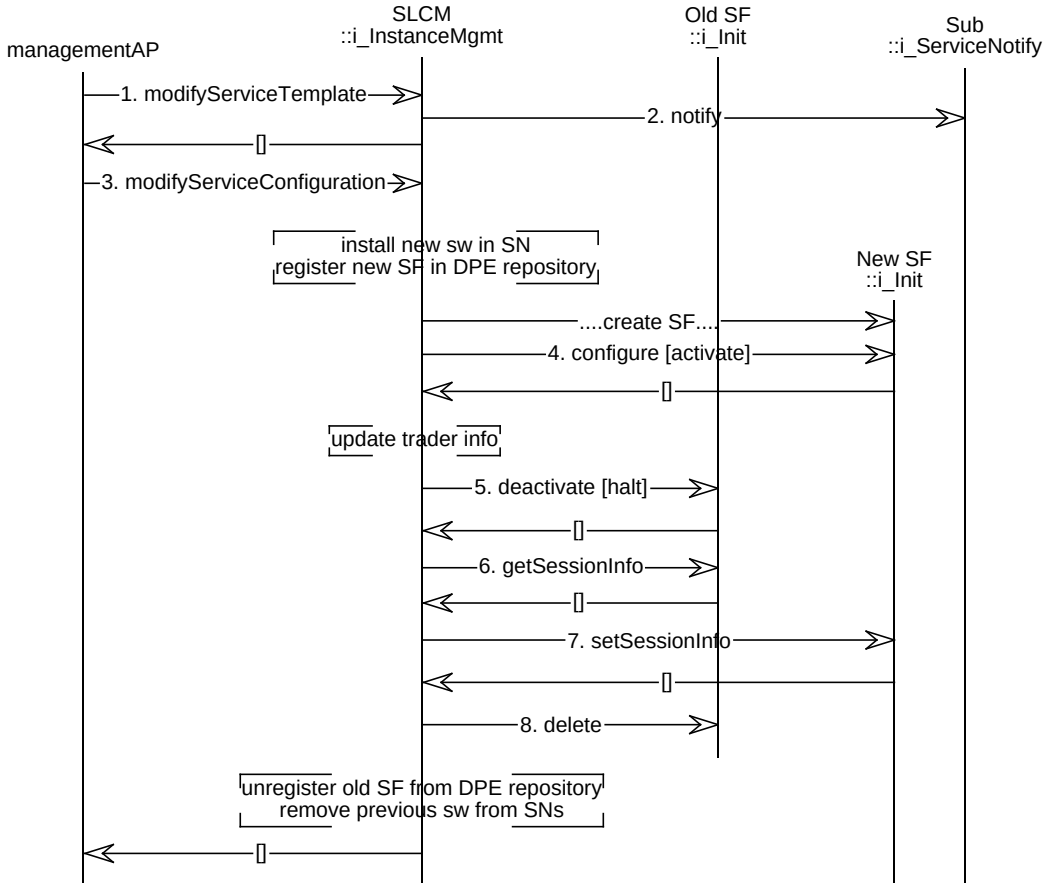


Figure 4-36. Modify an existing service.

1. managementAP -> SLCM::i_InstanceMgmt::modifyServiceTemplate()
   The management application requests the modification of the service template. A new service template is passed as a parameter.
2. SLCM -> Sub::i_ServiceNotify::notify()
   SLCM notifies the Subscription component about the new service instance. The instance identifier and its new service template are passed as input parameters in this operation.

10. We are considering changes in the SF software. Updates of other service components might not affect the running SFs and service sessions. As it represents a simpler case, it is not shown in this document.

Sub will take the required actions, like checking the consistency of the new service template with the already defined service profiles. In some cases, it may require a new contract negotiation with the subscribers.

3. managementAP -> SLCM::i_InstanceMgmt::modifyServiceConfiguration()
   The management application requests the modification of the service configuration. If this new configuration involves an update of the service software, this is deployed over the corresponding service nodes and registered in each involved node DPE repository.

   If the SF software is updated, a new SF is started.

4. SLCM -> new SF::i_Init::configure[&activate]()
   SLCM configures the new SF and changes its state to `active`. It can be done in the same `configure` operation or require an additional call to the `activate` operation.

   Then, the SLCM updates the information in the involved trader and location DPE services, so that the new SFs and not the old ones are reachable by new clients.

5. SLCM -> new SF::i_Init::configure[&activate]()
   SLCM configures the new SF and changes its state to `active`. It can be done in the same `configure` operation or require an additional call to the `activate` operation.

6. SLCM -> old SF::i_Init::deactivate[halt]()
   SLCM changes the state of the replaced SF to `deactivating` or `inactive`, (operations `deactivate` or `halt`, respectively) depending on its will of keeping the existing sessions alive or not after the request.

   The next two operations show a mechanism for transferring the control of existing service sessions from one SF to another. This allows to delete the old SFs immediately and keep the existing sessions alive.

7. SLCM -> old SF::i_Init::getServiceSessionInfo()
   SLCM gets the information about all the existing service sessions (active or suspended).

8. SLCM -> new SF::i_Init::setServiceSessionInfo()
   SLCM transfers the control of the old SF's service session instances to the new SF. The new SF may notify the UAs involved in suspended service sessions about the new `resume` interfaces at the new SF.

9. SLCM -> old SF::i_Init::delete()
   At this moment, the SLCM can delete the old SF. Its related software is removed from the service node and the related information is deleted from the DPE repositories.

The Sub component is working with the new service template and the already defined service profiles are consistent with the new template.

The clients can no longer use the service through the replaced SFs. Existing sessions can still be running under the control of the new SFs and all the new sessions are requested to the new SFs. Service nodes, DPE repositories and DPE services are updated with the information required to make the new service software active and reachable.

4.5.9 Withdraw a service

This event trace represents the procedure followed by the SLCM in the withdrawal of a service.

The service instance is already deployed, active and giving service and there are customers subscribed to the service.

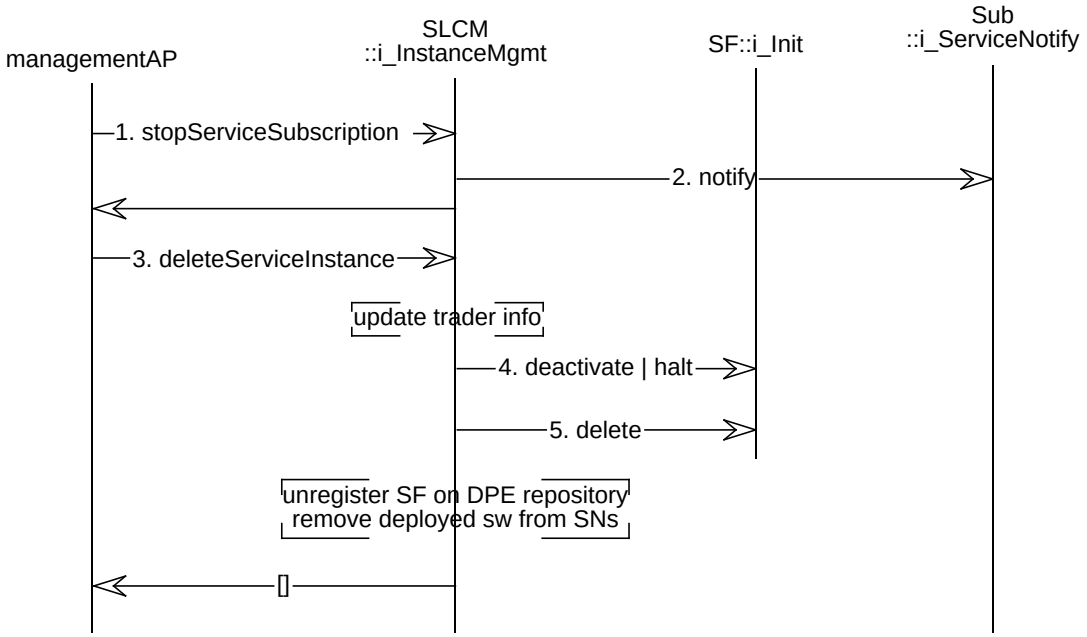


Figure 4-37. Withdraw a service,

1. managementAP -> SLCM::i_InstanceMgmt::stopServiceSubscription()
   Some time before withdrawing the service, the administrator requests the stop of the subscription to that service to the SLCM.

2. SLCM -> Sub::i_ServiceNotify::notify()
   SLCM notifies the Subscription component about the service instance withdrawal.

   Sub will take the required actions, like informing the customers subscribed to that service that the service will be withdrawn. The service contracts shall be cancelled or replaced by contracts to a new service instance, in case the withdrawn service instance is substituted by a new one.

3. managementAP -> SLCM::i_InstanceMgmt::deleteServiceInstance()
   The management application requests the service instance deletion. The SLCM withdraws the trader entries for that service instance, so that it is no longer reachable.

4. SLCM -> SF::i_Init::deactivate[or halt]()
   SLCM requests the SF stop. For this sake, it changes the SFs state to `inactive`. It can be done in two ways: waiting for the existing sessions to finish (`deactivate`) or deleting all sessions immediately (`halt`), after the proper indications.

5. SLCM -> SF::i_Init::delete()
   Once the SFs are inactive, the SLCM deletes them.

   After the deletion, the service software is unregistered from the DPE repositories and removed from the service nodes.

The Sub component is aware of the withdrawal and is not offering the withdrawn service in the list of available (subscribable) services.

The customers can no longer use the service or contract it. There is no information about the removed service instance in DPE repositories, services or facilities and the related service software is not residing in any service node.

4.5.10 Ancillary On-line Accounting Service

Basic on-line accounting approach built-in the TINA service architecture is already described in Section 4.3.5 and Section 4.4.17. User-customized billing can be provided as a separate service (a specific USM/SSM) to control the user billing information in a retailer domain. In this case, an accounting management specific interface should be defined for this service. This approach gives more flexibility to on-line accounting.

- By way of ancillary accounting USM (AA-USM), asUAP are able to contact namedUA, to derive accounting/billing information, and to pay the bill electronically. A user customized accounting/billing applications can be run as ssUAP, which can contact namedUA in the same manner.
- AA-USM can be upgraded or customized totally independent of other interface specifications in SCS and Ret. Therefore, user-customized or retailer-specific billing interfaces can be offered through AA-USM, from which more intelligent or sophisticated billing options may be made available.

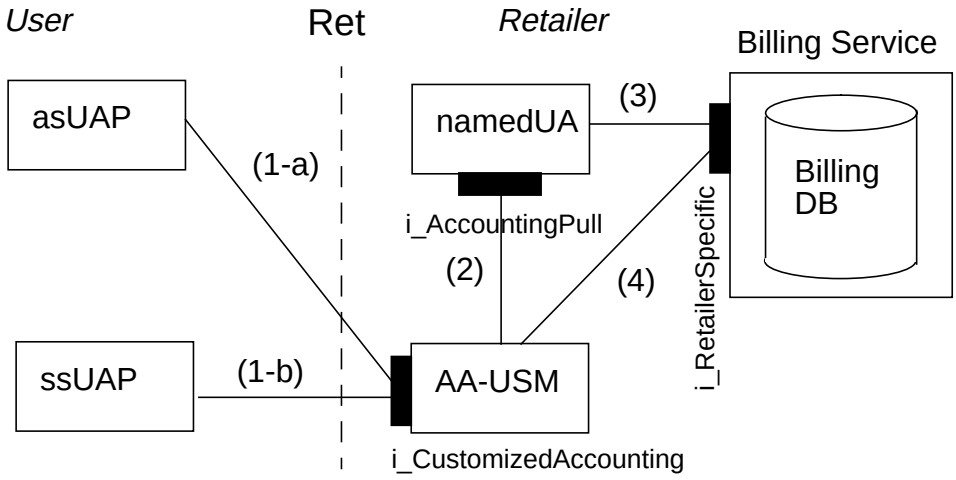


Figure 4-38. Ancillary On-line Accounting Service

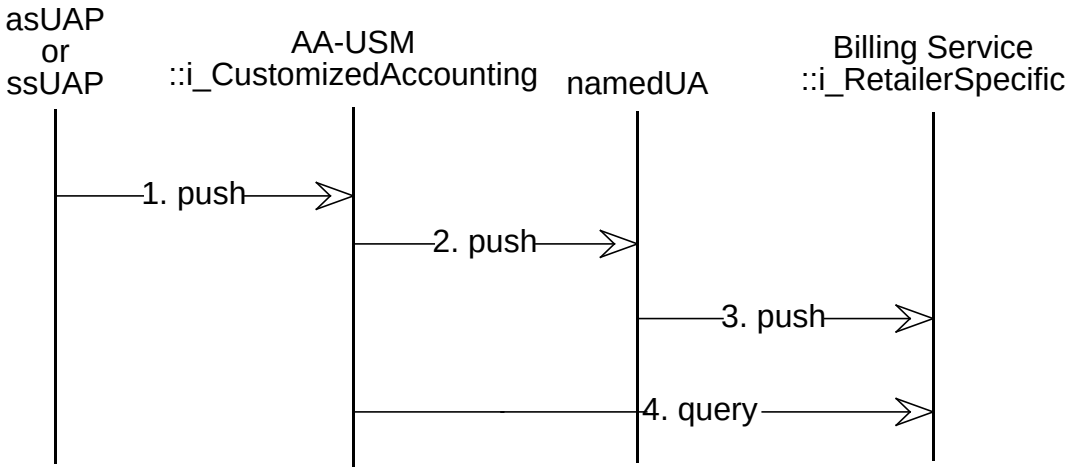
Figure 4-38 illustrates ancillary on-line accounting service. There are two scenario shown in the figure. In the first scenario (1-a), an ancillary on-line accounting service (AOA-service) is requested from asUAP, and the requested component (AA-USM) is created by the retailer. In the second senario, a special user billing application is started (ssUAP), while an ancillary on-line accounting service is requested at the same time. In either case, the rest of the sequence (2-4) are the same.

1-a. An access session UAP (asUAP) requests an AOA-service to the retailer. The retailer creates an AA-USM, whose interface reference is returned to the asUAP.

1-b. a special user billing application (ssUAP) is started in the user domain, while an AOA-service is requested at the same time. The retailer creates an AA-USM, whose interface reference is returned to ssUAP.

2. NamedUA only supports basic per user accounting interface (i_AccountingPull). NamedUA may cache accounting/billing information of active service sessions, however, billing records need to be stored in a billing database to give the records permanency. NamedUA is acting as a front end to a possibly retailer-specific billing service, as far as billing records are concerned.

3. When billing query are obtained from AA-USM, of which NamedUA has no cached billing records, namedUA translates and forwards the query to the billing database.
4. AA-USM can directly access to Billing DB, when necessary. In particular, when billing needs are highly retailer-specific and have to be customized, this approach is more direct, which would enable simpler realization. Currently, billing service interface is not within the scope of SCS. When electronic bill-payment is considered, in particular, additional information such as credit card number may have to be passed from the user to the retailer, with more involved interaction between (as or ss)UAP and AA-USM, and between AA-USM and the billing service.



Due to the nature of AA-USM, details of its interface can be quite service specific, thus it is not defined in the current SCS. Some examples may be included in the future version, however. For example, the specified AA-USM may convert its billing information to an e-mail, with double certificates by the user and the retailer attached, are sent to a virtual banking firm, which handles all the payment request of the user. More sophisticated e-commerce oriented approach can be implemented as an AA-USM within the basic TINA accounting architecture.

It is to be noted that neither asUAP or ssUAP are allowed to access namedUA directly, which would violate security principle of the current Ret design.

5. Acronyms

Amc	Accounting Management component
anonUA	anonymous User Agent
asUAP	Access Session User Application
CC	Connection Coordinator
CO	Computational Object
CompD_USS	Composer Domain Usage Service Session
CompUSM	Composer Usage Session Manager
ConS	Connectivity Service inter-domain reference point
CORBA	Common Object Request Broker Architecture
COS	Common Object Service
CP	Connectivity Provider
CSM	Communication Session Manager
DPE	Distributed Processing Environment
D_USS	Domain Usage Service Session
FCAPS	Fault, Configuration, Accounting, Performance, Security
FS	Feature Set
GDMO	Guidelines for the Definition of Managed Objects
GRM	General Relationship Model
IA	Initial Agent
IDL	Interface Definition Language
IR	Interface Reference
KTN	Kernel Transport Network
LNC	Layer Network Coordinator
management AP	Management Application
Mgmt Ctxt	Management Context
MUSM	Member Usage Service Session Manager
namedUA	Named User Agent
NAP	Network Access Point
NCF	Network Flow Connection
NCS	Network Component Specification
NFEP	Network Flow End Point
NRA	Network Resource Architecture
NRIM	Network Resource Information Model
ODL	Object Definition Language
oIs	On-line Subscription
PA	Provider Agent
PeerA	Peer Agent
PeerD_USS	Peer Domain Usage Service Session
PeerUSM	Peer Usage Session Manager
PM	Performance Monitoring
prim	Primary Service
PSS	Provider Service Session
PS_USS	Provider Domain User Service Session

QoS	Quality of Service
Ret	Retailer inter-domain reference point
RP	Reference Point
SAE	Subscription Assignment Entity
SAG	Subscriber Assignment Group
SC	Service Component
SCM	Service Contract Manager
SCS	Service Component Specification
SDM	Service Deployment Manager
SF	Service Factory
SFC	Stream Flow Connection
SFEP	Stream Flow End Point
SIM	Service Instance Manager
SLCM	Service Life Cycle Management
SN	Service Network
SSM	Service Session Manager
ssUAP	Service Session User Application
STM	Service Type Manager
Sub	Subscription Management Component
SubAg	Subscriber Agent
SubMgr	Subscriber Manager
TCM	Trail Connection Manager
TCon	Terminal Connection inter-domain reference point
TCSM	Terminal Communication Session Manager
TINA [-C]	Telecommunications Information Networking Architecture [Consortium]
ToM	Terms of Management
UA	User Agent
UAF	User Agent Factory
UD_USS	User Domain Usage Service Session
USM	User Service Session Manager
WYSWYP	What You See is What You Pay

6. References

The TINA-C documents may be acquired from the TINA-C WWW page at:

http://tinac.com:4070/index.html

The TINA-C web site provides a search engine. To find a particular document use the document number or the title of the document as search parameters.

TINA-C Documents

TINA-C Valid Baseline Documents

[1] C**.** Abarca, P. Farley, J. Forslöw, J. C. García, T. Hamada, P. F. Hansen, S. Hogg, H. Kamata, L. Kristiansen, C. A. Licciardi, H.Mulder, E. Utsunomiya, M. Yates, Service Architecture, Version 5.0, TINA-C, June 1997; Public.
/u/tinac/97/services/docs/sa/sa5.0/final/main.ps (+annex.ps; not part of the baseline)

[2] M. Yates, W. Takita, L. Demounem, R. Jansson, H. Mulder(ed.) *TINA Business Model and Reference Points,* Version 4.0, TINA-C, May 1997; Public.
/u/tinac/97/integration/docs/business/viewable/final_v4.0.ps

[3] H. Christensen, E. Colban, *Information Modelling Concepts,* Document No. TB_EAC.001_1.2_94, TINA-C, April1995; public.
/u/tinac/94p2/viewable/info.ps

[4] T. Handegård, Many TINA-C Core Team Members, *Computational Modelling Concepts,* Version 3.2, Document No. TP_HC.012_3.2_96, TINA-C, May1996; TINA-C internal.
/u/tinac/96/dpe/docs/computational_model/v3.2/cmc.ps

[5] A. Parhar, *TINA Object Definition Language Manual,* Version 2.3, Document No. TR_NM.002_2.3_96, TINA-C, July1996; TINA-C internal.
/u/tinac/96/dpe/viewable/odl_manual_v2.3.ps

[6] N. Natarajan, H. Flinck, R. Rosli, *Network Resource Information Model Specification (NRIM)*, June 1997; to be released.

[7] C. Abarca, J. Forslow, T. Hamada, S. Hogg, H. B. Jeon, D. S. Kim, H. Y. Lee, N. Natarajan, F. Steegmans (Ed.), *Network Resource Architecture (NRA) Version 3.0*, TINA-C, Feb. 1997; public.
/u/tinac/96/resources/viewable/nra_v3.0.ps

[8] H.Mulder (ed.), , *TINA Glossary of Terms*, Version 2.1, TINA-C, January 1997; Public.
/u/tinac/97/integration/docs/glossary/v2.1/GLOSSARY.ps

Miscellaneous TINA-C Core Team Documents

[9] A. Parhar, *TINA-C File and Directory Style Guide*, TINA-C, April 1996; TINA-C internal.
/u/tinac/general/file_style_guide.txt

[10] T. Hamada, *Accounting Management Architecture,* Version1.2, Document No. EN_TH.001_1.2_95, TINA-C, March1996; TINA-C internal. /u/tinac/95/resources/viewable/accounting.ps

[11] TINA-C Core Team, *Response to a Request for Refinements and Solutions - The Ret Reference Point*, Version 1.1, TINA-C; TINA-C internal. /u/tinac/96/integration/rfrs/RFR-96-01/Responses/core_team_v1.1.ps

[12] P. Farley (ed.), S. Hogg, L. Kristiansen, C. A. Licciardi, M. Mampaey, R. Minetti, S. Pensivy, C. Smith, R.S. Westerga, M. Yates, *Ret Reference Point Specifications,* Version 0.8, TINA-C; October 1997, TINA-C internal. /u/tinac/97/integration/rfrs/RFR-96-01/interim/draft0.8/doc/ret.ps

[13] T. Hamada, et al., *Service Quality in TINA*, to be presented at EDOC'97 (Oct. 1997).

International Standards Documents

ISO/IEC and CCITT/ITU-Tdocuments

[14] ISO/IEC DIS 10165-4, CCITT Recommendation X.722, *Information Technology - Open Systems Interconnection - Structure of Management Information - Part 4: Guidelines for the Definition of Managed Objects (GDMO),* International Organization for Standardization and International Electrotechnical Committee, September 1991.

Books

[15] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, N.J., 1991.

7. Acknowledgments

The authors want to thank the following:

- VITAL ACTS project for providing valuable input such as deliverable D10 and IDL specifications for VITAL v2;
- The Ret EG for providing specification of Access and Usage parts of Ret, that has been considered a basis for our specification work;
- Vital and Sprint for providing valuable comments on early versions of the document
- Hiroshi Kamata for his useful comments to early sspecifications;
- The Service Management WG for providing useful input to Subscription components
- The authors would like to thank particularly the external reviewers from the following companies that all provided useful comments:
  - Alcatel: M. Mampay et al.;
    BT: P.Loosemore, M. Ellis;
    France Telecom: D. Guy;
    Hitachi: K. Kusama;
    Lucent Technologies: B. Opdyke;
    NTT: H. Kobayashi
    Sprint: M. Barrow;
    Telia Research: J. Bengtsson et al.

Chelo Abarca
Alcatel Telecom, Madrid
Spain

Patrick Farley
BT
United Kingdom

Juan Carlos García
Telefónica
Spain

Takeo Hamada
Fujitsu Laboratories
Japan

Per Fly Hansen
Tele Danmark
Denmark

Patrick Hellemans
Alcatel
Belgium

Carlo A. Licciardi
CSELT
Italy

Koki Nakashiro
Hitachi
Japan

Martin Yates
BT
United Kingdom

Annex 1. ODL-specs

This section gives the available ODL-specs:

1.1 TINAObjASUA

```
/** TINAObjASUAP.odl                    */
/**                                     */
/** Access Session User Application     */
/**                                     */
/** Author: Patrick Farley (BT)         */
/** Reviewer: Carlo Licciardi (CSELT)   */
/** Creation date: September 5th, 1997  */
/** Review:                             */

#ifndef TINAObjASUAP_ODL
#define TINAObjASUAP_ODL

#include "TINACommonTypes.idl"
#include "TINAUserAccess.idl"
#include "TINAAccessCommonTypes.idl"
#include "TINAScsASUAPIntra.idl"
#include "TINAScsPAIntra.idl"


object TINAObjASUAP {
requires
        // asUAP
        TINAScsPAIntra::i_Initial,
        TINAScsPAIntra::i_Access,
        TINAScsPAIntra::i_AccountingPull,

        // USM
        TINAScsUSMIntra::i_MgmtCtxt;

supports
        // PA client
        TINAScsASUAPIntra::i_Access;

initial
        TINAScsPAIntra::i_Initial;
};

#endif  // TINAObjASUAP_ODL
```

1.2 TINAObjPA

```
/** TINAObjPA.odl                       */
/**                                     */
/** Provider Agent                      */
/**                                     */
/** Author: Patrick Farley (BT)         */
/** Reviwers: Carlo Licciardi (CSELT)   */
/** Creation date: August 25th, 1997    */


#ifndef TINAObjPA_ODL
#define TINAObjPA_ODL

#include "TINAProviderInitial.idl"
#include "TINAProviderAccess.idl"
#include "TINAAccessCommonTypes.idl"
#include "TINACommonTypes.idl"
#include "TINAUserInitial.idl"
#include "TINAUserAccess.idl"
#include "TINAScsPAIntra.idl"
```

```
#include "TINAScsASUAPIntra.idl"
#include "TINAScsSSUAPIntra.idl"


object TINAObjPA {
requires
        /** asUAP       */
        TINAScsASUAPIntra::i_Access,

        /** ssUAP       */
        TINAScsSSUAPIntra::i_AccessInitialise,

        /** IA  */
        TINAProviderInitial::i_ProviderInitial,
        TINAProviderInitial::i_ProviderAuthenticate,

        /** UA  */
        TINAProviderAccess::i_ProviderNamedAccess,
        TINAProviderAccess::i_ProviderAnonAccess,
        TINAScsNamedUAIntra::i_AccountingPull,
        TINAProviderAccess::i_DiscoverServicesIterator;

supports
        TINAScsPAIntra::i_Init,                 /** Initial interface   */

        /** asUAP client        */
        TINAScsPAIntra::i_Initial,
        TINAScsPAIntra::i_Access,               /* also ssUAP as client */
        TINAProviderInitial::i_ProviderAuthenticate,
        TINAScsPAIntra::i_AccountingPull,

        /** UA client   defined in Ret-RP       */
        TINAUserInitial::i_UserInitial,
        TINAUserAccess::i_UserAccess,
        TINAUserAccess::i_UserInvite,
        TINAUserAccess::i_UserTerminal,
        TINAUserAccess::i_UserAccessSessionInfo,
        TINAUserAccess::i_UserSessionInfo;

initial
        TINAScsPAIntra::i_Init;
};

#endif  /** TINAObjPA_ODL */
```

1.3 TINAObjIA

```
/** TINAObjIA.odl                       */
/**                                     */
/** Inital Agent                        */
/**                                     */
/** Author: Martin yates (BT)           */
/** Reviwers: Carlo Licciardi (CSELT)   */
/** Creation date: August 25th, 1997    */
/** Modified by Koki NAKASHIRO          */
/**     date:97-11-10                   */
/**     adding two includes.            */
/**     NamedUA, ProviderInitial        */

#ifndef TINAObjIA_ODL
#define TINAObjIA_ODL

#include "TINAAccessCommonTypes.idl"
#include "TINACommonTypes.idl"
#include "TINAScsNamedUAIntra.idl"
#include "TINAProviderInitial.idl"


object TINAObjIA {
```

```
requires
        TINASCSNamedUAIntra::i_Initial;

supports
        TINAProviderInitial::i_ProviderAuthenticate,
        TINAProviderInitial::i_ProviderInitial;

initial
        TINASCSNamedUAIntra::i_Initial;
};

#endif  /** TINAObjIA_ODL **/
```

1.4 TINAObjNamedUA

```
/** TINAObjNamedUA.odl                 */
/**                                    */
/** Named User Agent                   */
/**                                    */
/** Author: Chelo Abarca (Alcatel)     */
/**         Carlo Licciardi (CSELT)    */
/** Creation date: August 25th, 1997   */
/** Reviewed: September 9th            */

#ifndef TINAObjNamedUA_odl
#define TINAObjNamedUA_odl

/** #include "TINAScsCommonTypes.idl"           */
/** #include "TINAScsAccessCommonTypes.idl"     */
/** #include "TINAScsSubCommonTypes.idl"        */
/** #include "TINAScsSF.idl"                    */
#include "TINAScsNamedUAIntra.idl"
#include "TINAProviderAccess.idl"

object TINAObjNamedUA {
    requires
/**              PA server      */
        TINAUserAccess::i_UserInvite,
        TINAUserAccess::i_UserAccess,
        TINAUserAccess::i_UserTerminal,
        TINAUserAccess::i_UserAccessSessionInfo,
        TINAUserAccess::i_UserSessionInfo,

/**              SSM server     */
        TINAScsSSMIntra::i_Join,

/**              SF server      */
        TINAScsSF::i_SSCreate,
        TINAScsSF::i_SSManage,
        TINAScsSF::i_Resume,

/**              Sub server     */
        TINAScsSubscriberInfoAccess::i_SubscriberInfoQuery;

supports
        TINAScsNamedUAIntra::i_Initial,                 /** IA client   */
        TINAScsNamedUAIntra::i_SessionInfo,             /** USM client  */
        TINAScsNamedUAIntra::i_InvitationDelivery,      /** SSM and PeerA clients  */
        TINAScsNamedUAIntra::i_AccountingPull,          /** asUAP and PA client  */
        TINAScsNamedUAIntra::i_AccountingPush,          /** SSM and USM clients  */
        TINAScsNamedUAIntra::i_Initial,
        TINAScsNamedUAIntra::i_ServiceProfileCustomization,
        TINAScsNamedUAIntra::i_UsrProfileManagement,

/**             PA client       */
        TINAProviderAccess::i_DiscoverServicesIterator,
        TINAProviderAccess::i_ProviderAccess,
        TINAProviderAccess::i_ProviderAccessGetInterfaces,
        TINAProviderAccess::i_ProviderAccessInterfaces,
```

```
        TINAProviderAccess::i_ProviderAccessRegisterInterfaces,
        TINAProviderAccess::i_ProviderNamedAccess,


        TINAScsNamedUAIntra::i_SubscriptionNotify;      /** Sub client  */

initial
        TINAScsNamedUAIntra::i_Init;

};


#endif
```

1.5 TINAObjSub

```
//
// File TINAObjSub.odl
// It describes the Subscription Management Component
//

#ifndef TINAOBJSUB_ODL
#define TINAOBJSUB_ODL

#include "TINAScsSubInitial.idl"// Main interfaces: Initial, Subscribe
                                // and Notify
#include "TINAScsSubscriberInfoAccess.idl"
                          // Interfaces related to subscriber information mgt:
                          //     Subscriber, UAQueryInfo
#include "TINAScsServiceContractInfoAccess.idl"
                          // Interfaces related to service contract management.

// missing module
// #include "TINAScsServiceLCMgmt.idl"
#include "TINAScsNamedUAIntra.idl"

//  Subscription Management component

object TINAObjSub{
behavior
requires
// SLCM
         TINAScsServiceLCMgmt::i_ServiceQuery,
// UA client
         TINAScsNamedUAIntra::i_SubscriptionNotify;

supports
// All clients
         TINAScsSubInitial::i_InitialAccess,
// UA client
         TINAScsSubscriberInfoAccess::i_SubscriberInfoQuery,
// SSMols (or management application)
         TINAScsSubInitial::i_Subscribe,
         TINAScsSubscriberInfoAccess::i_SubscriberInfoMgmt,
         TINAScsServiceContractInfoAccess::i_ServiceContractInfoMgmt,
// SLCM
         TINAScsSubInitial::i_ServiceNotify;

initial
        TINAScsSubInitial::i_InitialAccess;
};

#endif
```

1.6 TINAObjSSMols

```
// File: TINAObjSSMols.odl
//
// Service Session Manager specific for the online subscription
// management service.
//
// Author: Juan C Garcia (Telefonica)
// Creation date: October 21st, 1997
//
// Revision by Koki NAKASHIRO(HITACHI)
// date:97-11-10
// some syntax check(miner change)
//
// Last modification date: November 10th, 1997

#ifndef TINAObjSSMols_ODL
#define TINAObjSSMols_ODL

#include "TINAScsSSMInit.idl"
#include "TINAScsSSMIntra.idl"
#include "TINAScsUSMIntra.idl"
#include "TINAScsSF.idl"
#include "TINAScsNamedUAIntra.idl"
#include "TINAScsSSMProviderBasicUsage.idl"
#include "TINAScsSubscriptionService.idl"
#include "TINAPartyBasicExtUsage.idl"

object TINAObjSSMols {

requires

// from Ret on ssUAP
        TINAPartyBasicExtUsage::i_PartyBasicExtReq, /** add "Req" OK? **/

// UA server
        TINAScsNamedUAIntra::i_AccountingPush,

// anonUA server
//      TINAScsAnonUA::i_AccountingPush, /** AnonUA? where? **/

// SF server
        TINAScsSF::i_SSEvents, /** Event -> Events **/
        TINAScsSF::i_SSManage;

supports

// Ret - UAP client
        TINAProviderBasicUsage::i_ProviderBasicReq,

// Ret - Subscription Service Specific interface
        TINAScsSubscriptionService::i_ProviderOlsSi,
```

```
// UA client
        TINAScsUSMIntra::i_SessionCtrl, /** SSM->USM **/

// SF client
        TINAScsSSMInit::i_Init,
        TINAScsSSMIntra::i_AccountingPushMgmt,

// SSM Client resume
        TINAScsSSMIntra::i_Resume;

initial
        TINAScsSSMInit::i_Init;

};

#endif // TINAObjSSMols_ODL
```

1.7 TINAObjSLCM

```
//
// File       : TINAObjSLCM.odl
// Decription : Service LifeCycle Management object
// Authors    : Hiroshi Kamata
//
// $Log$
//      9-25-97 v0.1 Initial Draft      by Hiroshi Kamata

#ifndef TINAObjSLCM_ODL
#define TINAObjSLCM_ODL

// missing module
// #include "TINAScsServiceLCMgmt.idl"

#include "TINAScsSF.idl"

//  Service LifeCycle Mangement group object
object TINAObjSLCM {
behavior
        "To be provided." ;

requires
// I'd like to change the name to i_SLCMgmt
// sinse it includes management, not only for initialization

        // SF
        TINAScsSF::i_Init,
        // Sub
        TINAScsSubInitial::i_ServiceNotify;

supports
        // Interface for Service Type management
        TINAScsServiceLCMgmt::i_TypeMgmt,

        //old...TINAScsServiceLCMgmt::i_SessionInstanceMgmt,
        //sorry. for the consistency to the SCS document.
        TINAScsServiceLCMgmt::i_InstanceMgmt,

        // Interface for Service Deployment/Withdrawal management
        TINAScsServiceLCMgmt::i_DeploymentMgmt,

        // Query interface for information required to subscription
        TINAScsServiceLCMgmt::i_ServiceQuery;

initial
```

```
        //TINAScsServiceLCMgmt::i_Init;
        TINAScsSF::i_Init;

};

#endif // TINAObjSLCM_ODL
```

1.8 TINAObjSF

```
// File TINAObjSF.odl
// Authors: Carlo Licciardi
// Last update: 9/22/97

#ifndef TINAObjSF_ODL
#define TINAObjSF_ODL

#include "TINAScsSF.idl"
#include "TINAScsSSMInit.idl"
#include "TINAScsSSMIntra.idl"
#include "TINAScsUSMInit.idl"
#include "TINAScsUSMIntra.idl"
//  Service Factory Object

object TINAObjSF {
behavior
requires
// SSM
         TINAScsSSMInit::i_Init,
         TINAScsSSMIntra::i_Resume,
         TINAScsSSMIntra::i_AccountingPushMgmt,
// USM
         TINAScsUSMInit::i_Init,
         TINAScsUSMIntra::i_Resume;
// PeerUSM
//       TINAScsPeerUSMInit::i_Init,
//       TINAScsPeerUSMIntra::i_Resume,
// CompUSM
//       TINAScsCompUSMInit::i_Init,
//       TINAScsCompUSMIntra::i_Resume;
supports
         TINAScsSF::i_SSCreate,
         TINAScsSF::i_SSManage,
         TINAScsSF::i_Init,
         TINAScsSF::i_Resume,
         TINAScsSF::i_SSEvents;
initial
         TINAScsSF::i_Init;
};

#endif
```

1.9 TINAObjSSM

```
// File: TINAObjSSM.odl
//
// Service Session Manager
//
// Author: Per Fly Hansen (Tele Danmark)
// Creation date: August 21st, 1997
// Modification date: September 24thd, 1997
//
// by Koki NAKASHIRO (HITACHI)
// Last Modification date:12 November 1997

#ifndef TINAObjSSM_ODL
#define TINAObjSSM_ODL

#include "TINAScsSSMInit.idl"
```

```
#include "TINAScsSSMIntra.idl"
#include "TINAScsSSMProviderBasicUsage.idl"
#include "TINAScsSSMProviderControlSRUsage.idl"
#include "TINAScsSSMProviderMultipartyUsage.idl"
#include "TINAScsSSMProviderPaSBUsage.idl"
#include "TINAScsSSMProviderVotingUsage.idl"

#include "TINAScsUSMIntra.idl"
#include "TINAScsUSMPartyBasicExtUsage.idl"
#include "TINAScsUSMPartyControlSRUsage.idl"
#include "TINAScsUSMPartyMultipartyIndUsage.idl"
#include "TINAScsUSMPartyMultipartyUsage.idl"
#include "TINAScsUSMPartyVotingUsage.idl"
#include "TINAScsSF.idl"

#include "TINAScsNamedUAIntra.idl"

object TINAObjSSM {
requires
        // namedUA server
        TINAScsNamedUAIntra::i_InvitationDelivery, /* IntraNamedUA -> */
        TINAScsNamedUAIntra::i_AccountingPush,     /*   NamedUAIntra  */

        // anonUA server
        //      TINAScsAnonUA::i_AccountingPush,

        // (specified for PeerA which is not yet defined)
        //      TINAScsPeerA::i_InvitationDelivery,
        //      TINAScsPeerA::i_AccountingPush,

        // SF server
        TINAScsSF::i_SSEvent,
        TINAScsSF::i_SSManage,
        TINAScsSF::i_SSCreate,

        // USM Server
        TINAScsUSMPartyBasicExtUsage::i_PartyBasicExtReq,
        TINAScsUSMPartyBasicExtUsage::i_PartyGetInterfaces,
        TINAScsUSMPartyControlSRUsage::i_PartyControlSRInd,
        TINAScsUSMPartyControlSRUsage::i_PartyControlSRInfo,
        TINAScsUSMPartyMultipartyIndUsage::i_PartyMultipartyInd,
        TINAScsUSMPartyMultipartyUsage::i_PartyMultipartyExe,
        TINAScsUSMPartyMultipartyUsage::i_PartyMultipartyInfo,
        TINAScsUSMPartyVotingUsage::i_PartyVotingInfo,
        TINAScsUSMIntra::i_Resume,
        TINAScsUSMIntra::i_AccountingPush,
        // stream binding on USM
        TINAScsUSMPartyPaSBIndUsage::i_PartyPaSBInd,
        TINAScsUSMPartyPaSBUsage::i_GeneralStreamInfo,
        TINAScsUSMPartyPaSBUsage::i_PartyGeneralStreamInfo,
        TINAScsUSMPartyPaSBUsage::i_PartyPaSBExe,
        TINAScsUSMPartyPaSBUsage::i_PartyPaSBInfo;

supports
        // USM client (eventually PeerUSM, CompUSM)
        TINAScsSSMProviderBasicUsage::i_ProviderGetInterfaces,
        TINAScsSSMProviderBasicUsage::i_ProviderRegisterInterfaces,
        TINAScsSSMProviderBasicUsage::i_ProviderInterfaces,
        TINAScsSSMProviderBasicUsage::i_ProviderBasicReq,
        TINAScsSSMProviderControlSRUsage::i_ProviderControlSRReq,
        TINAScsSSMProviderMultipartyUsage::i_ProviderMultipartyReq,
        TINAScsSSMProviderVotingUsage::i_ProviderVotingReq,

        // stream binding
        TINAScsSSMProviderPaSBUsage::i_ProviderPaSBReq,

        // UA (named+anon), PeerA client
        TINAScsSSMIntra::i_Join,

        // SF client
        TINAScsSSMInit::i_Init,
```

```
        TINAScsSSMIntra::i_Resume,

        // USM or SF,
        TINAScsSSMIntra::i_AccountingPushMgmt,

        //CSM client
        TINAScsSSMIntra::i_AccountingPush;

// CompUSM, PeerUSM ...
//
initial
        TINAScsSSMInit::i_Init;

};

#endif // TINAObjSSM_ODL
```

1.10 TINAObjSSUAP

```
// FILE: TINAObjSSUAP.odl
//
// VERSION: 1
// DATE 21 August 97
//
// ODL for service session User APplication
// for the TINA- SCS
//
// COMMENTS:
// This is incomplete refer to the IDL file for proper definitions.
//
// MODIFICATIONS:
//      Revision 1.0.1 97-09-29
//      Changed on 29 September 97 to accomodate new naming conventions,
//      remove unwanted SB interfaces
//
//      Revision 1.0.2 97-11-10 by Koki NAKASHIRO, Carlo Alberto Licciardi
//      update to new SCS naming scheme and module structure
//      fixing inconsistency idl, odl, html files.
//      (add new includes, move definitions to other file, etc.)

#ifndef _SSUAP_ODL
#define _SSUAP_ODL

#include "TINAProviderAccess.idl"
#include "TINAPartyBasicExtUsage.idl"
#include "TINAPartyMultipartyUsage.idl"
#include "TINAPartyMultipartyIndUsage.idl"
#include "TINAPartyVotingUsage.idl"
#include "TINAPartyControlSRUsage.idl"
#include "TINAScsSSUAPIntra.idl"
#include "TINAPartyPaSBUsage.idl"
#include "TINAPartyPaSBIndUsage.idl"

object TINAObjSSUAP {
requires
                // PA server
        TINAScsPAIntra::i_Access,
                // Ret USM server
        TINAProviderBasicUsage::i_ProviderBasicReq,
        TINAProviderMultipartyUsage::i_ProviderMultipartyReq,
        TINAProviderVotingUsage::i_ProviderVotingReq,
        TINAProviderControlSRUsage::i_ProviderControlSRReq,
        TINAProviderPaSBUsage::i_ProviderPaSBReq;
supports
                // Ret - USM client
        TINAPartyBasicExtUsage::i_PartyBasicExtReq,     /* Req added */
        TINAPartyMultipartyUsage::i_PartyMultipartyExe,
        TINAPartyMultipartyUsage::i_PartyMultipartyInfo,
        TINAPartyMultipartyIndUsage::i_PartyMultipartyInd,
```

```
        TINAPartyVotingUsage::i_PartyVotingInfo,
        TINAPartyControlSRUsage::i_PartyControlSRInd,   /* SR added */
        TINAPartyControlSRUsage::i_PartyControlSRInfo,  /* SR added */
        TINAPartyPaSBUsage::i_PartyPaSBExe,
        TINAPartyPaSBUsage::i_PartyPaSBInfo,
        TINAPartyPaSBIndUsage::i_PartyPaSBInd,
                // PA client
        TINAScsSSUAPIntra::i_AccessInitialise;

initial
        TINAScsSSUAPIntra::i_AccessInitialise;
};

#endif //_SSUAP_ODL
```

1.11 TINAObjUSM

```
// FILE: TINAObjUSM.odl
//
// VERSION: 1.02
// DATE 21 August 97
//
// ODL for Usage Sesion Manager
// for the TINA- SCS
//
// COMMENTS:
// This is incomplete refer to the IDL file for proper definitions.
//
// MODIFICATIONS:
//      Revision 1.0.1  97-9-4 by Takeo Hamada
//          i_usmMgmtCtxt has been added to object USM.
//      Revision 1.0.2 97-9-4 by Martin Yates
//          corrections to revisions in 1.0.1, extra comments, syntactic corrections
//      Revision 1.0.3 97-09-29 by Martin Yates
//              update to new SCS naming scheme and module structure
//
//      Revision 1.0.4 97-11-10 by Koki NAKASHIRO, Carlo Alberto Licciardi
//              update to new SCS naming scheme and module structure
//              fixing inconsistency idl, odl, html files.
//              (add new includes, move definitions to other file, etc.)
//              What do we do to TINAScsUSMExtMngt...???

#ifndef _USM_ODL
#define _USM_ODL

#include "TINAScsUSMIntra.idl"
#include "TINAScsUSMPartyBasicExtUsage.idl"
#include "TINAScsUSMPartyMultipartyIndUsage.idl"
#include "TINAScsUSMPartyMultipartyUsage.idl"
#include "TINAScsUSMPartyPaSBIndUsage.idl"
#include "TINAScsUSMPartyPaSBUsage.idl"
#include "TINAScsUSMPartyVotingUsage.idl"
#include "TINAScsUSMPartyControlSRUsage.idl"
#include "TINAScsUSMInit.idl"
#include "TINAProviderBasicUsage.idl"
#include "TINAProviderMultipartyUsage.idl"
#include "TINAProviderVotingUsage.idl"
#include "TINAProviderControlSRUsage.idl"
#include "TINAProviderPaSBUsage.idl"
#include "TINAPartyPaSBIndUsage.idl"
//#include "TINAScsUSMExtMngt.idl"

object TINAObjUSM {
        behavior
                //      Represents one particpant in a service session.
                //      Executes session and service specific control commands from UAP
                //      Indicates and informs UAP of service session activity
                //      Maintains and issues accounting information for its particpant
```

```
        requires
                //from Ret on ssUAP
                TINAPartyBasicExtUsage::i_PartyBasicExt,
                TINAPartyMultipartyUsage::i_PartyMultipartyExe,
                TINAPartyMultipartyUsage::i_PartyMultipartyInfo,
                TINAPartyMultipartyUsage::i_PartyMultipartyInd,
                TINAPartyVotingUsage::i_PartyVotingInfo,
                TINAPartyControlSRUsage::i_PartyControlSRInd, /** insert SR **/
                TINAPartyControlSRUsage::i_PartyControlSRInfo,/** insert SR **/
                // stream binding on ssUAP
                TINAPartyPaSBIndUsage::i_PartyPaSBInd,
                TINAPartyPaSBUsage::i_PartyPaSBExe,
                TINAPartyPaSBUsage::i_GeneralStreamInfo,
                TINAPartyPaSBUsage::i_PartyGeneralStreamInfo,
                TINAPartyPaSBUsage::i_PartyPaSBInfo,

                //      UA server
                TINAScsNamedUAIntra::i_AccountingPush,
                TINAScsNamedUAIntra::i_SessionInfo,

                //      SSM server
                TINAScsSSMProviderBasicUsage::i_ProviderGetInterfaces,
                TINAScsSSMProviderBasicUsage::i_ProviderRegisterInterfaces,
                TINAScsSSMProviderBasicUsage::i_ProviderInterfaces,
                TINAScsSSMProviderBasicUsage::i_ProviderBasicReq,
                TINAScsSSMProviderControlSRUsage::i_ProviderControlSRReq,
                TINAScsSSMProviderMultipartyUsage::i_ProviderMultipartyReq,
                        /** Req --> Usage ok? **/
                TINAScsSSMProviderPaSBUsage::i_ProviderPaSBReq,
                TINAScsSSMProviderVotingUsage::i_ProviderVotingReq,
                TINAScsSSMIntra::i_AccountingPushMgmt,
                // stream binding on SSM. Need to verify that this remains consistent
                // with ret streams evolution. Currently based on Ret0.7
                TINAScsSSMProviderPaSBUsage::i_ProviderPaSBReq;

        supports
                // Ret - UAP client
                TINAProviderBasicUsage::i_ProviderBasicReq,
                TINAProviderMultipartyUsage::i_ProviderMultipartyReq,
                TINAProviderVotingUsage::i_ProviderVotingReq,
                TINAProviderControlSRUsage::i_ProviderControlSRReq,
                        /* SR added */
                TINAProviderPaSBUsage::i_ProviderPaSBReq,

                // Non Ret - UAP Client (change ExtMgmt -> Intra)
                TINAScsUSMIntra::i_MgmtCtxt,

                // UA client
                TINAScsUSMIntra::i_SessionCtrl,
                TINAScsUSMIntra::i_AccountingPushMgmt,

                // SF client
                TINAScsUSMInit::i_Init,

                // SSM Client  resume
                TINAScsUSMIntra::i_Resume,

                // SSM Client accounting
                TINAScsUSMIntra::i_AccountingPush,

                // SSM Client usage
                TINAScsUSMPartyBasicExtUsage::i_PartyBasicExtReq,
                TINAScsUSMPartyBasicExtUsage::i_PartyGetInterfaces,
                TINAScsUSMPartyMultipartyIndUsage::i_PartyMultipartyInd,
                TINAScsUSMPartyMultipartyUsage::i_PartyMultipartyExe,
                TINAScsUSMPartyMultipartyUsage::i_PartyMultipartyInfo,
                TINAScsUSMPartyVotingUsage::i_PartyVotingInfo,
                TINAScsUSMPartyControlSRUsage::i_PartyControlSRInd,
                TINAScsUSMPartyControlSRUsage::i_PartyControlSRInfo,
                // stream binding interfaces may require some changes from Ret
                // when Ret streams are stable
```

```
                TINAScsUSMPartyPaSBIndUsage::i_PartyPaSBInd,
                TINAScsUSMPartyPaSBUsage::i_PartyPaSBExe,
                TINAScsUSMPartyPaSBUsage::i_PartyPaSBInfo,
                TINAScsUSMPartyPaSBUsage::i_GeneralStreamInfo,  /** add **/
                TINAScsUSMPartyPaSBUsage::i_PartyGeneralStreamInfo; /** add **/
        initial
                TINAScsUSMInit::i_Init;

};

#endif  // _USM_ODL
```

1.12 TINAObjAmcLadder

```
#ifndef _TINAScsAmcLadder_odl_
#define _TINAScsAmcLadder_odl__

//
//      File Name: TINAObjAmcLadder.odl
//
//      Decription: accounting ladder element
//        A ladder element is an object that participates in a chain
//      of objects that receive, (optionally) generates and sends
//      accounting events.
//
//      Revision History:
//              9-03-97 v0.21 by Juan C. Garcia
//              9-25-97 v0.3 by Takeo Hamada
//                  with new naming scheme
//

#include "TINAScsAmcObject.idl"

object TINAObjAmcLadder {
behavior
/**
 *  A ladder element is an object that participates in a chain
 *  of objects that receive, (optionally) generates and sends
 *  accounting events.
 **/

requires
    TINAScsAmcObject::i_AmcLadderElement;

supports
    // Interface for accounting control
    // (start/stop/setNotifDestination/...)
    TINAScsAmcObject::i_AccObjectManagement,
    // Interface to receive events.
    TINAScsAmcObject::i_AmcLadderElement;

initial
    TINAScsAmcObject::i_AccObjectManagement;
};

#endif
```

Annex 2. IDL-specs

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

2.1 TINAScsMgmtCtxt

```
#ifndef _TINAScsMgmtCtxt_idl_
#define _TINAScsMgmtCtxt_idl_

//
//      File Name: TINAScsMgmtCtxt.idl
//      Decription: definitions for management context
//      Comments:
//      Revision Histroy:
//         8-30-97 v0.1 by Takeo Hamada
//          9-04-97 v0.11 by Takeo Hamada
//              i_usmMgmtCtxt is moved to USM.odl
//          9-10-97 v0.2 by Takeo Hamada
//              passed hidl compiler
//          9-17-97 v0.25 by Takeo Hamada
//              module structure revised, passed hidl
//              compiler.
//          9-25-97 v0.30 by Takeo Hamada
//              with new naming scheme
//

#ifdef debug
#include "PLATyToolsFix.idl"
#else
#include "CosTrading/CosTrading.idl"
#endif
#include "Security.idl"

module TINAScsMgmtCtxt {

//
//  Caution:  we assume that a structure for ToM and its associated
//      service template are defined somewhere else.
//

    typedef string t_ToMID;     // temporary fix

    /**
     *
     *  Management Context definition.
     *
     *  @member ctxtype F,C,A,P,S context type.
     *  @member props we assume that management context is
     *          represented by a list of property-value
     *          pairs.  For example, an accounting management
     *          context may have properties such as tariffId,
     *          billing  options, recovery options, etc.
     *          The set of properties (management schema) will be
     *          defined separately in respective management
     *          architecture document.
     *  @member proof proof of origin, proof of receipt, which
     *          proves both parties agreed on the content.
     *          it is assumed that MgmtCtxts are negotiated
     *          between User and GCCM.  A proof is attached
     *          at the conclusion of negotiation.
     *  @member tom ID of ToM, from which this MgmtCtxt may be
     *          derived.  When no such ToM exists, the value
     *          should be NULL.
     **/
    struct t_MgmtCtxt {
```

```
        string<8> ctxtype;      // F,C,A,P,S context type

        CosTrading::PropertySeq props;
                // we assume that management context is
                // represented by a list of property-value
                // pairs.  For example, an accounting management
                // context may have properties such as tariffId,
                // billing  options, recovery options, etc.
                // The set of properties (management schema) will be
                // defined separately in respective management
                // architecture document.
        Security::Opaque proof;
                // proof of origin, proof of receipt, which
                // proves both parties agreed on the content.
                // it is assumed that MgmtCtxts are negotiated
                // between User and GCCM.  A proof is attached
                // at the conclusion of negotiation.
        t_ToMID tom;
                // ID of ToM, from which this MgmtCtxt may be
                // derived.  When no such ToM exists, the value
                // should be NULL.
    };

    typedef sequence <t_MgmtCtxt> t_MgmtCtxtList;

    /**
     * Management context ID:
     * a reference to a consistent (submittable) set of
     * management contexts (t_MgmtCtxtList), which
     * can be bound to a service session in question.
     **/
    typedef string t_MgmtCtxtID;

    exception e_usmMgmtCtxt {
        enum t_usmMgmtCtxt_error {
            cannotBind,
            cannotRebind,
            cannotUnbind,
            inconsistentMgmtContext,
            mgmtContextinUse
        } error;
        string reason;
    };

//
//  following interface is commented out, since it is already
//  a part of USM.odl (Rev. 1.0.1, 9-04-97).
//
//    //
//    //  No component is identified yet, to which the following
//    //  interface is to be attached.  Possibly USM.
//    //
//
//    interface i_usmMgmtCtxt {
//      // bindList: binds a list of management context to service
//      // session.
//      boolean bindList(
//          in t_MgmtCtxtList contexts
//      ) raises (e_usmMgmtCtxt);
//      // bindID: binds a list of management context referenced
//      // by ID to service session.
//      boolean bindID(
//          in t_MgmtCtxtID contexts
//      ) raises (e_usmMgmtCtxt);
//
//      // unbindList: unbinds a list of management context to service
//      // session.
//      boolean unbindList(
//          in t_MgmtCtxtList contexts
//      ) raises (e_usmMgmtCtxt);
//      // unbindID: unbinds a list of management context referenced
```

```
//      // by ID to service session.
//      boolean unbindID(
//          in t_MgmtCtxtID contexts
//      ) raises (e_usmMgmtCtxt);
//
//      // rebindList: rebinds a list of management context to service
//      // session.
//      boolean rebindList(
//          in t_MgmtCtxtList contexts
//      ) raises (e_usmMgmtCtxt);
//      // rebindID: rebinds a list of management context referenced
//      // by ID to service session.
//      boolean rebindID(
//          in t_MgmtCtxtID contexts
//      ) raises (e_usmMgmtCtxt);
//    };
};

#endif
```

2.2 TINAScsCommonTypes

```
// FILE: TINAScsCommonTypes.idl
//
// VERSION: 1
// DATE 25 September 97
//
// IDL for Common Types used in TINA Service Component Specs
// that are not found in Ret
//
// AUTHOR: Martin Yates
//
// COMMENTS:
//
// MODIFICATIONS:
//
//
//

#ifndef _TINAScsCommonTypes_IDL
#define  _TINAScsCommonTypes_IDL

#include "TINACommonTypes.idl"

/** Contains common types in the service components specification
        that are not defined in Ret.
**/
module TINAScsCommonTypes {

        enum t_UserIdErrorCode {
                userIdUnknown
        };

        exception e_UserIdError{
                t_UserIdErrorCode UserIdError;
        };

        enum t_IdentifierErrorCode {
                GlobalSessionIdUnknown,
                PartyIdUnknown
        };

        exception e_IdentifierError {
                t_IdentifierErrorCode IdError;
        };

        enum t_PartyIdListOptions {
                OnlyListedIds,          // use all PartId in list and no others
                FirstListedId,          // use the first PartyId in List and ignore others
```

```
            AllIdsIgnoreList        // ignore the list and use all PartyIds known
        };

        struct t_PartyIdListHandler {
            TINACommonTypes::t_PartyIdList forPartIds;
            t_PartyIdListOptions handlerOption;
        };

        enum t_PartyIdListErrorCode {
            NullList,
            InvalidPartyIds
        };

        exception e_PartyIdListError {
            t_PartyIdListErrorCode  listErrorCode;
        };

        enum t_PartySessionErrorCode {
            UnknownError,
            OpNotSupported
        };

        exception e_PartySessionError {
            t_PartySessionErrorCode partySErrorCode;
        };

        enum t_ProviderSessionErrorCode {
            UnknownUsageError,
            UsageNotAllowed,
            UsageNotAccepted,
            UsageOpNotSupported,
            PartySuspended
        };

        exception e_ProviderSessionError {
            t_ProviderSessionErrorCode provSErrorCode;
        };

        enum t_PartyIdErrorCode {
            invalidPartyId
        };

        exception e_PartyIdError {
            t_PartyIdErrorCode idErrorCode;
        };

        enum t_PartyTypeErrorCode {
            invalidPartyType
        };

        exception e_PartyTypeError {
            t_PartyTypeErrorCode typeErrorCode;
        };


};  // TINAScsCommonTypes

#endif //_TINAScsCommonTypes_IDL
```

2.3 TINAScsAmcCommon

```
#ifndef _TINAScsAmcCommon_idl_
#define _TINAScsAmcCommon_idl_

//
//      We assume the following types are defined as
//      common definitions for TINA component specs.
//
//      TINACommonTypes::t_UserId
```

```
//      TINACommonTypes::t_SessionId
//

// #include "NCS_common.idl"    - not needed yet
#include "TINACommonTypes.idl"
#include "Security.idl"

//
//      File Name: TINAScsAmcCommon.idl
//      Decription: common definitions for accounting
//              management in NCS
//      Comments:
//              Original VITAL inputs were provided by
//              George Pavlou, UCL, UK.
//      Revision Histroy:
//          8-1-97 v0.1 by Takeo Hamada
//          8-27-97 v0.2 by Takeo Hamada
//          8-30-97 v0.21 by Takeo Hamada
//          9-03-97 v0.22 by Takeo Hamada
//              added t_ServiceTransactionId
//          9-10-97 v0.23 by Takeo Hamada
//              passed hidl compiler
//          9-17-97 v0.25 by Takeo Hamada
//              module structure revised, passed hidl
//              compiler.
//          9-25-97 v0.3 by Takeo Hamada
//              with new naming scheme.
//

module TINAScsAmcCommon {

    /**
     *
     *  Type of DST corrections.
     *
     * @member  None    not on dst
     * @member  USA     USA
     * @member  AUST    Australia
     * @member  WET     Western Europe
     * @member  EET     Eastern Europe
     * @member  CAN     Canada
     * @member  UK      UK and Eire
     * @member  RUM     Rumania
     * @member  TUR     Turkey
     * @member  AUSTLT  Australia with shift in 1986
     *
     **/

    enum t_amcDsttype {
        None,   // not on dst
        USA,    // USA
        AUST,   // Australia
        WET,    // Western Europe
        MET,    // Middle Europe
        EET,    // Eastern Europe
        CAN,    // Canada
        UK,             // UK and Eire
        RUM,    // Rumania
        TUR,    // Turkey
        AUSTLT  // Australia with shift in 1986
    };

    typedef unsigned short ushort;
    typedef unsigned short ulong;

    /**
     *
     *  Accounting time stamp.
     *
     * @member tm_usec microseconds (0...999)
     * @member tm_sec seconds (0...59)
```

```
 * @member tm_min minutes (0...59)
 * @member tm_hour hour (0...23)
 * @member tm_mday day of month (1...31)
 * @member tm_mon month (1...12)
 * @member tm_year year
 * @member tm_wday day of the week (0...6)
 * @member tm_yday day of the year (1...366)
 *
 **/

struct t_DateTime {
    // it should contain time resolution up to micro sec.,
    // yy/mm/dd/hh/mm/ss forms, timezone.
    ushort tm_usec; // microseconds (0...999)
    ushort tm_sec;  // seconds (0...59)
    ushort tm_min;  // minutes (0...59)
    ushort tm_hour; // hour (0...23)
    ushort tm_mday; // day of month (1...31)
    ushort tm_mon;  // month (1...12)
    ulong tm_year;
    ushort tm_wday; // day of the week (0...6)
    ushort tm_yday; // day of the year (1...366)

    string tm_zone;
    short tz_minuteswest;   // minutes west of Greenwich
    t_amcDsttype tz_dsttime;        // type of dst correction
};

/**
 *  Accounting measurement time duration.
 *
 *  Time duration = tv_usec + 1.0e-6 * tv_usec [sec]
 **/

struct t_Timeval {
    long tv_sec;    // in sec.
    long tv_usec;   // in microsec.
};

enum amc_TimePeriod {
    usec,   // microsecond
    msec,   // millisecond
    sec,
    min,
    hour,
    half_day,
    day,
    three_days,
    week,
    half_month,
    month,
    three_months,
    half_year,
    year,
    three_years
};

struct t_fromTo {
    t_DateTime from;
    t_DateTime to;
};

/**
 *  Unit of measurement
 **/

enum amc_UnitType {
    // number of bits
    bits,
    Kbits,  // Kilo = 2^10
    Mbits,  // Mega = 2^20
```

```
        Gbits,  // Gita = 2^30
        Tbits,  // Tera = 2^40
        Pbits,  // Penta = 2^50

        // bit per second
        bps,
        Kbps,
        Mbps,
        Gbps,
        Tbps,
        Pbps,

        // number of bytes
        bytes,
        Kbytes,
        Mbytes,
        Gbytes,
        Tbytes,
        Pbytes,

        // byte per second
        Byps,
        KByps,
        MByps,
        GByps,
        TByps,
        PByps,

        // number of ATM cells
        cells,
        Kcells,
        Mcells,
        Gcells,
        Tcells,
        Pcells,

        // cell per second
        cps,
        Kcps,
        Mcps,
        Gcps,
        Tcps,
        Pcps
    };

    /**
     *  Accounting event type.
     */

    enum t_AccountingEventType {
        // essential accounting events on Stream Binding
        bindingSB,
        unbindingSB,
        measurementSB,

        // non-essential accounting events on
        // Stream Flow Connection
        doneSFC,
        undoneSFC,
        measurementSFC,

        // Network Flow Connection
        doneNFC,
        undoneNFC,
        measurementNFC,

        // Terminal Flow Connection
        doneTFC,
        undoneTFC,
        measurementTFC,
```

```
        // Trail
        doneTrail,
        undoneTrail,
        measurementTrail,

        // Subnetwork Connection
        doneSNC,
        undoneSNC,
        measurementSNC,

        // Link Connection
        doneLC,
        undoneLC,
        measurementLC,

        // Tandem Connection
        doneTC,
        undoneTC,
        measurementTC
    };

    /**
     *  Accounting measurement type.
     **/

    enum t_AccountingMeasurementType {
        differential_1, // differential measurement form 1
        differential_2, // differential measurement form 2
        regular         // continuous, regular measurement
    };

    /**
     *  Accounting event definition.
     **/

    union t_amcMdata switch (t_AccountingMeasurementType) {
        case differential_1: t_Timeval tval;
        case differential_2: t_fromTo fromTo;
        case regular: t_DateTime time;
    };

    struct t_amcMeasurement {
        t_DateTime timestamp;
        t_AccountingMeasurementType amcMtype;
        t_amcMdata measured;
        amc_UnitType utype;
        long uvalue;
    };

    typedef sequence <t_amcMeasurement> t_amcMeasurementList;

    struct t_AccountingEvent {
        TINACommonTypes::t_UserId userId;
        TINACommonTypes::t_SessionId sessionId;
        t_amcMeasurementList measured;
        Security::Opaque proof; // proof of origin attached by the
                        // originator of the event, when
                        // necessary.
    };

    typedef sequence <t_AccountingEvent> t_AccountingEventList;

    /**
     *  Tariff structure definition.
     *
     *  Details of the interface which admits accessing tariff
     *  structure is yet to be defined.  This id is used by
     *  SSM, and is resolved such that the SSM can derive
     *  necessary tariffing information to calculate bills.
     *  For more details, see TINAScsAmcTariff.idl.
     **/
```

```
    typedef string t_TariffId;

    /**
     *  Service transaction definition.
     *
     *  Service transaction is an information object, as such
     *  it only carries Id to distinguish one from the other.
     *  It purpose is to give use specific service management,
     *  when management context of one user has to be distinguished
     *  from others.  For example, an SSM needs to distinguish
     *  notification interfaces of USM, when its billing information
     *  is to be sent on-line, with which the user (ServiceTransaction)
     *  is associated.
     **/

    typedef string t_ServiceTransactionId;

};

#endif
```

2.4 TINAScsAmcObject

```
#ifndef _TINAScsAmcObject_idl_
#define _TINAScsAmcObject_idl_

#include "CosEventComm.idl"
#include "UMLogManager.idl"
#include "TINAScsMgmtCtxt.idl"
#include "TINAScsAmc.idl"

//
//      File Name: TINAScsAmcObject.idl
//      Decription: accounable object in NCS
//      Revision Histroy:
//          8-1-97 v0.1 by Takeo Hamada
//          8-26-97 v0.2 by Takeo Hamada
//          8-30-97 v0.21 by Takeo Hamada
//          9-03-97 v0.21 by Juan C. Garcia
//              inheritance from i_AccountableObject removed
//              in i_AmcLadderElement definition
//          9-03-97 v0.22 by Takeo Hamada
//              generic Pull and Push interfaces added.
//          9-10-97 v0.23 by Takeo Hamada
//              passed hidl compiler
//          9-17-97 v0.25 by Takeo Hamada
//              module structure revised, passed hidl
//              compiler.
//          9-25-97 v0.3 by Takeo Hamada
//              with new naming scheme
//          10-16-97 v0.4 by Takeo Hamada
//              userId added to getUserLogEntries
//

module TINAScsAmcObject {

    enum t_AmcObject_error {
            cannotStart,
            cannotStop,
            cannotSuspend,
            cannotResume,
            cannotSetState,
            unknownState,
            cannotSetAccountingCycle,
            accountingCycleTooShort,
            cannotSuspendNotification,
            cannotResumeNotification,
            cannotFlushNotification,
```

```
        cannotSetVerbosityLevel,
        cannotSetNotificationDestination,
        unknownNotificationDestination,
        invalidSessionId,
        invalidServiceTransactionId,
        cannotResetNotificationDestination,
        cannotResetAllNotificationDestination
};

exception e_AmcObject {
    t_AmcObject_error error;
    string reason;
};

interface i_AccObjectManagement {
    /**
     * mimimum value of accounting cycle settable
     * by set_accounting_cycle
     **/
    readonly attribute TINAScsAmcCommon::t_Timeval minimum_cycle;

    // control operations
    boolean start() raises (e_AmcObject);
    boolean stop() raises (e_AmcObject);
    boolean suspend() raises (e_AmcObject);
    boolean resume() raises (e_AmcObject);
    boolean set_state(
        in any acc_object_state     // service specific.
    ) raises (e_AmcObject);
    boolean set_accounting_cycle (
        in TINAScsAmcCommon::t_Timeval cycle
    ) raises (e_AmcObject);

    // notification control operations
    boolean suspend_notification() raises (e_AmcObject);
    boolean resume_notification() raises (e_AmcObject);
    /**
     * flush all the events in the internal queue
     * to the notification destination.  This
     * operation may be necessary before the
     * object is taken out of event management
     * ladder and subsequently destroyed.
     **/
    boolean flush_notification() raises (e_AmcObject);
    boolean set_verbosity_level(
        in long level
    ) raises (e_AmcObject);

    /**
     * set destination of accounting events to be
     * notified, adding the destination to an
     * internal notification destination list.
     * Note that it is possible that a ladder
     * element may send its events to multiple
     * destinations, e. g. an SSM can send its
     * events to multiple USMs.
     **/
    boolean set_Notification_Destination(
        in Object destination,
            // destination interface
        in TINACommonTypes::t_SessionId sessionId,
            // Session ID, to which the ladder belongs to.
        in TINAScsAmcCommon::t_ServiceTransactionId serviceTransactionId
            // This argument is optional, set NULL when it
            // is not used.  When it is set, however, it
            // specifies which service transaction the
            // destination interface is associated with.
    ) raises (e_AmcObject);

    /**
     * reset destination of accounting events to be
```

```
         * notified.  The destination is removed from the
         * notification recipient list.
         **/
        boolean reset_Notification_Desination(
            in Object destination
        ) raises (e_AmcObject);
        /**
         * reset destinations, with which sessionId is associated.
         */
        boolean reset_Notification_sessionID(
            in TINACommonTypes::t_SessionId sessionId
        ) raises (e_AmcObject);
        /**
         * reset destinations, with which serviceTransactionId
         * is associated.
         **/
        boolean reset_Notification_serviceTransactionID(
            in TINAScsAmcCommon::t_ServiceTransactionId serviceTransactionId
        ) raises (e_AmcObject);

        /**
         * reset all the destination of accounting events,
         * and the destination list becomes empty.
         **/
        boolean reset_all_Notification_Desination(
        ) raises (e_AmcObject);
    };

    /**
     * element of event management ladder :
     * an accountable object with operations with
     * CosEventComm::PushConsumer interface.
     **/
    interface i_AmcLadderElement : CosEventComm::PushConsumer {
        // maintenance operations, if necessary.
    };

    /**
     * generic Pull interface for components with Pull
     * functionality (e.g. namedUA, SSM)
     **/
    interface i_AccountingPull {
        /**
         * same as i_nameduaBillingLog::getUserLogEntries
         **/
        boolean GetUserLogEntries(
            in TINACommonTypes::t_UserId userId,
            in TINAScsAmcCommon::t_DateTime from,
            in TINAScsAmcCommon::t_DateTime to,
            out TINAScsAmc::t_BillingEventList events
        ) raises (UMLogManager::e_UMLogOperation);

        /**
         * same as i_nameduaBillingLog::getSessionLogEntries
         **/
        boolean GetSessionLogEntries(
            in TINACommonTypes::t_SessionId sessionId,
            out TINAScsAmc::t_BillingEventList events
        ) raises (UMLogManager::e_UMLogOperation);
    };

    /**
     * generic Pull interface for components with Push
     * functionality (e.g. namedUA)
     **/
    interface i_AccountingPush : TINAScsAmcObject::i_AmcLadderElement {
        /**
         * same as i_nameduaBillingLog::store
         **/
//      boolean StoreBillingEvent(
//          in t_BillingEvent event
```

```
//      ) raises (UMLogManager::e_UMLogOperation);
        boolean StoreBillingEventList(
            in TINAScsAmc::t_BillingEventList events
        ) raises (UMLogManager::e_UMLogOperation);

        /**
         * same as i_nameduaBillingLog::remove
         **/
//      boolean RemoveBillingEvent(
//          in t_BillingEvent event
//      ) raises (UMLogManager::e_UMLogOperation);
        boolean RemoveBillingEventList(
            in TINAScsAmc::t_BillingEventList events
        ) raises (UMLogManager::e_UMLogOperation);

        /**
         * same as i_nameduaBillingLog::removeUserLogEntries
         **/
        boolean RemoveUserLogEntries(
            in TINAScsAmcCommon::t_DateTime from,
            in TINAScsAmcCommon::t_DateTime to
        ) raises (UMLogManager::e_UMLogOperation);
    };
};

#endif
```

2.5 TINAScsAmc

```
#ifndef _TINAScsAmc_idl_
#define _TINAScsAmc_idl_

//
//      accounting management common definitions
//      are included from corresponding NCS spec.
//

#include "TINACommonTypes.idl"
#include "TINAScsAmcCommon.idl"
#include "TINAScsMgmtCtxt.idl"

//
//      File Name: TINAScsAmc.idl
//      Decription: common definitions for accounting
//              management in SCS
//      Comments:
//              VITAL inputs were provided by
//              George Pavlou, UCL, UK.
//      Revision Histroy:
//          8-11-97 v0.1 by Takeo Hamada
//          8-26-97 v0.2 by Takeo Hamada
//          8-30-97 v0.3 by Takeo Hamada
//          9-10-97 v0.31 by Takeo Hamada
//              passed hidl compiler
//          9-17-97 v0.35 by Takeo Hamada
//              module structure revised, passed hidl
//              compiler.
//          9-25-97 v0.37 by Takeo Hamada
//              with new naming scheme
//

module TINAScsAmc {

    //
    //  Billing Unit : currencies in the world
    //

    /**
     * international currency acronym should be used.
```

```
 **/
typedef string<16> t_BillingUnit;

//  Examples of currency acronym
//          Ref. http://www.oanda.com, 164 currencies in the world
//
//  major TINA currencies in the world
//
//  AUD     : Australian Dollar
//  BEF     : Belgian Franc
//  GBP     : British Pound
//  CAD     : Canadian Dollar
//  CLP     : Chilean Peso
//  CNY     : Chinese Yuan Renminbi
//  DKK     : Danish Krone
//  DYD     : Disney Dollar
//  NLG     : Dutch Guilder
//  DEM     : Deutsche Mark
//  ECU     : ECU
//  FIM     : Finnish Markka
//  FRF     : French Franc
//  GRD     : Greek Drachma
//  ITL     : Italian Lira
//  JPY     : Japanese Yen
//  KRW     : Korean Won
//  MYR     : Malaysian Ringgit
//  NOK     : Norwegian Kroner
//  PTE     : Portuguese Escudo
//  SUR     : Russian Rouble
//  ESP     : Spanish Peseta
//  SEK     : Swedish Krona
//  CHF     : Swiss Franc
//  USD     : US Dollar
//
//  less known international currencies
//
//  AFA     : Afghanistan Afghani
//  ALL     : Albanian Lek
//  DZD     : Algerian Dinar
//  INR     : Indian Rupee
//  ILS     : Israeli New Shekel
//  XAU     : Gold (oz.)
//  PGK     : Papua New Guinea Kina
//  XPT     : Platinum (oz.)
//  VND     : Vietnamese Dong
//  etc.
//

/**
 *  Billing Status : status of billing information
 *
 *          @member inProcess service session is on-going.
 *          @member terrminated user is out of the service session,
 *          but service transaction not concluded.
 *  @member final correlation with PM done, service transaction
 *          concluded.
 **/
enum t_BillingStatus {
    inProcess,      // service session is on-going
    terminated,     // user is out of the service session,
            // but service transaction not concluded
    final   // correlation with PM done, service transaction
            // concluded
};

//
//  Billing Event Type
//

enum t_BillingEventType {
    // per session billing event types
```

```
        differential,   // differential billing form
        normal,         // normal billing for a period

        // periodic billing event types, which can be used
        // as reponses to billing query
        hourly,
        daily,
        weekly,
        monthly,
        bi_monthly,
        yearly
    };

    /**
     *  Billing Event definition
     *
     *          @member userId left blank when billing event
     *          is not per user.
     *          @member sessionId left blank when billing event
     *          is not per session.
     *          @member tariffId Id of tariff, from which billing
     *          info. is calculated.
     *          @member mgmtctxtId Management Context Id to which
     *          this service transaction
     *          follows
     **/
    struct t_BillingMeasurement {
        TINACommonTypes::t_UserId userId;
                        // left blank when billing event
                        // is not per user
        TINACommonTypes::t_SessionId sessionId;
                        // left blank when billing event
                        // is not per session
        TINAScsAmcCommon::t_TariffId tariffId;
                        // Id of tariff, from which billing
                        // info. is calculated
        TINAScsMgmtCtxt::t_MgmtCtxtID mgmtctxtId;
                        // Management Context Id to which
                        // this service transaction
                        // follows
        TINAScsAmcCommon::t_DateTime time;
        t_BillingStatus status;
        t_BillingEventType eventType;
        union eventData switch (t_BillingEventType) {
        case differential: TINAScsAmcCommon::t_Timeval tval;
        case normal: TINAScsAmcCommon::t_fromTo fromTo_normal;
        case hourly: TINAScsAmcCommon::t_fromTo fromTo_hourly;
        case daily: TINAScsAmcCommon::t_fromTo fromTo_daily;
        case weekly: TINAScsAmcCommon::t_fromTo fromTo_weekly;
        case monthly: TINAScsAmcCommon::t_fromTo fromTo_monthly;
        case bi_monthly: TINAScsAmcCommon::t_fromTo fromTo_bi_monthly;
        case yearly: TINAScsAmcCommon::t_fromTo fromTo_yearly;
        } event_data;
        t_BillingUnit billingUnit;
        double uvalue;  // is there any decimal arithmetic
                        // format?
        // the following solution is too ambiguous.
        // long uvalueA;        // uvalue above decimal point
        // long uvalueB;        // uvalue below decimal point
    };

    typedef sequence <t_BillingMeasurement> t_BillingSequence;

    /**
     *  Billing event definition.
     *
     *  @member time issue date of billing statement.
     *          @member seq billing event sequence.
     *          @member proof proof of origin attached by the
     *          originator of the event, when
     *          necessary.
```

```
     **/
    struct t_BillingEvent {
        TINAScsAmcCommon::t_DateTime time;
                        // issue date of billing statement
        t_BillingSequence seq;
        Security::Opaque proof;
                        // proof of origin attached by the
                        // originator of the event, when
                        // necessary.
    };

    typedef sequence <t_BillingEvent> t_BillingEventList;

};

#endif
```

2.6 TINAScsAmcTariff

```
#ifndef _TINAScsAmcTariff_idl_
#define _TINAScsAmcTariff_idl_

#ifdef debug
#include “PLATyToolsFix.idl”
#else
#include “CosTrading/CosTrading.idl”
#endif
#include “TINAScsAmc.idl”

//
//      File Name: TINAScsAmcTariff.idl
//      Decription: tariff definitions for SCS accounting
//      Comments:
//      Revision Histroy:
//          11-09-97 v0.1 by Takeo Hamada
//

//
//      Warning : The following tariff structure is given
//      more as an example; Arbitrary tariff structure can
//      be provided by Retailer, as far as tariffId (a string)
//      can be resolved to a tariff structure, from which
//      Billing info. can be calculated.
//
//      In a later stage of SCS development, however, federation
//      and composition will be considered, where common tariff
//      structure will be necessary, to generate composed tariffing
//      structure to the end-user as a part of a management context.
//

//
//      Tariff structure
//
//      Description: i_TariffStructure provides information on the
//      tariff structure, which gives sufficient information on the
//      pricing information Retailer/Communication Service Provider
//      will be charging to the user, upon the completion of the
//      service transaction.
//
//      A tariff structure is expressed by the following formula.
//      d, t represent amount of data and time, respectively.
//                    ___
//                    \
//      charge = A + /   ( B(QoS) * t + C(QoS) * d )
```

```
//                   ---
//                  over SFEPs
//
//      A : a fixed service access charge.  It could be included as
//          a part of subscription contract (e. g. monthly flat rate).
//          Therefore it is possible that A = 0, when the tariff
//          represents only the usage part.  This fixed charge covers
//          basic access to Ret reference point and associated kTN
//          usage.
//
//      B : per SFEP/per time charge.  Charging rate B is a
//          function of QoS schema to be used for SFEP and its
//          stream binding.
//
//      C : per SFEP/per data charge.  Charging rate C is a
//          function of QoS.
//

module TINAScsAmcTariff {

    enum t_FixedChargeType {
        perAccessSession,
        perServiceSession,
        hourlyFlat,
        dailyFlat,
        weeklyFlat,
        bi_weeklyFlat,
        monthlyFlat,
        two_montlyFlat,
        three_montlyFlat,
        half_yearlyFlat,
        yearlyFlat,
        two_yearlyFlat
    };

    enum t_Tariff_error {
        e_nonEffectiveQosSchema,
        e_priceStructureNotFound,
        e_tariffStructureNotAvailable
    };

    exception e_Tariff {
        t_Tariff_error error;
        string reason;
    };

    struct t_TariffPricing {
        // fixed charge corresponds to A
        double a;
        t_FixedChargeType fixedChargeType;

        // per time charge corresponds to B
        double b;
        // measurement time period : sec, min, etc.
        TINAScsAmcCommon::amc_TimePeriod perTime;

        // per data charge coresponds to C
        double c;
        // measurement data unit : Kbytes, Mbytes, etc.
        TINAScsAmcCommon::amc_UnitType perData;

        // monetary unit of pricing info. e.g. USD
        TINAScsAmc::t_BillingUnit punit;
    };
```

```
    typedef sequence <t_TariffPricing> t_TariffPricingSet;

    interface i_TariffStructure {
        boolean tariff_structure(
            in CosTrading::PropertySeq qos_schema,
                // name-value pairs which describes QoS schema
                // the list may also contain protocol name such
                // as RSVP.
            out t_TariffPricing tprice  // pricing info.
        ) raises (e_Tariff);

    };
};

#endif
```

2.7 TINAScsASUAPIntra

```
/* TINAScsASUAPIntra.idl
**
** Access Session User Application
**
** Author: Patrick Farley (BT)
** Reviewer: Carlo Licciardi (CSELT)
** Creation date: September 5th, 1997
** Review: Sept. 10th
*/

#ifndef TINAScsASUAPIntra_IDL
#define TINAScsASUAPIntra_IDL

#include "TINACommonTypes.idl"
#include "TINAUserAccess.idl"
#include "TINAAccessCommonTypes.idl"


module TINAScsASUAPIntra
{
/* draft YUCK, and is shared with ssUAP, so should be defined elsewhere. */
interface i_Init
{
};      /* interface i_Init     */


/* draft        */
interface i_Access {

        void cancelAccessSession(
                in TINAUserAccess::t_CancelAccessSessionProperties options
        );


        void inviteUser (
                in  TINAAccessCommonTypes::t_SessionInvitation invitation,
                out TINACommonTypes::t_InvitationReply reply
        );

        void cancelInviteUser (
                in TINAAccessCommonTypes::t_InvitationId id
        ) raises (
                TINAAccessCommonTypes::e_InvitationError
        );


oneway  void newSessionInfo (
                in TINAAccessCommonTypes::t_SessionInfo session
        );
```

```
oneway  void endSessionInfo (
                in TINACommonTypes::t_SessionId sessionId
        );

oneway  void endMyParticipationInfo (
                in TINACommonTypes::t_SessionId sessionId
        );

oneway  void suspendSessionInfo (
                in TINACommonTypes::t_SessionId sessionId
        );

oneway  void suspendMyParticipationInfo (
                in TINACommonTypes::t_SessionId sessionId
        );

oneway  void resumeSessionInfo (
                in TINAAccessCommonTypes::t_SessionInfo session
        );

oneway  void resumeMyParticipationInfo (
                in TINAAccessCommonTypes::t_SessionInfo session
        );

oneway  void joinSessionInfo (
                in TINAAccessCommonTypes::t_SessionInfo session
        );

};      /* interface i_Access   */


};      /* module TINAScsASUAPIntra     */

#endif  /* TINAScsASUAPIntra_IDL        */
```

2.8 TINAScsPAIntra

```
/* TINAScsPAIntra.idl
**
** Provider Agent
**
** Author: Patrick Farley (BT)
** Reviwers: Carlo Licciardi (CSELT)
** Creation date: August 25th, 1997
*/

#ifndef TINAScsPAIntra_IDL
#define TINAScsPAIntra_IDL

#include "TINAProviderInitial.idl"
#include "TINAProviderAccess.idl"
#include "TINAAccessCommonTypes.idl"
#include "TINACommonTypes.idl"
#include "TINAUserInitial.idl"
#include "TINAScsAmcObject.idl"

module TINAScsPAIntra
{

struct t_UserCtxtNameASId {
        TINACommonTypes::t_UserCtxtName name;
        TINAAccessCommonTypes::t_AccessSessionId accessSession;
};

typedef sequence<t_UserCtxtNameASId> t_UserCtxtNameASIdList;

typedef TINACommonTypes::t_PropertyList t_ProviderProperties;
```

```
enum t_ContactNotPossibleErrorCode {            /* draft        */
        InvalidProviderId
};


exception e_ContactNotPossible {                /* draft        */
        t_ContactNotPossibleErrorCode errorCode;
};



exception e_ProviderPropertiesError {
        TINACommonTypes::t_PropertyErrorStruct propertyError;
};



enum t_InvitationsOutsideAccessSessionErrorCode {
        InvitationIRNotAvailableForUserCtxt
};

exception e_InvitationsOutsideAccessSessionError {
        t_InvitationsOutsideAccessSessionErrorCode errorCode;
        TINACommonTypes::t_UserCtxtName ctxtName;
};


interface i_Init                /* inherit from a common i_Init interface? */
{

/* YUCK no idea what goes in here yet   */

};


interface i_Initial {


        void contactProvider (                  /* draft        */
                in TINAUserInitial::t_ProviderId providerId,
                in t_ProviderProperties providerProperties
        ) raises (
                e_ContactNotPossible,
                e_ProviderPropertiesError
        );


        void requestNamedAccess (
                in TINACommonTypes::t_UserId userId,
                in TINACommonTypes::t_UserProperties userProperties,
                out Object  PAaccessIR,
                out TINAAccessCommonTypes::t_AccessSessionId asId
        ) raises (
                TINAProviderInitial::e_AccessNotPossible,
                TINAAccessCommonTypes::e_UserPropertiesError
        );


        void requestAnonymousAccess (
                in TINACommonTypes::t_UserProperties userProperties,
                out Object  PAaccessIR,
                out TINAAccessCommonTypes::t_AccessSessionId asId
        ) raises (
                TINAProviderInitial::e_AccessNotPossible,
                TINAAccessCommonTypes::e_UserPropertiesError
        );

};      /* interface iInitial       */
```

```
interface i_Access {

        void registerInterface (
                in TINACommonTypes::t_InterfaceStruct itf,
                out TINACommonTypes::t_InterfaceIndex index
        ) raises (
                TINAAccessCommonTypes::e_AccessError,
                TINACommonTypes::e_InterfacesError,
                TINACommonTypes::e_RegisterError
        );

        void registerInterfaces (
                inout TINACommonTypes::t_RegisterInterfaceList itfs
        ) raises (
                TINAAccessCommonTypes::e_AccessError,
                TINACommonTypes::e_InterfacesError,
                TINACommonTypes::e_RegisterError
        );


                // register interfaces which will be accessible
                // outside the access session.

        void registerInterfaceOutsideAccessSession (
                in TINACommonTypes::t_InterfaceStruct itf,
                out TINACommonTypes::t_InterfaceIndex index
        ) raises (
                TINAAccessCommonTypes::e_AccessError,
                TINACommonTypes::e_InterfacesError,
                TINACommonTypes::e_RegisterError
        );


        void registerInterfacesOutsideAccessSession (
                inout TINACommonTypes::t_RegisterInterfaceList itfs
        ) raises (
                TINAAccessCommonTypes::e_AccessError,
                TINACommonTypes::e_InterfacesError,
                TINACommonTypes::e_RegisterError
        );

        void listRegisteredInterfaces (
                in TINAAccessCommonTypes::t_SpecifiedAccessSession as,
                out  TINACommonTypes::t_RegisterInterfaceList itfs
        ) raises (
                TINAAccessCommonTypes::e_AccessError,
                TINAAccessCommonTypes::e_SpecifiedAccessSessionError,
                TINACommonTypes::e_ListError
        );

         void unregisterInterface (
                in TINACommonTypes::t_InterfaceIndex index
        ) raises (
                TINAAccessCommonTypes::e_AccessError,
                TINACommonTypes::e_InterfacesError,
                TINACommonTypes::e_UnregisterError
        );

          void unregisterInterfaces (
                in TINACommonTypes::t_InterfaceIndexList indexes
        ) raises (
                TINAAccessCommonTypes::e_AccessError,
                TINACommonTypes::e_InterfacesError,
                TINACommonTypes::e_UnregisterError
        );

        void getUserCtxtNames (
                in TINAProviderAccess::t_SpecifiedUserCtxt ctxt,
                out TINACommonTypes::t_UserCtxtNameList userCtxts
        ) raises (
```

```
        TINAAccessCommonTypes::e_AccessError,
        TINAProviderAccess::e_UserCtxtError,
        TINACommonTypes::e_ListError
);

void getUserCtxtNamesAccessSessions (
        in TINAAccessCommonTypes::t_SpecifiedAccessSession as,
        out t_UserCtxtNameASIdList userCtxts
) raises (
        TINAAccessCommonTypes::e_AccessError,
        TINAAccessCommonTypes::e_SpecifiedAccessSessionError,
        TINACommonTypes::e_ListError
);

void listAccessSessions (
        out TINAAccessCommonTypes::t_AccessSessionList asList
) raises (
        TINAAccessCommonTypes::e_AccessError,
        TINACommonTypes::e_ListError
);

void endAccessSession(
        in TINAAccessCommonTypes::t_SpecifiedAccessSession as,
        in TINAProviderAccess::t_EndAccessSessionOption option
) raises (
        TINAAccessCommonTypes::e_AccessError,
        TINAAccessCommonTypes::e_SpecifiedAccessSessionError,
        TINAProviderAccess::e_EndAccessSessionError
);

void getUserInfo(
        out TINAAccessCommonTypes::t_UserInfo userInfo
) raises (
        TINAAccessCommonTypes::e_AccessError
);


void listSubscribedServices (
        in TINAProviderAccess::t_SubscribedServiceProperties desiredProperties,
        out TINAAccessCommonTypes::t_ServiceList services
) raises (
        TINAAccessCommonTypes::e_AccessError,
        TINACommonTypes::e_PropertyError,
        TINACommonTypes::e_ListError
);


void discoverServices(
        in TINAProviderAccess::t_DiscoverServiceProperties desiredProperties,
        in unsigned long howMany,
        out TINAAccessCommonTypes::t_ServiceList services,
        out Object iteratorIR   /* type: i_DiscoverServicesIterator */
) raises (
        TINAAccessCommonTypes::e_AccessError,
        TINACommonTypes::e_PropertyError,
        TINACommonTypes::e_ListError
);



void listServiceSessions (
        in TINAAccessCommonTypes::t_SpecifiedAccessSession as,
        in TINAProviderAccess::t_SessionSearchProperties desiredProperties,
        out TINAAccessCommonTypes::t_SessionList sessions
) raises (
        TINAAccessCommonTypes::e_AccessError,
        TINAAccessCommonTypes::e_SpecifiedAccessSessionError,
        TINACommonTypes::e_PropertyError,
        TINACommonTypes::e_ListError
);
```

```
void getSessionModels (
        in TINACommonTypes::t_SessionId sessionId,
        out TINACommonTypes::t_SessionModelList sessionModels
) raises (
        TINAAccessCommonTypes::e_AccessError,
        TINAProviderAccess::e_SessionError,
        TINACommonTypes::e_ListError
);

void getSessionInterfaceTypes (
        in TINACommonTypes::t_SessionId sessionId,
        out TINACommonTypes::t_InterfaceTypeList itfTypes
) raises (
        TINAAccessCommonTypes::e_AccessError,
        TINAProviderAccess::e_SessionError,
        TINACommonTypes::e_ListError
);

void getSessionInterface (
        in TINACommonTypes::t_SessionId sessionId,
        in TINACommonTypes::t_InterfaceTypeName itfType,
        out TINACommonTypes::t_InterfaceStruct itf
) raises (
        TINAAccessCommonTypes::e_AccessError,
        TINAProviderAccess::e_SessionError,
        TINACommonTypes::e_InterfacesError
);

void getSessionInterfaces (
        in TINACommonTypes::t_SessionId sessionId,
        out TINACommonTypes::t_InterfaceList itfs
) raises (
        TINAAccessCommonTypes::e_AccessError,
        TINAProviderAccess::e_SessionError,
        TINACommonTypes::e_ListError
);



void listSessionInvitations (
        out TINAAccessCommonTypes::t_InvitationList invitations
) raises (
        TINAAccessCommonTypes::e_AccessError,
        TINACommonTypes::e_ListError
);

void listSessionAnnouncements (
        in TINAProviderAccess::t_AnnouncementSearchProperties desiredProperties,
        out TINACommonTypes::t_AnnouncementList announcements
) raises (
        TINAAccessCommonTypes::e_AccessError,
        TINACommonTypes::e_PropertyError,
        TINACommonTypes::e_ListError
);

void startService (
        in TINAAccessCommonTypes::t_ServiceId serviceId,
        in TINAProviderAccess::t_ApplicationInfo app,
        in TINACommonTypes::t_SessionModelReq sessionModelReq,
        in TINAProviderAccess::t_StartServiceUAProperties uaProperties,
        in TINAProviderAccess::t_StartServiceSSProperties ssProperties,
        out TINAAccessCommonTypes::t_SessionInfo sessionInfo
) raises (
        TINAAccessCommonTypes::e_AccessError,
        TINAProviderAccess::e_ServiceError,
        TINAProviderAccess::e_ApplicationInfoError,
        TINACommonTypes::e_SessionModelError,
        TINAProviderAccess::e_StartServiceUAPropertyError,
        TINAProviderAccess::e_StartServiceSSPropertyError
);
```

```
/* draft        */
void startServiceWithUAP (
        in TINAAccessCommonTypes::t_ServiceId serviceId,
        in TINAProviderAccess::t_ApplicationInfo app
) raises (
        TINAAccessCommonTypes::e_AccessError,
        TINAProviderAccess::e_ServiceError,
        TINAProviderAccess::e_ApplicationInfoError
);


void endSession (
        in TINACommonTypes::t_SessionId sessionId
) raises (
        TINAAccessCommonTypes::e_AccessError,
        TINAProviderAccess::e_SessionError
);


void endMyParticipation (
        in TINACommonTypes::t_SessionId sessionId
) raises (
        TINAAccessCommonTypes::e_AccessError,
        TINAProviderAccess::e_SessionError
);

void suspendSession (
        in TINACommonTypes::t_SessionId sessionId
) raises (
        TINAAccessCommonTypes::e_AccessError,
        TINAProviderAccess::e_SessionError
);

void suspendMyParticipation (
        in TINACommonTypes::t_SessionId sessionId
) raises (
        TINAAccessCommonTypes::e_AccessError,
        TINAProviderAccess::e_SessionError
);

void resumeSession (
        in TINACommonTypes::t_SessionId sessionId,
        in TINAProviderAccess::t_ApplicationInfo app,
        out TINAAccessCommonTypes::t_SessionInfo sessionInfo
) raises (
        TINAAccessCommonTypes::e_AccessError,
        TINAProviderAccess::e_SessionError,
        TINAProviderAccess::e_ApplicationInfoError
);

void resumeMyParticipation (
        in TINACommonTypes::t_SessionId sessionId,
        in TINAProviderAccess::t_ApplicationInfo app,
        out TINAAccessCommonTypes::t_SessionInfo sessionInfo
) raises (
        TINAAccessCommonTypes::e_AccessError,
        TINAProviderAccess::e_SessionError,
        TINAProviderAccess::e_ApplicationInfoError
);

void joinSessionWithInvitation (
        in TINAAccessCommonTypes::t_InvitationId invitationId,
        in TINAProviderAccess::t_ApplicationInfo app,
        out TINAAccessCommonTypes::t_SessionInfo sessionInfo
) raises (
        TINAAccessCommonTypes::e_AccessError,
        TINAProviderAccess::e_SessionError,
        TINAAccessCommonTypes::e_InvitationError,
        TINAProviderAccess::e_ApplicationInfoError
);
```

```
        void joinSessionWithAnnouncement (
                in TINAAccessCommonTypes::t_AnnouncementId announcementId,
                in TINAProviderAccess::t_ApplicationInfo app,
                out TINAAccessCommonTypes::t_SessionInfo sessionInfo
        ) raises (
                TINAAccessCommonTypes::e_AccessError,
                TINAProviderAccess::e_SessionError,
                TINAProviderAccess::e_AnnouncementError,
                TINAProviderAccess::e_ApplicationInfoError
        );

        void replyToInvitation (
                in TINAAccessCommonTypes::t_InvitationId invitationId,
                in TINACommonTypes::t_InvitationReply reply
        ) raises (
                TINAAccessCommonTypes::e_AccessError,
                TINAAccessCommonTypes::e_InvitationError,
                TINACommonTypes::e_InvitationReplyError
        );


        void receiveInvitationsOutsideAccessSession (
                in TINAProviderAccess::t_SpecifiedUserCtxt ctxt
        ) raises (
                TINAAccessCommonTypes::e_AccessError,
                TINAProviderAccess::e_UserCtxtError,
                e_InvitationsOutsideAccessSessionError
        );


};      /* interface i_paAccess */


/* draft        */
interface i_AccountingPull : TINAScsAmcObject::i_AccountingPull
/* YUCK not yet defined.        */
{
};

};
#endif  /* PA_IDL       */
```

2.9 TINAScsNamedUAIntra

```
// TINAScsNamedUAIntra.idl
//
// Named User Agent
//
// Author: Chelo Abarca (Alcatel)
//         Carlo Licciardi (CSELT)
// Creation date: August 25th, 1997
// Reviewed: September 9th
// Reviewed: November 10th by Koki NAKASHIRO(HITACHI)

#ifndef TINAScsNamedUAIntra_IDL
#define TINAScsNamedUAIntra_IDL


#include "TINACommonTypes.idl"
#include "TINAAccessCommonTypes.idl"
#include "TINASubCommonTypes.idl"
#include "TINAScsSF.idl"
#include "TINAScsAmcObject.idl"
#include "TINAScsCommonTypes.idl"       /** by Koki **/

module TINAScsNamedUAIntra {

        /**
```

```
        This interface allows its clients to get a reference of interface *i_ProviderNamedA
ccess that is
         necessary for access session interaction to take *place.
        **/

interface i_Initial {
        exception e_setupFailure
        {
                string reason;
        };

        /**
        This operation allows its clients to get a reference of interface *i_ProviderNamedA
ccess for an access
        session and the access session identifiers.
        **/

        void setupAccessSession (
           in   TINAAccessCommonTypes::t_UserInfo              userinfo,
           out TINAAccessCommonTypes::t_AccessSessionId            asId,
           out TINAAccessCommonTypes::t_AccessSessionSecretId    asSecretId,
           out string                                              sIOR
        ) raises (
                e_setupFailure
        );

};      // i_Initial

        /**
        This interface allows its clients to give the namedUA the information necessary to
keep an
        updated list of sessions the corresponding user is participating in and the status
of both
        the session and the user's participation.
        **/

interface i_SessionInfo {

        /**
        It allows its clients to notify the namedUA of the suspension of the user's partici
pation in a
        service session, and provide it with a reference to an interface of a SF that will
be used for
        resuming the participation and the relevant accounting information for the suspende
d service session.
        **/
        void participationSuspended (
                in TINAScsSF::t_GlobalSessionId         globalSessionId,
                in TINACommonTypes::t_PartyId           partyId,
                in TINACommonTypes::t_InterfaceList     resumeIR,
                in any                                  AccountingInfo
        )
        raises (
                TINAScsCommonTypes::e_IdentifierError
        );

        /**
        It allows its clients to notify the namedUA of the end of the user's participation
in a
        service session, and provide it with the relevant accounting information for the en
ded service session.
        **/

        void participationEnded (
                in TINAScsSF::t_GlobalSessionId         globalSessionId,
                in TINACommonTypes::t_PartyId           partyId,
                in TINACommonTypes:: t_InterfaceList    resumeIR,
                in any                                  AccountingInfo
        )
        raises (
                TINAScsCommonTypes::e_IdentifierError
```

```
        );

        /**
        It allows its clients to notify the namedUA of the suspension of a service session
where the
        user was taking part, and provide it with a reference to an interface of a SF that
will be used
        for resuming the service session and the relevant accounting information for the en
ded service session.
        **/
        void sessionSuspended (
                in TINAScsSF::t_GlobalSessionId         globalSessionId,
                in TINACommonTypes::t_PartyId           partyId,
                in TINACommonTypes:: t_InterfaceList    resumeIR,
                in any                                  AccountingInfo
        )
        raises (
                TINAScsCommonTypes::e_IdentifierError
        );

        /**
        It allows its clients to notify the namedUA of the suspension of a service session
where the
        user was taking part, and provide it with a reference to an interface of a SF that
will be used
        for resuming the service session and the relevant accounting information for the en
ded service session.
        **/

        void sessionEnded (
                in TINAScsSF::t_GlobalSessionId         globalSessionId,
                in TINACommonTypes::t_PartyId           partyId,
                in any                                  AccountingInfo
        )
        raises (
                TINAScsCommonTypes::e_IdentifierError
        );

        /**
        It allows its clients to notify the namedUA that a previously suspended service ses
sion,
        where the user was taking part, has been resumed.
        **/
        void sessionResumed (
                in TINACommonTypes::t_PartyId           partyId,
                in TINAScsSF::t_GlobalSessionId GlobalSessionId
        )
        raises (
                TINAScsCommonTypes::e_IdentifierError
        );

};      //i_SessionInfo


        /**
        This interface allows its clients to send invitations to the namedUA's user or canc
el them.
        **/
interface i_InvitationDelivery {

        /**
        It allows its clients to send an invitation to the service session to the namedUA's

        corresponding user. An identifier for the service the user is being invited to join
 a
        session of, as well as the name of the inviting party, the purpose of the session,
        and the reason for the invitation, are included in the invocation to allow for diff
erent
        invitation screening policies. An invitation identifier unique for this user and pr
ovider is
        given as well.
```

```
        **/

        void invite (
                in TINAAccessCommonTypes::t_SessionInvitation   invitation,
                out TINACommonTypes::t_InvitationReply          reply
        ) raises (
                TINAScsCommonTypes::e_UserIdError
        );

        /**
        allows its clients to cancel an invitation previously issued for reasons like the e
nd
        of the service session before the invited user joins; the unique invitation identif
ier
         is passed to identify the invitation to be cancelled.
        **/
        void cancel (
                in TINACommonTypes::t_UserId            userId,
                in TINAAccessCommonTypes::t_InvitationId inviteId
        ) raises (
                TINAScsCommonTypes::e_UserIdError
        );

}; //i_InvitationDelivery


        /**
        This interface allows its clients to notify the namedUA about new, modified or with
drawn services
        in the portfolio of the namedUA's corresponding user.
        **/

interface i_SubscriptionNotify {

        /**
        It allows its clients to notify the namedUA of new services added to the user's por
tfolio,
        or modifications or withdrawal of existing ones. It includes the list of services t
o which
        the notification refers and their corresponding service profiles.
        **/
        void notify (
                in TINACommonTypes::t_UserId    userId,
                in TINASubCommonTypes::t_subNotificationType    notificationType,
                in TINASubCommonTypes::t_ServiceIdList  serviceList,
                in TINASubCommonTypes::t_SagServiceProfileList serviceProfileList
        ) raises (
                TINAScsCommonTypes::e_UserIdError
        );

};  //i_SubscriptionNotify

        /**
        It allows its client to retrieve accounting information corresponding to a
        time interval or to a particular user session.
        **/

interface i_AccountingPull : TINAScsAmcObject::i_AccountingPull {

};      //i_AccountingPull

        /**
        It allows the namedUA to receive accounting events from the USM/SSM.
        These events derives in accounting records that can be used for
        (offline or online) billing.
        **/

interface i_AccountingPush : TINAScsAmcObject::i_AccountingPush {
};
```

```
interface i_Init
{

};


        /**
        It allows the customization of user service profiles. The user service profile
        includes a service part, describing customized service characteristics, and a servi
ce
        management part, detailing the management contexts for the different management are
as (FCAPS).
        **/
interface i_ServiceProfileCustomization
{

};

        /**
        It allows to manage the user profile. This profile contains information like:
        usage context, defining the user location (and/or terminal) registration and local
        context in every location (and/
or terminal), and personal configuration, like invitation
        handling policies and registration schedule. The usage context allows the client to
 locate the
        user and know about the context (terminal type, NAP type, and available service cap
abilities) in
        the current location. The personal preferences allow to model the behaviour of serv
ice components
        in access and service sessions depending on certain context conditions (time, date,
 location,
         session owner or participants, etc).
        **/

interface i_UsrProfileManagement
{

};

};

#endif // TINAScsNamedUAIntra_IDL
```

2.10 TINAScsServiceContractInfoAccess

```
//
// File: TINAScsServiceContractInfoAccess.idl
// Author: Juan C. Garcia (TINA CoreTeam - Telefonica I+D)
// Last modification: Aug 27, 97
// Contents:
//
// MODIFICATIONS:Sep. 24th 1997 by Hiroshi
//               Oct. 28th 1997 by Juan C.Garcia update according to new
//                              SCS release.
//
// IDL INTERFACES
// _____________________________ END DESCRIPTION HEADER ____________________________

#ifndef TINAScsServiceContractInfoAccess_IDL
#define TINAScsServiceContractInfoAccess_IDL

#include "TINASubCommonTypes.idl"

module TINAScsServiceContractInfoAccess {

/**
 *  i_ServiceContractInfoMgmt
 *
 *  Behavior:
```

```
 *     This interface is used to manage the information related to a
 *     service contract. This includes the subscription profile (service
 *     profile by default for all users) and a set of SAG service profiles
 *     (service profiles for users belonging to a specific SAG).
 *     It provides operations for:
 *     - retrieving the service template.
 *     - defining, modifying and retrieving service contract information.
 *       This information includes the associated service profiles.
 **/

interface i_ServiceContractInfoMgmt {

exception e_applicationError{};
exception e_invalidContractInfo{};
exception e_invalidSubscriptionProfile{};
exception e_invalidSAGServiceProfile{
          TINASubCommonTypes::t_ServiceProfileId     spId;
};
exception e_unknownServiceProfile{
          TINASubCommonTypes::t_ServiceProfileId     spId;
};
exception e_unknownSAG{
          TINASubCommonTypes::t_SagId     sagId;
};
exception e_unknownSAE{
          TINASubCommonTypes::t_entityId  saeId;
};

        /** This operation returns the template for the service.
         **/
        void getServiceTemplate (
                out TINASubCommonTypes::t_ServiceTemplate  template
        ) raises (e_applicationError);

        /** This operation creates a service contract.
         * This contract can include a set of service profiles.
         **/
        void defineServiceContract (
                in TINASubCommonTypes::t_ServiceContract        serviceContract,
                out TINASubCommonTypes::t_ServiceProfileIdList  spIdList
        )
        raises (e_applicationError,
                e_invalidContractInfo,
                e_invalidSubscriptionProfile,
                e_invalidSAGServiceProfile);

        /** This operation creates a set of service profiles.
         **/
        void defineServiceProfiles (
                in TINASubCommonTypes::t_SubscriptionProfile        subscriptionProfile,
                in TINASubCommonTypes::t_SagServiceProfileList  sagServiceProfiles,
                out TINASubCommonTypes::t_ServiceProfileIdList  spIdList
        )
        raises (e_applicationError,
                e_invalidSubscriptionProfile,
                e_invalidSAGServiceProfile);

        /** This operation deletes a set of service profiles and their
         * associated SAGs.
         **/
        void deleteServiceProfiles (
                in TINASubCommonTypes::t_ServiceProfileIdList  spIdList
        )
        raises (e_applicationError,
                e_unknownServiceProfile);

        /** This operation returns the list of service profiles identifiers
         **/
        void listServiceProfiles (
                out TINASubCommonTypes::t_ServiceProfileIdList  spIdList
        )
```

```
        raises (e_applicationError);

        /** This operation returns the service contract information.
         * If a (list of) SAG(s) is specified it returns the set of
         * SAG service profile for that(those) SAG(s).
         **/
        void getServiceContractInfo (
                in  TINASubCommonTypes::t_ServiceProfileIdList                spIdList,
                out TINASubCommonTypes::t_ServiceContract          serviceContract
        )
        raises (e_applicationError,
                e_unknownServiceProfile);

        /** This operation assigns a service profile to a list of SAGs and
         * SAEs.
         **/
        void assignServiceProfile (
                in TINASubCommonTypes::t_ServiceProfileId  spId,
                in TINASubCommonTypes::t_SagIdList  sagIdList,
                in TINASubCommonTypes::t_entityIdList  saeIdList
        )
        raises (e_applicationError,
                e_unknownSAG,
                e_unknownSAE,
                e_unknownServiceProfile);

        /** This operation removes a service profile assignment to a list
         * of SAGs and SAEs.
         **/
        void removeServiceProfile (
                in TINASubCommonTypes::t_ServiceProfileId  spId,
                in TINASubCommonTypes::t_SagIdList  sagIdList,
                in TINASubCommonTypes::t_entityIdList  saeIdList
        )
        raises (e_applicationError,
                e_unknownSAG,
                e_unknownSAE,
                e_unknownServiceProfile);

        /** This operation activates a set of service profiles
         **/
        void activateServiceProfiles (
                in TINASubCommonTypes::t_ServiceProfileIdList  spIdList
        )
        raises (e_applicationError,
                e_unknownServiceProfile);

        /** This operation deactivates a set of service profiles
         **/
        void deactivateServiceProfiles (
                in TINASubCommonTypes::t_ServiceProfileIdList  spIdList
        )
        raises (e_applicationError,
                e_unknownServiceProfile);

}; // Interface i_ServiceContractInfoMgmt

/**
 *  i_ServiceContractInfoQuery
 *
 *  Behavior:
 *     This interface is used to retrieve the service profiles related to the
 *     service contract.
 *     It is used by the Subscriber Manager (SubM).
 **/

interface i_ServiceContractInfoQuery {

exception e_applicationError{};
exception e_unknownServiceProfile{
          TINASubCommonTypes::t_ServiceProfileId     spId;
```

```
};

        /** This operation returns the service profiles associated to
         * the service contract.
         * If a (list of) SAG(s) is specified it returns the set of
         * SAG service profile for that(those) SAG(s).
         **/
        void getServiceProfiles (
                in  TINASubCommonTypes::t_ServiceProfileIdList                spIdList,
                out TINASubCommonTypes::t_ServiceProfileList        serviceProfileList
        )
        raises (e_applicationError,
                e_unknownServiceProfile);

}; // Interface i_ServiceContractInfoQuery


};

#endif // TINAScsServiceContractInfoAccess_IDL
```

2.11 TINAScsServiceContractMgmt

```
//
// File: TINAScsServiceContractMgmt.idl
// Author: Juan C. Garcia (TINA CoreTeam - Telefonica I+D)
// Last modification: Aug 27, 97
// Contents:
//
// MODIFICATIONS:Sep. 24th 1997 by Hiroshi
//
// IDL INTERFACES
// _____________________________ END DESCRIPTION HEADER ____________________________

#ifndef TINAScsServiceContractMgmt_IDL
#define TINAScsServiceContractMgmt_IDL

#include "TINASubCommonTypes.idl"

/**
 *  Module Service Contract Management.
 *
 **/
module TINAScsServiceContractMgmt {


/**
 *  Interface: i_ServiceContractLCMgmt
 *
 *  Behavior:
 *   It allows the lifecycle management of service contracts.
 *   It is used by the Subscription Coordinator (SCoo).
 **/

interface i_ServiceContractLCManagement {

        // TBD
        void createServiceContracts ();

        // TBD
        void deleteServiceContracts ();

        // TBD
        void listServiceContracts ();


}; // Interface i_ServiceContractLCManagement
```

```
};

#endif // TINAScsServiceContractMgmt_IDL
```

2.12 TINASubCommonTypes

```
//
// File TINASubCommonTypes.idl
// Author: Juan C. Garcia
// Inputs: - previous TINA-C specifications (March, Oct '95).
//         - VITAL V2 specifications.
//         - TINA Service Architecture 5.0.
//         - TINA Ret-RP Specifications.
// Contents:
// This file include common definitions required by the subscription
// magement component and its clients.
// @see TINACommonTypes
// @see TINAAccessCommonTypes
// History: Modified on 9/29/97 Juan C. Garcia (t_ServiceProfile definition)
//          Modified on 10/27/97 Juan C. Garcia
//                according to SCS document.
//

#ifndef TINASUBCOMMONTYPES_IDL
#define TINASUBCOMMONTYPES_IDL

//
// This file provides definitions that are common to the service architecture.
//
#include "TINAAccessCommonTypes.idl"

/**
 * Module with common types definitions for subscription management.
 **/
module TINASubCommonTypes {

/**
 * List of Service Identifiers.
 **/
typedef sequence<TINAAccessCommonTypes::t_ServiceId> t_ServiceIdList;

/**
 * Service Types
 **/
typedef string t_ServiceType;

/**
 * Terminal Type: Just an example.
 **/
enum t_TermType {UndefinedTermType,PersonalComputer, WorkStation, TVset, Videotelephone, Ce
llularphone, PBX, VideoServer, VideoBridge, Telephone, G4Fax};

/**
 * NAP type: used to determine the instantation of available QoS.
 **/
enum t_NapType {UndefinedNapType,NapTypeFixed, NapTypeWireless};

/**
 * List of NAPs
 **/
typedef sequence<TINAAccessCommonTypes::t_NAPId> t_NAPIdList;

/**
 *Terminal presentation technology. This is just an example
 **/
enum t_PresentationSupport { UndefinedPresSupp,X11R6, WINDOWS95, MGEG };

/**
 * The Account Number represents the Subscriber identifier.
```

```
 **/
typedef string t_AccountNumber;
typedef sequence<t_AccountNumber> t_SubscriberIdList;

// Types required for SAG management (through i_subSMSubscriber).
//
/**
 * Users, terminals and NAPs are considered (subscription) entities
 **/
enum t_entityType {user, terminal, nap};

/**
 * Entity Id allows to identity uniquely an entity inside the retailer
 * domain.
 **/
union t_entityId switch (t_entityType) {
        case user:     TINACommonTypes::t_UserId            userId;
        case terminal: TINAAccessCommonTypes::t_TerminalId  terminalId;
        case nap:      TINAAccessCommonTypes::t_NAPId       napId;
};
typedef sequence<t_entityId> t_entityIdList;

/**
 * An entity is characterized by its identifier and name and a set
 * of properties.
 **/
struct t_Entity {
        t_entityId                      entityId;
        string                          entityName;
        TINACommonTypes::t_PropertyList   properties;
};
/**
 * List of entities.
 **/
typedef sequence<t_Entity> t_EntityList;

/**
 * The SAE is characterized by an identifier, a name and a set of properties
 **/
struct t_Sae {
        t_entityId       entityId;
        string           entityName;
        TINACommonTypes::t_PropertyList   properties; // like password
};

/**
 * The SAG identifier identifies a SAG uniquely inside the retailer domain.
 **/
typedef short    t_SagId;
/**
 * List of SAG Ids.
 **/
typedef sequence<t_SagId> t_SagIdList;
/**
 * A SAG is characterized by its identifier, a textual description of the
 * group and the list of entities composing it.
 * The identifier is the same as the one for SAG Service profile corresponding
 * to that SAG.
 **/
struct t_Sag {
        t_SagId          sagId;
        string           sagDescription;
        t_entityIdList   entityList;
};
/**
 * List of SAGs.
 **/
typedef sequence<t_Sag>   t_SagList;

// Subscriber Information:
/**
```

```
 * Time and Date.
 **/
struct t_DateTime {
        string          date;
        string          time;
};
/**
 * Textual identification of a person. For example, name, address and
 * position.
 **/
typedef string       t_Person;
/**
 * Indicates the date and time an authorization expires on and the
 * person who granted it.
 **/
struct t_AuthLimit {
        t_DateTime      limitDate;
        string          authority;
}; // Shouldn't this be per service contract?

/**
 * A subscriber is identified by its account number and characterized by
 * name, address, monthly charge, payment record, credit information,
 * date which its subscription expires on, the list of subscribed services
 * and the list of defined SAGs.
 **/
struct t_Subscriber {
        t_AccountNumber          accountNumber;
        TINACommonTypes::t_UserId subscriberName;
        t_Person                 identificationInfo;
        t_Person                 billingContactPoint;
        string                   RatePlan;
        any                      paymentRecord;
        any                      credit;
};

/**
 * List of Subscribers.
 **/
typedef sequence<t_Subscriber> t_SubscriberList;

/**
 * This structure contains information about the minimal required configuration
 * of a service. This is used to specify a configuration for a particular
 * service session
 **/
struct t_RequiredConfiguration {
        t_TermType termType;
        t_NapType nap_type;
        t_PresentationSupport presentation_support;
        TINACommonTypes::t_PropertyList others; // to be determined.
};

/**
 * Service access rights.
 **/
enum t_AccessRight {create,join,be_invited};
/**
 * List of possible service access rights.
 **/
typedef sequence<t_AccessRight> t_AccessRightList;

/**
 **/
//struct t_ServiceInfo {
//      TINAAccessCommonTypes::t_ServiceId   serviceTypeId;
//      t_RequiredConfiguration              reqConf;
//      t_AccessRightList                    accessRightList;
//};
//typedef sequence<t_ServiceInfo> t_ServiceInfoList;
```

```
/**
 * Service Parameter name.
 **/
typedef string  t_ParameterName;

/**
 * Service Parameter configurability.
 **/
enum t_ParameterConfigurability {
    FIXED_BY_PROVIDER, CONFIGURABLE_BY_SUBSCRIBER, CUSTOMIZABLE_BY_USER
};

/**
 * Service Parameter value.
 **/
typedef any     t_ParameterValue;

/**
 * Service Parameters definition:
 **/
struct t_Parameter {
    t_ParameterName             name;
    t_ParameterConfigurability  configurability;
    t_ParameterValue            value;
};
/**
 * Service Parameter list:
 **/
typedef sequence<t_Parameter> t_ParameterList;

/**
 * Service Description: this type is used to describe a specific
 * service. It is used for the description of service types, service
 * instances and service profiles.
 * An example of service common params could be:
 * parameterName = ACCESS RIGHTS
 * configurability = CONFIGURABLE_BY_SUBSCRIBER
 * parameterValue = {join, be_invited}
 *
 * parameterName = REQUIRED TERMINAL TYPE
 * configurability = FIXED_BY_PROVIDER
 * parameterValue = {PersonalComputer, WorkStation}
 *
 * parameterName = REQUIRED PRESENTATION SUPPORT
 * configurability = FIXED_BY_PROVIDER
 * parameterValue = {WINDOWS 95, X11R6}
 *
 * An example of service specific params could be:
 * parameterName = MAX NUMBER OF PARTIES IN SESSION
 * configurability = FIXED_BY_PROVIDER
 * parameterValue = 20
 *
 * parameterName = TIME LIMIT FOR SUSPENDED SESSIONS
 * configurability = CONFIGURABLE_BY_SUBSCRIBER
 * parameterValue = 1 WEEK
 *
 * parameterName = AUTOMATIC POSITIVE ANSWER TO VOTINGS
 * configurability = CUSTOMIZABLE_BY_USER
 * parameterValue = YES
 *
 * The list of parameter names and value types that is common to
 * all services will be defined in a separate file:
 *     TINAServiceCommonParameters.idl
 *
 * The list of parameter names and value types specific to particular
 * service will be defined in a separate file:
 *     TINAServiceSpecificParameters.idl
 *
 **/
struct t_ServiceDescription {
       TINAAccessCommonTypes::t_ServiceId              serviceId;
```

```
        TINAAccessCommonTypes::t_UserServiceName        serviceName;
        t_ParameterList                                 serviceCommonParams;
        t_ParameterList                                 serviceSpecificParams;
};

/**
 * t_serviceTemplate: It describes a service instance.
 **/
struct t_ServiceTemplate {
        TINAAccessCommonTypes::t_ServiceId         serviceInstanceId;
        TINAAccessCommonTypes::t_UserServiceName   serviceInstanceName;
        t_ServiceIdList                            requiredServices;
        t_ServiceDescription                       serviceDescription;
};
typedef sequence<t_ServiceTemplate> t_ServiceTemplateList;

/**
 * t_ServiceProfile: It describes a service customization.
 **/
typedef string t_ServiceProfileId;
typedef sequence<t_ServiceProfileId> t_ServiceProfileIdList;
struct t_ServiceProfile {
        t_ServiceProfileId                         spId;
        t_ServiceDescription                       serviceDescription;
};
typedef sequence<t_ServiceProfile> t_ServiceProfileList;

/**
 * Service Profile for a SAG.
 **/
typedef t_ServiceProfile t_SagServiceProfile;

/**
 * Service Profile by default in a Service Contract.
 **/
typedef t_ServiceProfile t_SubscriptionProfile;

/**
 * List of SAG Service Profiles
 **/
typedef sequence<t_SagServiceProfile> t_SagServiceProfileList;

/**
 * Service Contract: Describes the relationship of a subscriber with the
 * provider for the provision of a service.
 **/
struct t_ServiceContract {
        TINAAccessCommonTypes::t_ServiceId  serviceId;
        t_AccountNumber                    accountNumber;
        short                              maxNumOfServiceProfiles;
        t_DateTime                         actualStart;
        t_DateTime                         requestedStart;
        t_Person                           requester;
        t_Person                           technicalContactPoint;
        t_AuthLimit                        authorityLimit;
        t_SubscriptionProfile              subscriptionProfile;
        t_SagServiceProfileList            sagServiceProfileList;
};


/**
 * Notification Type, used in "i_SubscriptionNotify::notify"
 **/
enum t_subNotificationType {NEW_SERVICES,PROFILE_MODIFIED,SERVICES_WITHDRAWN};

/**
 * Notification Type, used in "i_ServiceNotify::notify"
 **/
enum t_slcmNotificationType {NEW_SERVICE,TEMPLATE_MODIFIED,SERVICE_WITHDRAWN};

};
```

```
#endif // File TINASubCommonTypes.idl
```

2.13 TINAScsSubInitial

```
//
// File: TINAScsSubscriberInitial.idl
// Author: Juan C. Garcia (TINA CoreTeam - Telefonica I+D)
// Last modification: Aug 27, 97
// Contents:
// This file describes the main interfaces in the subscription component:
// - Initial: allows clients to get their access interfaces.
// - Subscribe: allows customers to subscribe to the retailer and
//   contract services.
// - Notify: allows SLCM to notify SUB about new services available in
//   the network or modifications
//   in the existing ones.
//
// MODIFICATIONS:Sep. 24th 1997 by Hiroshi
//
// IDL INTERFACES
// _____________________________ END DESCRIPTION HEADER ____________________________

#ifndef TINAScsSUBInitial_IDL
#define TINAScsSUBInitial_IDL

#include "TINASubCommonTypes.idl"

/** Module TINA Subscription Initial
**/
module TINAScsSubInitial {

/**
 *  Interface: i_InitialAccess
 *
 *  Behavior:
 *     This interface allows the SUB client to retrieve the
 *     appropiate interfaces for subscription management.
 *     Depending on the client type it returns:
 *     - If client=UA     -> i_SubscriberInfoQuery
 *     - If client=SSMols -> i_Subscribe
**/

interface i_InitialAccess
{
enum t_ErrorType { UnknownEntityId, InvalidClientType, InvalidTerminalConfig};
exception e_initError{
     t_ErrorType  errorType;
};
exception e_unknownEntityId{ };
exception e_applicationError{};

enum t_ClientType { UA, SM, SLCM };

        /**
         * It allows the client to get the appropiate interfaces to access the
         * Subscription component. These interfaces may exist before this
         * operation is invoked or may be created on demand. This will
         * depend on the implementation criteria.
         **/
        void init (
                in TINASubCommonTypes::t_entityId          entityId,
                in t_ClientType                            clientType,
                in TINAAccessCommonTypes::t_TerminalConfig termConfiguration,
                out TINACommonTypes::t_InterfaceList       subInterfaceList
        ) raises ( e_initError,
                   e_applicationError );

        /**
         * It allows the client to inform the server that he will no longer use the
```

```
         * interfaces. If convenient, the server will release the resources (delete interfa
ces
         * or objects) that were allocated or assigned in the init operation.
         **/
        void terminate (
                in TINASubCommonTypes::t_entityId          entityId,
                in t_ClientType                            clientType,
                in TINACommonTypes::t_InterfaceList        subInterfaceList
        )
        raises ( e_unknownEntityId,
                 e_applicationError );
}; // Interface i_InitialAccess


/**
 *  Interface: i_Subscribe
 *
 *  Behavior:
**/

interface i_Subscribe {

exception e_unknownServiceId{
      TINAAccessCommonTypes::t_ServiceId serviceId;
};
exception e_unknownSubscriber{
      TINASubCommonTypes::t_AccountNumber       subscriberId;
};
exception e_invalidSearchCriteria{};
exception e_notSubscribedService{
          TINAAccessCommonTypes::t_ServiceId serviceId;
};
exception e_applicationError{};
exception e_invalidSubscriberInfo{};
exception e_invalidServiceId{};

        /** This operation returns the list of subscribers matching a
         * search criteria (ALL to retrieve all the subscribers).
         * It can only be used by an authorized user (retailer operator)
         * If a service Id is specified other than the NULL one (0), only
         *  subscribers to the specified service are returned.
         * Note: the way to express the search criteria is to be defined.
         * SQL-like statements will be used. Parameters to use in the search are TBD.
         **/
        void listSubscribers (
                in string                                searchCriteria,
                in TINAAccessCommonTypes::t_ServiceId    serviceId,
                out TINASubCommonTypes::t_SubscriberIdList subscriberList
        )
        raises (e_unknownServiceId,
                e_invalidSearchCriteria,
                e_applicationError);

        /** This operation returns the subscription management interface
         * references for a specific subscriber. If a service id list is
         * given, only the service contract management interfaces for those
         * services are returned. The subscriber information management
         *  interface (i_subSMSubscribe) is always returned.
         **/
        void getReferences (
                in TINASubCommonTypes::t_AccountNumber     subscriber,
                in TINASubCommonTypes::t_ServiceIdList     serviceList,
                out TINACommonTypes::t_InterfaceList       interfaceList
        ) raises (e_unknownSubscriber,
                  e_notSubscribedService,
                  e_applicationError);

        /** This operation returns the list of services available for
         *  subscription and use.
         **/
        void listServices (
```

```
            out TINAAccessCommonTypes::t_ServiceList      serviceList
        ) raises (e_applicationError);

        /** This operation creates a subscription for a new customer.
         * The initial list of services the subscriber wants to contract
         * can be specified.
         * It returns:
         * - a unique identifier for the subscriber.
         * - an interface for subscriber information management.
         * - a list of service contract management interfaces, one for
         *   each of the services initially contracted.
         *  These two last items are included in the t_InterfaceList structure.
         **/
        void subscribe (
                in TINASubCommonTypes::t_Subscriber            subscriberInfo,
                in TINASubCommonTypes::t_ServiceIdList         serviceList,
                out TINASubCommonTypes::t_AccountNumber        subscriberId,
                out TINACommonTypes::t_InterfaceList       interfaceList
        ) raises (e_invalidSubscriberInfo,
                  e_unknownServiceId,
                  e_applicationError);

        /** This operation creates a (set of) new service contract(s) for
         * an existing customer.
         * A list of services the subscriber wants to contract
         * is specified.
         * It returns a list of service contract management interfaces,
         *  one for each of the services requested.
         **/
        void contractService (
                in TINASubCommonTypes::t_AccountNumber         subscriberId,
                in TINASubCommonTypes::t_ServiceIdList         serviceList,
                out TINACommonTypes::t_InterfaceList       interfaceList
        ) raises (e_unknownSubscriber,
                  e_unknownServiceId,
                  e_applicationError);

        /** This operation withdraws a subscription or a list of service
         * contracts.
         * The list of services the subscriber wants to unsubscribe
         * is an input parameter. If this list is empty, that means
         *  the withdrawal of all the services, and thus the subscription.
         **/
        void unsubscribe (
                in TINASubCommonTypes::t_AccountNumber         subscriberId,
                in TINASubCommonTypes::t_ServiceIdList         serviceList
        ) raises (e_unknownSubscriber,
                  e_unknownServiceId,
                  e_applicationError);

}; // Interface i_Subscribe


/**
 *  Interface: i_ServiceNotify
 *
 *  Behavior: It is used by the SLCM to notify Sub about new services
 *  deployed and available for subscription and use or about changes
 *  (/withdrawals) of already deployed ones.
**/

interface i_ServiceNotify {

        /** It indicates a change in the services available for subscription
         *  and use.
         **/
        void notify ();

}; // Interface i_ServiceNotify

};
```

```
#endif // TINAScsSUBInitial_IDL
```

2.14 TINAScsSubscriberInfoAccess

```
//
// File: TINAScsSubscriberInfoAccess.idl
// Author: Juan C. Garcia (TINA CoreTeam - Telefonica I+D)
// Last modification: Aug 27, 97
// Contents:
// This file describes the interfaces corresponding to subscriber
// information management:
// - UA Query Info: allows clients to get their access interfaces.
// - Subscriber: allows customers to query and update subscriber information.
//
// MODIFICATIONS:
//   Sep. 24th 1997 by Hiroshi
//   Sep. 29th 1997 by Juan C. Garcia (subscriber Id added as parameter in
//                                     subscriber management operations)
//   Oct 28th  1997 by Juan C. Garcia made consistent with SCS release (28/10)
//
// IDL INTERFACES
// _____________________________ END DESCRIPTION HEADER _____________________

#ifndef TINAScsSubscriberInfoAccess_IDL
#define TINAScsSubscriberInfoAccess_IDL

#include "TINASubCommonTypes.idl"

/**
 * Module TINA Subscriber Information Access
**/
module TINAScsSubscriberInfoAccess {

/**
 *  i_SubscriberInfoMgmt
 *
 *  Behavior:
 *     This interface allows the management of the information related
 *     to a particular subscriber.
 *     This interface can be used either by the subscriber or by
 *     the retailer operator on behalf of the subscriber.
 *     It provides operations for:
 *     - creating and deleting entities.
 *     - creating SAGs.
 *     - assigning/removing entities to/from a SAG.
 *     - listing entities and SAGs corresponding to that subscriber.
 *     - querying and modifying the subscriber information.
**/

interface i_SubscriberInfoMgmt {

enum t_errorType { NameTooLong, AddressTooLong };
exception e_invalidSubscriberInfo{
      t_errorType errorType;
};
exception e_unknownSubscriber{
      TINASubCommonTypes::t_AccountNumber       subscriberId;
};
exception e_applicationError{};
exception e_invalidEntityInfo{};
exception e_unknownSAE{
      TINASubCommonTypes::t_entityId    entityId;
};
exception e_unknownSAG{
      TINASubCommonTypes::t_SagId       sagId;
};
exception e_invalidSAG{
      TINASubCommonTypes::t_SagId       sagId;
```

```
};

        /** This operation creates a set of entities. If the client does
         * not specify an identifier for an entity or the specified identifier
         * coincides with an existing one, Sub generates and returns an
         * identifier for that entity. In other case, it returns the same
         * identifier that has received.
        **/
        void createSAEs (
                in TINASubCommonTypes::t_AccountNumber  subscriberId,
                  // This parameter is optional, it should be used if the
                  // implementation does not consider multiple subscriber
                  // management interfaces (one per subscriber).
                in TINASubCommonTypes::t_EntityList     entityList,
                out TINASubCommonTypes::t_entityIdList  entityIdList
        ) raises (e_applicationError,
                  e_unknownSubscriber,
                  e_invalidEntityInfo);

        /** This operation deletes a set of entities. The entity is
         * removed from all the SAGs it could be assigned to and then
         * deleted.
        **/
        void deleteSAEs (
                in TINASubCommonTypes::t_AccountNumber  subscriberId,
                  // This parameter is optional, it should be used if the
                  // implementation does not consider multiple subscriber
                  // management interfaces (one per subscriber).
                in TINASubCommonTypes::t_entityIdList entityList
        ) raises (e_applicationError,
                  e_unknownSubscriber,
                  e_unknownSAE);

        /** This operation creates a set of SAGs. If the identifier for a
         * SAG is not provided or matches an already existing one, Sub
         * generates a new one and returns it. If this is not the case,
         * it returns the one received.
        **/
        void createSAGs (
                in TINASubCommonTypes::t_AccountNumber  subscriberId,
                  // This parameter is optional, it should be used if the
                  // implementation does not consider multiple subscriber
                  // management interfaces (one per subscriber).
                in TINASubCommonTypes::t_SagList    sagList,
                out TINASubCommonTypes::t_SagIdList sagIdList
        ) raises (e_applicationError,
                  e_unknownSubscriber,
                  e_invalidSAG,
                  e_unknownSAG);

        /** This operation deletes a SAG. The entities belonging to
         *  that SAG are not deleted.
        **/
        void deleteSAGs (
                in TINASubCommonTypes::t_AccountNumber  subscriberId,
                  // This parameter is optional, it should be used if the
                  // implementation does not consider multiple subscriber
                  // management interfaces (one per subscriber).
                in TINASubCommonTypes::t_SagIdList sagIdList
        ) raises (e_applicationError,
                  e_unknownSubscriber,
                  e_unknownSAG);

        /** This operation assigns a list of entities to a SAG.
        **/
        void assignSAEs (
                in TINASubCommonTypes::t_AccountNumber  subscriberId,
                  // This parameter is optional, it should be used if the
                  // implementation does not consider multiple subscriber
                  // management interfaces (one per subscriber).
                in TINASubCommonTypes::t_entityIdList entityList,
```

```
        in TINASubCommonTypes::t_SagId        sagId
) raises (e_unknownSAE,
          e_unknownSAG,
          e_unknownSubscriber,
          e_applicationError);

/** This operation removes a list of entities from a SAG.
**/
void removeSAEs (
        in TINASubCommonTypes::t_AccountNumber  subscriberId,
          // This parameter is optional, it should be used if the
          // implementation does not consider multiple subscriber
          // management interfaces (one per subscriber).
        in TINASubCommonTypes::t_entityIdList entityList,
        in TINASubCommonTypes::t_SagId        sagId
) raises (e_unknownSAE,
          e_unknownSAG,
          e_unknownSubscriber,
          e_applicationError);

/** This operation returns the list of entities assigned to a SAG.
 * If a SAG is not specified, it returns all the entities for that
 * subscriber.
**/
void listSAEs (
        in TINASubCommonTypes::t_AccountNumber  subscriberId,
          // This parameter is optional, it should be used if the
          // implementation does not consider multiple subscriber
          // management interfaces (one per subscriber).
        in TINASubCommonTypes::t_SagId         sagId,
        out TINASubCommonTypes::t_entityIdList entityList
) raises (e_unknownSAG,
          e_unknownSubscriber,
          e_applicationError);

/** This operation returns the list of SAGs for that subscriber.
**/
void listSAGs (
        in TINASubCommonTypes::t_AccountNumber  subscriberId,
          // This parameter is optional, it should be used if the
          // implementation does not consider multiple subscriber
          // management interfaces (one per subscriber).
        out TINASubCommonTypes::t_SagIdList     sagIdList
) raises (e_applicationError,
          e_unknownSubscriber);

/** This operation returns the information about a specific subscriber
**/
void getSubscriberInfo (
        in TINASubCommonTypes::t_AccountNumber  subscriberId,
          // This parameter is optional, it should be used if the
          // implementation does not consider multiple subscriber
          // management interfaces (one per subscriber).
        out TINASubCommonTypes::t_Subscriber     subscriberInfo
) raises (e_applicationError,
          e_unknownSubscriber);

/** This operation modifies the information about a specific subscriber
 * Only name and address fields are modifiable. The rest are updated
 * only by Sub as a result of other operations -createSAGs,...-
**/
void setSubscriberInfo (
        in TINASubCommonTypes::t_AccountNumber  subscriberId,
          // This parameter is optional, it should be used if the
          // implementation does not consider multiple subscriber
          // management interfaces (one per subscriber).
        in TINASubCommonTypes::t_Subscriber     subscriberInfo
) raises (e_unknownSubscriber,
          e_invalidSubscriberInfo,
          e_applicationError);
```

```
        /** This operation returns the list of services subscribed by
         *  a specific subscriber.
        **/
        void  listSubscribedServices (
                in TINASubCommonTypes::t_AccountNumber  subscriberId,
                  // This parameter is optional, it should be used if the
                  // implementation does not consider multiple subscriber
                  // management interfaces (one per subscriber).
                out TINAAccessCommonTypes::t_ServiceList      service_list
        ) raises (e_applicationError,
                  e_unknownSubscriber);

}; // i_SubscriberInfoMgmt


/**
 *  i_SubscriberInfoQuery
 *
 *  Behavior:
 *     This interface allows the UA to retrieve the required
 *     subscription information: list of subscribed services,
 *     list of service profiles.
**/

interface i_SubscriberInfoQuery {

enum t_UserErrorExceptionType {
          UnknownUser,
          userTemporarilyOutOfService
};
exception e_subUserError {
     t_UserErrorExceptionType    exceptionType;
     string      additionalInformation;
};
enum t_ProfileErrorExceptionType {
          ServiceNotContracted,
          InvalidServiceProfile,
          profileTemporarilyOutOfService
};
exception e_subProfileError {
     t_ProfileErrorExceptionType    exceptionType;
     TINAAccessCommonTypes::t_ServiceId serviceId;
     string      additionalInformation;
};


        /** This operation returns the list of services the user is
         *  subscribed to and usable with the current terminal configuration.
        **/
        void listServices (
                in TINACommonTypes::t_UserId  userId,
                out TINASubCommonTypes::t_ServiceIdList      serviceList
        )
        raises (e_subUserError);

        /** This operation returns the profiles corresponding to the
         * specified user for the specified services
        **/
        void getServiceProfiles (
                in TINACommonTypes::t_UserId                  userId,
                in TINASubCommonTypes::t_ServiceIdList           serviceList,
                out TINASubCommonTypes::t_ServiceProfileList      serviceProfileList
       )
        raises (e_subUserError, e_subProfileError);

        /** This operation checks that the specified service profile
         * matches with the subscribed (SAG) service profile and the
         * current terminal configuration
        **/
        void checkServiceProfile (
                in TINACommonTypes::t_UserId                  userId,
```

```
                in TINASubCommonTypes::t_ServiceProfile          serviceProfile,
                out boolean                    accepted
        )
        raises (e_subUserError, e_subProfileError);

}; // i_SubscriberInfoQuery

};

#endif // TINAScsSubscriberInfoAccess_IDL
```

2.15 TINAScsSubscriberMgmt

```
//
// File: TINASubscriberMgmt.idl
// Author: Juan C. Garcia (TINA CoreTeam - Telefonica I+D)
// Last modification: Aug 27, 97
// Contents:
//
// MODIFICATIONS:Sep. 24th 1997 by Hiroshi
//
//
// IDL INTERFACES
// _____________________________ END DESCRIPTION HEADER ____________________________

#ifndef TINAScsSubscriberMgmt_IDL
#define TINAScsSubscriberMgmt_IDL

#include "TINASubCommonTypes.idl"

/**
 *  Module Subscriber Management
 **/
module TINAScsSubscriberMgmt {

/**
 *  Interface: i_SubscriberLCMgmt
 *
 *  Behavior:
 *   It allows to manage the lifecycle of subscribers.
 *   It is used by the subscription coordinator (SCoo).
 **/

interface i_SubscriberLCMgmt {

        // TBD
        void createSubscriber ();

        // TBD
        void deleteSubscriber ();

        // TBD
        void listSubscribers ();

        // TBD
        void listUsers ();

}; // Interface i_SubscriberLCMgmt


/**
 *  Interface: i_ServiceContractInfoUpdate
 *
 *  Behavior:
 *   It allows to update subscriber information about
 *   contracted services.
 *   It is used by the SCM each time a new service contract is signed ,
 *   modified or cancelled by the subscriber.
 **/
```

```
interface i_ServiceContractInfoUpdate {

        // TBD
        void notify ();

}; // Interface i_ServiceContractInfoUpdate

};

#endif // TINAScsSubscriberMgmt_IDL
```

2.16 TINAScsSubscriptionService

```
//
// File: TINAScsSubscriptionService.idl
// Author: Juan C. Garcia (TINA CoreTeam - Telefonica I+D)
// Last modification: Oct 27, 97
// Contents: Definition of the online subscription management
// service specific interface.
//
// MODIFICATIONS:
//   Oct 28th  1997 by Juan C. Garcia made consistent with SCS release (28/10)
//
// IDL INTERFACES
// _____________________________ END DESCRIPTION HEADER ____________________

#ifndef TINAScsSubscriptionService_IDL
#define TINAScsSubscriptionService_IDL

#include "TINASubCommonTypes.idl"

/**
 * Module TINA Subscription Service
**/
module TINAScsSubscriptionService {

/**
 *  i_ProviderOlsSi
 *
 *  Behavior:
 *     This interface is offered through Ret-RP to the subscription
 *     service specific ssUAP.
 *     It provides operations for:
 *     - subscribing to a retailer.
 *     - contracting services with that retailer.
 *     - unsubscribing and withdrawing service contracts.
 *     - creating and deleting entities.
 *     - creating SAGs.
 *     - assigning/removing entities to/from a SAG.
 *     - listing entities and SAGs corresponding to that subscriber.
 *     - querying and modifying the subscriber information.
**/

interface i_ProviderOlsSi {

enum t_errorType { NameTooLong, AddressTooLong, OtherErrors };
exception e_invalidSubscriberInfo{
      t_errorType errorType;
};
exception e_unknownSubscriber{
      TINASubCommonTypes::t_AccountNumber       subscriberId;
};
exception e_applicationError{};
exception e_invalidEntityInfo{};
exception e_unknownSAE{
      TINASubCommonTypes::t_entityId    entityId;
};
exception e_unknownSAG{
```

```
      TINASubCommonTypes::t_SagId       sagId;
};
exception e_invalidSAG{
      TINASubCommonTypes::t_SagId       sagId;
};
exception e_invalidContractInfo{};
exception e_invalidSubscriptionProfile{};
exception e_invalidSAGServiceProfile{
          TINASubCommonTypes::t_ServiceProfileId     spId;
};
exception e_unknownServiceProfile{
          TINASubCommonTypes::t_ServiceProfileId     spId;
};
exception e_unknownServiceId{
      TINAAccessCommonTypes::t_ServiceId serviceId;
};
exception e_invalidSearchCriteria{};
exception e_notSubscribedService{
          TINAAccessCommonTypes::t_ServiceId serviceId;
};

       /**
        *
        * Operations for Subscription and Service Contract handling
        *
        **/

        /** This operation returns the list of services available for
         *  subscription and use.
         **/
        void listServices (
                out TINAAccessCommonTypes::t_ServiceList      serviceList
        ) raises (e_applicationError);

        /** This operation creates a subscription for a new customer.
         * The initial list of services the subscriber wants to contract
         * can be specified.
         * It returns:
         * - a unique identifier for the subscriber.
         * - a list of service templates
         **/
        void subscribe (
                in TINASubCommonTypes::t_Subscriber            subscriberInfo,
                in TINASubCommonTypes::t_ServiceIdList         serviceList,
                out TINASubCommonTypes::t_AccountNumber        subscriberId,
                out TINASubCommonTypes::t_ServiceTemplateList     svcTemplateList
        ) raises (e_invalidSubscriberInfo,
                  e_unknownServiceId,
                  e_applicationError);

        /** This operation creates a (set of) new service contract(s) for
         * an existing customer.
         * A list of services the subscriber wants to contract
         * is specified.
         * It returns a list of service contract management interfaces,
         *  one for each of the services requested.
         **/
        void contractService (
                in TINASubCommonTypes::t_ServiceIdList         serviceList,
                out TINASubCommonTypes::t_ServiceTemplateList     svcTemplateList
        ) raises (e_unknownSubscriber,
                  e_unknownServiceId,
                  e_applicationError);

        /** This operation withdraws a subscription or a list of service
         * contracts.
         * The list of services the subscriber wants to unsubscribe
         * is an input parameter. If this list is empty, that means
         *  the withdrawal of all the services, and thus the subscription.
         **/
        void unsubscribe (
```

```
        in TINASubCommonTypes::t_ServiceIdList         serviceList
 ) raises (e_unknownSubscriber,
           e_unknownServiceId,
           e_applicationError);


/**
 *
 * Operations for Subscriber Information Management.
 *
 **/


 /** This operation creates a set of entities.
  * Sub generates a unique identifier for every entity.
 **/
 void createSAEs (
        in TINASubCommonTypes::t_EntityList     entityList,
        out TINASubCommonTypes::t_entityIdList  entityIdList
 ) raises (e_applicationError,
           e_invalidEntityInfo);

 /** This operation deletes a set of entities. The entity is
  * removed from all the SAGs it could be assigned to and then
  * deleted.
 **/
 void deleteSAEs (
        in TINASubCommonTypes::t_EntityList     entityList,
        in TINASubCommonTypes::t_entityIdList entityList
 ) raises (e_applicationError,
           e_unknownSAE);

 /** This operation creates a set of SAGs.
  * It returns a set of unique SAG identifiers.
 **/
 void createSAGs (
        in TINASubCommonTypes::t_SagList    sagList,
        out TINASubCommonTypes::t_SagIdList sagIdList
 ) raises (e_applicationError,
           e_invalidSAG,
           e_unknownSAG);

 /** This operation deletes a SAG. The entities belonging to
  *  that SAG are not deleted.
 **/
 void deleteSAGs (
        in TINASubCommonTypes::t_SagIdList sagIdList
 ) raises (e_applicationError,
           e_unknownSAG);

 /** This operation assigns a list of entities to a SAG.
 **/
 void assignSAEs (
        in TINASubCommonTypes::t_entityIdList entityList,
        in TINASubCommonTypes::t_SagId        sagId
 ) raises (e_unknownSAE,
           e_unknownSAG,
           e_applicationError);

 /** This operation removes a list of entities from a SAG.
 **/
 void removeSAEs (
        in TINASubCommonTypes::t_entityIdList entityList,
        in TINASubCommonTypes::t_SagId        sagId
 ) raises (e_unknownSAE,
           e_unknownSAG,
           e_applicationError);

 /** This operation returns the list of entities assigned to a SAG.
  * If a SAG is not specified, it returns all the entities for that
  * subscriber.
```

```
        **/
        void listSAEs (
                in TINASubCommonTypes::t_SagId         sagId,
                out TINASubCommonTypes::t_entityIdList entityList
        ) raises (e_unknownSAG,
                  e_applicationError);

        /** This operation returns the list of SAGs for that subscriber.
        **/
        void listSAGs (
                out TINASubCommonTypes::t_SagIdList     sagIdList
        ) raises (e_applicationError);

        /** This operation returns the information about a specific subscriber
        **/
        void getSubscriberInfo (
                out TINASubCommonTypes::t_Subscriber     subscriberInfo
        ) raises (e_applicationError,
                  e_unknownSubscriber);

        /** This operation modifies the information about a specific subscriber
         * Only name and address fields are modifiable. The rest are updated
         * only by Sub as a result of other operations -createSAGs,...-
        **/
        void setSubscriberInfo (
                in TINASubCommonTypes::t_Subscriber     subscriberInfo
        ) raises (e_unknownSubscriber,
                  e_invalidSubscriberInfo,
                  e_applicationError);

        /** This operation returns the list of services subscribed by
         *  a specific subscriber.
        **/
        void  listSubscribedServices (
                out TINAAccessCommonTypes::t_ServiceList      service_list
        ) raises (e_applicationError,
                  e_unknownSubscriber);

       /**
        *
        * Operations for Service Contract Management.
        *
        **/

        /** This operation returns the template for the service.
         **/
        void getServiceTemplate (
                in TINAAccessCommonTypes::t_ServiceId           serviceId,
                out TINASubCommonTypes::t_ServiceTemplate  template
        ) raises (e_applicationError);

        /** This operation creates a service contract.
         * This contract can include a set of service profiles.
         **/
        void defineServiceContract (
                in TINASubCommonTypes::t_ServiceContract        serviceContract,
                out TINASubCommonTypes::t_ServiceProfileIdList  spIdList
        )
        raises (e_applicationError,
                e_invalidContractInfo,
                e_invalidSubscriptionProfile,
                e_invalidSAGServiceProfile);

        /** This operation creates a set of service profiles.
         **/
        void defineServiceProfiles (
                in TINASubCommonTypes::t_SubscriptionProfile        subscriptionProfile,
                in TINASubCommonTypes::t_SagServiceProfileList  sagServiceProfiles,
                out TINASubCommonTypes::t_ServiceProfileIdList  spIdList
        )
        raises (e_applicationError,
```

```
        e_invalidSubscriptionProfile,
        e_invalidSAGServiceProfile);

/** This operation deletes a set of service profiles and their
 * associated SAGs.
 **/
void deleteServiceProfiles (
        in TINASubCommonTypes::t_ServiceProfileIdList  spIdList
)
raises (e_applicationError,
        e_unknownServiceProfile);

/** This operation returns the list of service profiles identifiers
 **/
void listServiceProfiles (
        in TINAAccessCommonTypes::t_ServiceId           serviceId,
        out TINASubCommonTypes::t_ServiceProfileIdList  spIdList
)
raises (e_applicationError);

/** This operation returns the service contract information.
 * If a (list of) SAG(s) is specified it returns the set of
 * SAG service profile for that(those) SAG(s).
 **/
void getServiceContractInfo (
        in  TINAAccessCommonTypes::t_ServiceId           serviceId,
        in  TINASubCommonTypes::t_ServiceProfileIdList                spIdList,
        out TINASubCommonTypes::t_ServiceContract           serviceContract
)
raises (e_applicationError,
        e_unknownServiceProfile);

/** This operation assigns a service profile to a list of SAGs and
 * SAEs.
 **/
void assignServiceProfile (
        in TINASubCommonTypes::t_ServiceProfileId  spId,
        in TINASubCommonTypes::t_SagIdList  sagIdList,
        in TINASubCommonTypes::t_entityIdList  saeIdList
)
raises (e_applicationError,
        e_unknownSAG,
        e_unknownSAE,
        e_unknownServiceProfile);

/** This operation removes a service profile assignment to a list
 * of SAGs and SAEs.
 **/
void removeServiceProfile (
        in TINASubCommonTypes::t_ServiceProfileId  spId,
        in TINASubCommonTypes::t_SagIdList  sagIdList,
        in TINASubCommonTypes::t_entityIdList  saeIdList
)
raises (e_applicationError,
        e_unknownSAG,
        e_unknownSAE,
        e_unknownServiceProfile);

/** This operation activates a set of service profiles
 **/
void activateServiceProfiles (
        in TINASubCommonTypes::t_ServiceProfileIdList  spIdList
)
raises (e_applicationError,
        e_unknownServiceProfile);

/** This operation deactivates a set of service profiles
 **/
void deactivateServiceProfiles (
        in TINASubCommonTypes::t_ServiceProfileIdList  spIdList
)
```

```
        raises (e_applicationError,
                e_unknownServiceProfile);

}; // i_ProviderOlsSi


};

#endif // TINAScsSubscriptionService_IDL
```

2.17 TINAScsSSUAPIntra

```
// FILE: TINAScsSSUAPIntra.idl
//
// VERSION: 1
// DATE 21 August 97
//
// IDL for service session User APplication
// for the TINA- SCS
//
// COMMENTS:
//
// MODIFICATIONS:
//
//
//
#ifndef TINAScsSSUAPIntra_IDL
#define TINAScsSSUAPIntra_IDL

#include "TINAProviderAccess.idl"
#include "TINACommonTypes.idl"
#include "TINAAccessCommonTypes.idl"

/**     Defines the interfaces ADDITIONAL to Ret that are supported by the ssUAP component.
//      within the domain supproting the ssUAP.
**/

module TINAScsSSUAPIntra{

/** Behaviour: Enables the PA to instruct an ssUAP in order that is can respond
// to opportunities to start, join, resume participation and resume session when
// these are initiated by the PA or by the asUAP via the PA.
**/

interface i_AccessInitialise {

        /** Instructs the ssUAP to start a service of a specified type with specified
                properties together with a PA interface(s) that can be used to invoke upon
                the PA to progress the scenario
        **/
        void startServiceInit (
                in TINAAccessCommonTypes::t_ServiceId serviceId,
                in TINACommonTypes::t_SessionProperties ssProperties,
                in TINACommonTypes::t_InterfaceList paintfs
        ) raises (
                TINAProviderAccess::e_ServiceError,
                TINAProviderAccess::e_StartServiceSSPropertyError,
                TINACommonTypes::e_InterfacesError
        );

        /** Instructs the ssUAP to join a session of a unique identity with specified
                properties together with a PA interface(s) that can be used to invoke upon
                the PA to progress the scenario
        **/
        void joinSessionInit (
                in TINAAccessCommonTypes::t_ServiceId serviceId,
                in TINACommonTypes::t_SessionProperties ssProperties,
                in TINACommonTypes::t_InterfaceList paintfs
        ) raises (
```

```
                // these exceptions need changing
                TINAProviderAccess::e_ServiceError,
                TINAProviderAccess::e_StartServiceSSPropertyError,
                TINACommonTypes::e_InterfacesError
        );

        /
** Instructs the ssUAP to resume participation in a session of a unique identity with speci
fied
                properties together with a PA interface(s) that can be used to invoke upon
                the PA to progress the scenario
        **/
        void resumeParticipationInit (
                in TINAAccessCommonTypes::t_ServiceId serviceId,
                in TINACommonTypes::t_SessionProperties ssProperties,
                in TINACommonTypes::t_InterfaceList paintfs
        ) raises (
                // these exceptions need changing
                TINAProviderAccess::e_ServiceError,
                TINAProviderAccess::e_StartServiceSSPropertyError,
                TINACommonTypes::e_InterfacesError
        );

        /** Instructs the ssUAP to resume a session of a unique identity with specified
                properties together with a PA interface(s) that can be used to invoke upon
                the PA to progress the scenario
        **/
        void ResumeServiceInit (
                in TINAAccessCommonTypes::t_ServiceId serviceId,
                in TINACommonTypes::t_SessionProperties ssProperties,
                in TINACommonTypes::t_InterfaceList paintfs
        ) raises (
                // these exceptions need changing
                TINAProviderAccess::e_ServiceError,
                TINAProviderAccess::e_StartServiceSSPropertyError,
                TINACommonTypes::e_InterfacesError
        );


};
//i_AccessInitialise

};
//module TINAScsSSUAPIntra


#endif //TINAScsSSUAPIntra_IDL
```

2.18 TINAScsSF

```
//
//   FILE: TINAScsSF.idl
//
//   VERSION: 1.3
//   DATE: 08/20/1997
//   DESCRIPTION:
//      IdL definition of SF (Service Factory)
//      for TINA-C SCS.
//
//   COMMENTS:compilable version, with comments in C style; new naming schema
//
//   MODIFICATIONS:Sep. 20th 1997
//
//
//   IDL INTERFACES
// _____________________________ END DESCRIPTION HEADER _____________________________

#ifndef TINAScsSF_ITF_IDL
#define TINAScsSF_ITF_IDL
```

```
#include "TINACommonTypes.idl"
#include "TINAAccessCommonTypes.idl"
#include "TINAProviderAccess.idl"

/** Module TINA Service Factory
**/
module TINAScsSF
{
/**
* this is a Session id, unique for a SF
**/
typedef unsigned long t_GlobalSessionId;
//
/**
*this is a USM ID unique within a session
**/
typedef unsigned long t_UserSessionId;

//  interface i_SSCreate
/**
 *  Behavior:
 *      This interface allows the UA to request for the creation of a new
 *      Service session and it allows the SSM to request for the creation
 *      of a new USM for a user who is joining the session.
**/
        interface i_SSCreate
        {

        /** This operation creates a new SSM and USM for a specified service.
        * It returns information on the newly created session (supported
        * session model and FSs) and SSM/USM IDs to be used to manage the
        * session
        **/
        void createSSession (
                in TINAAccessCommonTypes::t_ServiceId serviceId,
                in TINACommonTypes::t_UserId userId,
                in TINAProviderAccess::t_ApplicationInfo app,
                in TINACommonTypes::t_SessionModelReq sessionModelReq,
                in TINACommonTypes::t_InterfaceList uaRef,
                in TINAProviderAccess::t_StartServiceSSProperties ssProperties,
                out TINAAccessCommonTypes::t_SessionInfo sessionInfo,
                out t_GlobalSessionId sessionId
//              out t_UserSessionId usersessionId,/ not needed?
                ) raises (
                TINAProviderAccess::e_ServiceError,
                TINAProviderAccess::e_ApplicationInfoError,
                TINACommonTypes::e_SessionModelError,
                TINACommonTypes::e_PropertyError
        );

        /** This operation creates a new USM for a specified service session
        * that is managed by the SF.
        * It returns information on the newly created USM (supported
        * session model and FSs).
        **/
        void createUserSSession (
                in t_GlobalSessionId sessionId,
                in TINACommonTypes::t_PartyId partyId,
                in TINACommonTypes::t_InterfaceList uassmRefs,
                in TINAProviderAccess::t_ApplicationInfo app,
                out TINAAccessCommonTypes::t_SessionInfo sessionInfo
//              out t_UserSessionId usersessionId
        ) raises (
                TINAProviderAccess::e_SessionError,
                TINAProviderAccess::e_ApplicationInfoError
        );

// the following operations will be defined in the next version of the document
//              void createPeerSSession (
//              );
```

```
//              void createCompSSession (
//              );

        };

//  interface i_SSManage
/**
*  Behavior:
*      This interface allows the SSM (or the UA) to manage a specified
*      Service session and to request information on a specified Service
*       Session.
**/

        interface i_SSManage
        {


        /**  This operation allows to end a Service Session. SSM and USMs are
        *  deleted and related resources  are released.
        **/
        void endSSession (
                in t_GlobalSessionId sessionId
        ) raises (
                TINAProviderAccess::e_SessionError
        );

        /**
        *  This operation allows to end a User Service Session. The USM
        * (identified by the usersessionId) is deleted and resources are
        * released.
        **/
        void endUserSSession (
                in t_GlobalSessionId sessionId,
                in TINACommonTypes::t_PartyId partyId
//PFH           in t_UserSessionId usersessionId
        ) raises (
                TINAProviderAccess::e_SessionError
        );


// the following operations will be defined in the next version of the document
//              void endPeerSSession (
//              );
//
//              void endCompSSession (
//              );

        /**
        *  This operation allows the UA to request the list of Service session
        *  managed by the SF and which match desired properties
        **/
        void listSSessions (
                in  TINAProviderAccess::t_SessionSearchProperties desiredProperties,
                out TINAAccessCommonTypes::t_SessionList sessions
        ) raises (
                TINACommonTypes::e_PropertyError,
                TINACommonTypes::e_ListError
        );


        /**
        *  This operation returns the SSM interface references for a specified
        *  session
        **/
        void  getSsmRef (
                in t_GlobalSessionId sessionId,
                out TINACommonTypes::t_InterfaceList  ssmSession
                ) raises (
                TINAProviderAccess::e_SessionError
        );
```

```
        /**
        *  This operation allows the SSM to suspend a service session. It
        *  returns the Interface reference of a resume interface to be used by
        *  UA to resume the service session. Interfaces on USMs/SSM cannot
        *  be accessed until resumption.
        **/
        void suspendSSession (
                in t_GlobalSessionId sessionId,
                out  TINACommonTypes::t_InterfaceList  resumeItfRef
        ) raises (
                TINAProviderAccess::e_SessionError
        );

        /**
        *  This operation allows the SSM to request to suspend the
        *  participation of a specified user. It returns the interface
        *  reference of the resume interface to be used by the UA to resume
        *  the participation. Interfaces on USM are disabled until
        *  participation resumption
        **/
        void suspendParticipation (
                in t_GlobalSessionId sessionId,
                in TINACommonTypes::t_PartyId partyId,
// PFH          in t_UserSessionId usersessionId,
                out  TINACommonTypes::t_InterfaceList  resumeItfRef
        ) raises (
                TINAProviderAccess::e_SessionError
        );
};


//  interface i_Resume
/**
*  Behavior:
*      This interface allows the UA to resume a suspended
*      Service session or to resume the participation to an active service
*      session.
*
**/
        interface i_Resume
        {

        /**
        *  This operation allows a UA to resume a suspended service session.
        *  Interfaces on SSM and USM of the requesting user are enabled.
        *  UAs of suspended users are informed that the session has been
        *  resumed.
        **/
        void resumeSSession (
                in t_GlobalSessionId sessionId,
                in t_UserSessionId usersessionId,
                in TINAProviderAccess::t_ApplicationInfo app,
                out TINAAccessCommonTypes::t_SessionInfo sessionInfo
        ) raises (
                TINAProviderAccess::e_SessionError,
                TINAProviderAccess::e_ApplicationInfoError
        );


        /**
        *  This operation allows a UA to resume the participation in an active
        *  service session.
        *  Interfaces on USM of the requesting user (identified by the
        *  userSessionId) are enabled.
        **/
        void resumeParticipation (
                in t_GlobalSessionId sessionId,
                in TINACommonTypes::t_PartyId partyId,
// PFH          in t_UserSessionId usersessionId,
```

```
                in TINAProviderAccess::t_ApplicationInfo app,
                out TINAAccessCommonTypes::t_SessionInfo sessionInfo
        ) raises (
                TINAProviderAccess::e_SessionError,
                TINAProviderAccess::e_ApplicationInfoError
        );

        };


// interface  i_Init
/**
* Behavior:
* This interface allows the SLCM to initialize, configure and manage the SF.
**/

        interface i_Init

        {
//      operations' parameters will be defined at a later stage:

        /**  - allows the SLCM to configure the SF. **/
        void configure();

        /**  - allows the SLCM to set Service Session **/
        void setSessionInfo();

        /**  - allows the SLCM to get Service Session
         instances information **/
        void getSessionInfo();

        /**  - sets the state of the SF to deactivating. Future
         service session creation requests will be rejected. Once ongoing
         service sessions are finished, the SF passes to inactive state. **/
        void deactivate();

        /**  - sets the state of the SF to active. In this state,
         the SF can be used for service provision. **/
        void activate();

        /**  - sets the SF state to inactive without a deactivation
         phase. The SF reports the users using it about the fact and the
         service sessions are ended inmediately. **/
        void halt();

        /**  - kills the SF if it is in inactive state. **/
        void delete();

        };


// interface    i_SSEvents
/**
* Behavior:
* This interface allows the SSM,USM, PeerUSM and CompUSM  to notify SF about
* changes that occour in the Service Session.
**/

        interface i_SSEvents
        {
// is this operation/interface needed??
//              void  notifySSModification (
//                      in  t_GlobalSessionId sessionId,
//                      in  TINAAccessCommonTypes::t_SessionInfo sessionInfo
//              )raises (
//              TINAProviderAccess::e_SessionError
//              );
```

```
        };


};

#endif /*SF_ITF_IDL */
```

2.19 TINAScsSSMInit

```
// FILE: TINAScsSSMInit.idl
//
// IDL for Service Session Manager
//
// Author: Per Fly Hansen (Tele Danmark)
// Creation date: August 21st, 1997
// Last modification date: September 24th, 1997 /PFH
//

#ifndef TINAScsSSMInit_IDL
#define TINAScsSSMInit_IDL

#include "TINACommonTypes.idl"

module TINAScsSSMInit {

   /** this ID identified uniquely for SSM and USM, managed by SF. **/
  typedef unsigned long t_GlobalSessionId; // this is a Session id, unique for an SF

   /** use the errorCodes as for e_PropertyError **/
  exception e_initSSPropertyError {
      TINACommonTypes::t_PropertyErrorStruct propertyError;
  };

  enum t_InitialiseSSMErrorCode {
      CannotCreateInterfaces,
      UnknownError
  };

  exception e_initSSMError {
        t_InitialiseSSMErrorCode errorCode;
  };

  enum t_sessionSSMErrorCode {
      UnknownSession,
      NotAllowed,
      UnknownError
  };

  /** used in halt and suspend operations **/
  exception e_sessionSSMError {
        t_sessionSSMErrorCode errorCode;
  };

  interface i_Init {
        /** Used by SF, initialises a new Service Session, **
         ** returns common part of t_sessionInfo           **/
        void initialise (
          in TINACommonTypes::t_PropertyList initProperties,
          in t_GlobalSessionId sessionId,
          in TINACommonTypes::t_SessionModelReq sessionModelReq,
             /* Interfaces on UA and SF: */
          in TINACommonTypes::t_InterfaceList infs,
          out TINACommonTypes::t_PartyId partyId,
          out TINACommonTypes::t_SessionModelList sessionModels,
          out TINACommonTypes::t_InterfaceList initialSSMInfs
        ) raises (
           e_initSSPropertyError,
           e_initSSMError
        );
```

```
        /** Used by SF, halts a Service Session if allowed **/
        void halt (
          in t_GlobalSessionId sessionId
        ) raises (
          e_sessionSSMError
        );

        /** Used by SF, suspends a Service Session if allowed **/
        void suspend (
          in t_GlobalSessionId sessionId
        ) raises (
          e_sessionSSMError
        );
  }; // i_Init
};
#endif /* TINAScsSSMInit_IDL */
```

2.20 TINAScsSSMIntra

```
// FILE: TINAScsSSMIntra.idl
//
// IDL for Service Session Manager
//
// Author: Per Fly Hansen (Tele Danmark)
// Creation date: August 21st, 1997
// Modification date: September 3rd, 1997
//

#ifndef TINAScsSSMIntra_IDL
#define TINAScsSSMIntra_IDL

#include "TINAProviderAccess.idl"
#include "TINAScsAmcObject.idl"

module TINAScsSSMIntra {

  typedef unsigned long t_GlobalSessionID; // this is a Session id, unique for an SF

  interface i_Join {

        void joinSessionWithInvitation (
                in TINACommonTypes::t_UserName name,
                in TINAAccessCommonTypes::t_InvitationId invitationId,
                in TINAProviderAccess::t_ApplicationInfo app,
                out TINAAccessCommonTypes::t_SessionInfo sessionInfo
        ) raises (
                TINAAccessCommonTypes::e_AccessError,
                TINAAccessCommonTypes::e_InvitationError,
                TINAProviderAccess::e_ApplicationInfoError
        );

        void joinSessionWithAnnouncement (
                in TINACommonTypes::t_UserName name,
                in TINAAccessCommonTypes::t_AnnouncementId announcementId,
                in TINAProviderAccess::t_ApplicationInfo app,
                out TINAAccessCommonTypes::t_SessionInfo sessionInfo
        ) raises (
                TINAAccessCommonTypes::e_AccessError,
                TINAProviderAccess::e_AnnouncementError,
                TINAProviderAccess::e_ApplicationInfoError
        );

        void replyToInvitation (
                in TINACommonTypes::t_UserName name,
                in TINAAccessCommonTypes::t_InvitationId invitationId,
                in TINACommonTypes::t_InvitationReply reply
        ) raises (
                TINAAccessCommonTypes::e_AccessError,
```

```
                TINAAccessCommonTypes::e_InvitationError,
                TINACommonTypes::e_InvitationReplyError
        );
  }; // i_Join

  interface i_Resume {
        //  This operation allows an SF to resume a suspended service session.
        void resumeSession (
                in TINACommonTypes::t_PartyId partyId,
                in TINACommonTypes::t_InterfaceStruct usmItfs, // i_usmResume
                in TINAProviderAccess::t_ApplicationInfo app,
                out TINAAccessCommonTypes::t_SessionInfo sessionInfo
        ) raises (
                TINAProviderAccess::e_SessionError,
                TINAProviderAccess::e_ApplicationInfoError
        );

        //  This operation allows an SF to resume the participation of a
        //  given user in an active service session.
        void resumeParticipation (
                in TINACommonTypes::t_PartyId partyId,
                in TINACommonTypes::t_InterfaceStruct usmItfs,
                in TINAProviderAccess::t_ApplicationInfo app,
                out TINAAccessCommonTypes::t_SessionInfo sessionInfo
        ) raises (
                TINAProviderAccess::e_SessionError,
                TINAProviderAccess::e_ApplicationInfoError
        );
  }; // i_Resume

/** Receives accounting events pushed from CSM.
**/

interface i_AccountingPush : TINAScsAmcObject::i_AccountingPush {
    //no additional operations defined
  }; //i_AccountingPush


/** Controls the delivery of SSM generated accounting events push to
*accouting interface on the USM. Client UA/SF?
**/

interface i_AccountingPushMgmt: TINAScsAmcObject::i_AccObjectManagement {
    // no additional operations defined
  }; //i_AccountingPushMgmt

};

#endif /* TINAScsSSMIntra_IDL */
```

2.21 TINAScsSSMProviderBasicUsage

```
// File TINAScsSSMProviderBasicUsage.idl
//
// Author: Per Fly Hansen (Tele Danmark)
//
// Creation Date: September 23rd 1997
//
// Modifications:
//

#ifndef TINAScsSSMProviderBasicUsage_IDL
#define TINAScsSSMProviderBasicUsage_IDL

#include "TINACommonTypes.idl"
#include "TINAUsageCommonTypes.idl"
#include "TINASessionModel.idl"

module TINAScsSSMProviderBasicUsage {
```

```
interface i_ProviderGetInterfaces {

        void getInterfaceTypes (
                in TINACommonTypes::t_PartyId reqId,
                out TINACommonTypes::t_InterfaceTypeList itfTypeList
        ) raises (
                TINAUsageCommonTypes::e_UsageError
        );

        void getInterface (
                in TINACommonTypes::t_PartyId reqId,
                in TINACommonTypes::t_InterfaceTypeName type,
                in TINACommonTypes::t_MatchProperties desiredProperties,
                out TINACommonTypes::t_InterfaceStruct itf
        ) raises (
                TINAUsageCommonTypes::e_UsageError,
                TINACommonTypes::e_InterfacesError,
                TINACommonTypes::e_PropertyError
        );

        void getInterfaces (
                in TINACommonTypes::t_PartyId reqId,
                in TINACommonTypes::t_MatchProperties desiredProperties,
                out TINACommonTypes::t_InterfaceList itfList
        ) raises (
                TINAUsageCommonTypes::e_UsageError,
                TINACommonTypes::e_PropertyError
        );

};      // interface i_ProviderGetInterfaces


interface i_ProviderRegisterInterfaces {

        void registerInterface (
                in TINACommonTypes::t_PartyId reqId,
                in TINACommonTypes::t_InterfaceStruct itf,
                out TINACommonTypes::t_InterfaceIndex itfIndex
        ) raises (
                TINAUsageCommonTypes::e_UsageError,
                TINACommonTypes::e_InterfacesError
        );

        void registerInterfaces (
                in TINACommonTypes::t_PartyId reqId,
                inout TINACommonTypes::t_RegisterInterfaceList itfs
        ) raises (
                TINAUsageCommonTypes::e_UsageError,
                TINACommonTypes::e_InterfacesError,
                TINACommonTypes::e_RegisterError
        );

        void registerInterfaceTypes (
                in TINACommonTypes::t_PartyId reqId,
                in TINACommonTypes::t_InterfaceTypeList types
        ) raises (
                TINAUsageCommonTypes::e_UsageError,
                TINACommonTypes::e_InterfacesError
        );

        void listRegisteredInterfaces (
                in TINACommonTypes::t_PartyId reqId,
                out TINACommonTypes::t_RegisterInterfaceList registeredItfs
        ) raises (
                TINAUsageCommonTypes::e_UsageError
        );

        void unregisterInterface (
                in TINACommonTypes::t_PartyId reqId,
                in TINACommonTypes::t_InterfaceIndex index
```

```
        ) raises (
                TINAUsageCommonTypes::e_UsageError,
                TINACommonTypes::e_InterfacesError,
                TINACommonTypes::e_UnregisterError
        );

        void unregisterInterfaces (
                in TINACommonTypes::t_PartyId reqId,
                in TINACommonTypes::t_InterfaceIndexList indexes
        ) raises (
                TINAUsageCommonTypes::e_UsageError,
                TINACommonTypes::e_InterfacesError,
                TINACommonTypes::e_UnregisterError
        );

};      // interface i_ProviderRegisterInterfaces

interface i_ProviderInterfaces
                        : i_ProviderGetInterfaces,
                        i_ProviderRegisterInterfaces
{
};      // interface i_ProviderInterfaces


interface i_ProviderBasicReq
                : i_ProviderInterfaces,
                TINASessionModel::i_SessionModel
{
        void endSessionReq (
                in TINACommonTypes::t_PartyId   reqId,
                out any                         accountInfo
        ) raises (
                TINAUsageCommonTypes::e_UsageError
        );

        void suspendSessionReq(
                in TINACommonTypes::t_PartyId reqId,
                out TINACommonTypes::t_SessionId sessionId,
                out TINACommonTypes::t_InterfaceList resumeIR,
                out any                         accountInfo
        ) raises (
                TINAUsageCommonTypes::e_UsageError
        );

};      // interface i_ProviderBasicReq

};      // module TINAScsSSMProviderBasicUsage

#endif //TINAScsSSMProviderBasicUsage_IDL
```

2.22 TINAScsSSMProviderControlSRUsage

```
// File TINAScsSSMProviderControlSRUsage.idl
//
// Author: Per Fly Hansen (Tele Danmark)
//
// Creation Date: September 23rd 1997
//
// Modifications:
//

#ifndef TINAScsSSMProviderControlSRUsage_IDL
#define TINAScsSSMProviderControlSRUsage_IDL

#include "TINACommonTypes.idl"
#include "TINAUsageCommonTypes.idl"
#include "TINAControlSRTypes.idl"

module TINAScsSSMProviderControlSRUsage {
```

```
  interface i_ProviderControlSRReq {

    void setControlReq(
                in TINACommonTypes::t_PartyId reqId,
                in TINACommonTypes::t_PartyId controllerPartyId,
                in TINACommonTypes::t_ElementId controlledId,
                in TINAControlSRTypes::t_ControlDescription control
        ) raises (
                TINAUsageCommonTypes::e_UsageError,
                TINAUsageCommonTypes::e_PartyError
        );

  };    // interface i_ControlSRReq

};      // module TINAScsSSMProviderControlSRUsage

#endif //TINAScsSSMProviderControlSRUsage_IDL
```

2.23 TINAScsSSMProviderMultipartyUsage

```
// File TINAScsSSMProviderMultipartyUsage.idl
//
// Author: Per Fly Hansen (Tele Danmark)
//
// Creation Date: September 23rd 1997
//
// Modifications:
//

#ifndef TINAScsSSMProviderMultiPartyUsage_IDL
#define TINAScsSSMProviderMultiPartyUsage_IDL

#include "TINACommonTypes.idl"
#include "TINAUsageCommonTypes.idl"
#include "TINAScsCommonTypes.idl"

module TINAScsSSMProviderMultipartyUsage {

interface i_ProviderMultipartyReq {

        void listParties (
                in TINACommonTypes::t_PartyId reqId,
                out TINACommonTypes::t_PartyIdList partyIdList
        ) raises (
                TINAScsCommonTypes::e_ProviderSessionError,
                TINAScsCommonTypes::e_PartyIdError
        );

        void listPartiesWithDetails (
                in TINACommonTypes::t_PartyId reqId,
                out TINAUsageCommonTypes::t_PartyDetailsList partyDetailsList
        ) raises (
                TINAScsCommonTypes::e_ProviderSessionError
        );

        void getPartyDetails (
                in TINACommonTypes::t_PartyId reqId,
                in TINACommonTypes::t_PartyId partyId,
                out TINAUsageCommonTypes::t_PartyDetailsList partyDetailsList
        ) raises (
                TINAScsCommonTypes::e_ProviderSessionError,
                TINAScsCommonTypes::e_PartyIdError
        );

        void getMyPartyDetails (
                in TINACommonTypes::t_PartyId reqId,
                out TINAUsageCommonTypes::t_PartyDetails myDetails
        ) raises (
```

```
        TINAScsCommonTypes::e_ProviderSessionError,
        TINAScsCommonTypes::e_PartyIdError
);

void modifyPartyTypeReq (
        in TINACommonTypes::t_PartyId reqId,
        in TINACommonTypes::t_PartyId partyId,
        in TINAUsageCommonTypes::t_PartyType newType
) raises (
        TINAScsCommonTypes::e_ProviderSessionError,
        TINAScsCommonTypes::e_PartyIdError,
        TINAScsCommonTypes::e_PartyTypeError
);

void endMyParticipationReq (
        in TINACommonTypes::t_PartyId reqId,
        out any AccountingInfo
) raises (
        TINAScsCommonTypes::e_ProviderSessionError,
        TINAScsCommonTypes::e_PartyIdError
);

/** Cannot be used by a requester to end the requestor's participation
**/
void endPartyReq (
        in TINACommonTypes::t_PartyId reqId,
        in TINACommonTypes::t_PartyId endPartyId
) raises (
        TINAScsCommonTypes::e_ProviderSessionError,
        TINAScsCommonTypes::e_PartyIdError
);

void suspendMyParticipationReq (
        in TINACommonTypes::t_PartyId reqId,
        out TINACommonTypes::t_InterfaceList resumeIR,
        out any AccountingInfo

) raises (
        TINAScsCommonTypes::e_PartyIdError,
        TINAScsCommonTypes::e_ProviderSessionError
);

/** suspendPartyReq() cannot be used to suspend my particpation
because it cannot return a resume interface reference
**/

void suspendPartyReq (
        in TINACommonTypes::t_PartyId reqId,
        in TINACommonTypes::t_PartyId suspendPartyId
) raises (
        TINAScsCommonTypes::e_ProviderSessionError,
        TINAScsCommonTypes::e_PartyIdError
);

void inviteUserReq (
        in TINACommonTypes::t_PartyId reqId,
        in TINACommonTypes::t_UserDetails invitedUser,
        out TINAUsageCommonTypes::t_InvitationId invitationId,
        out TINACommonTypes::t_InvitationReply reply
) raises (
        TINAScsCommonTypes::e_ProviderSessionError,
        TINACommonTypes::e_UserDetailsError
);

void announceSessionReq (
        in TINACommonTypes::t_PartyId reqId,
        in TINACommonTypes::t_AnnouncementProperties announcement
) raises (
        TINAScsCommonTypes::e_ProviderSessionError,
        TINAUsageCommonTypes::e_AnnouncementError
);
```

```
};      // interface i_ProviderMultipartyReq

};      // module TINAScsSSMProviderMultipartyUsage

#endif // TINAScsSSMProviderMultiPartyUsage_IDL
```

2.24 TINAScsSSMProviderPaSBUsage

```
// File: TINAScsSSMProviderPaSBUsage.idl
//
// So far just a copy from RP0.7 /Per
//
// Author: Per Fly Hansen (Tele Danmark)
//
// Creation Date: September 23rd 1997
//
// Modifications:
//

#ifndef TINAScsSSMProviderPaSBUsage_IDL
#define TINAScsSSMProviderPaSBUsage_IDL

// DESCRIPTION:
//  TINA Participant Oriented Stream Binding Feature Set
//  Provider Module
//
// Interfaces and types needed to support the participant
// oriented stream binding feature set for providers.

#include "TINAPaSBTypes.idl"

module TINAScsSSMProviderPaSBUsage {

enum t_PaSBSetupErrors
   {
        PaSBSetup_InvalidSBId,                  // Unknown SB
        PaSBSetup_InvalidSBOp,                  // Invalid op for this SB
        PaSBSetup_UnknownSBType,                // Unknown stream binding type
        PaSBSetup_UnknownMediaType,             // Unknown media type
        PaSBSetup_IncompatibleParameters,       // Incompatible params:
                // E.g. media type and sb type, media params with media type
        PaSBSetup_InvalidParticipantId,         // Unknown participant
        PaSBSetup_UnknownParticipantType,       // Unknown type of participant
        PaSBSetup_SuspendedParticipant,         // Suspended participant
        PaSBSetup_UnknownCritieria,             // Unknown criterion
        PaSBSetup_InvalidCriteria,              // Not valid for this SB
        PaSBSetup_UnsupportedCritieria,         // Not supported by session
        PaSBSetup_CriteriaNotMet,               // Success criteria not met
        PaSBSetup_CommsNotAvailable,            // Supporting communications
                                                // not available
        PaSBSetup_InsuffisicentBandwidth,       // Not enough bandwidth
        PaSBSetup_QoSCannotBeMet,               // QoS requirements not met
        PaSBSetup_InsufficientResources,        // No resources for connection
        PaSBSetup_NoPathFound,                  // Could not connect points
        PaSBSetup_UnknownSFEP,                  // Given SFEP not known
        PaSBSetup_UnknownRFEP                   // No supporting RFEP
   };

// Exception on sb creation, addion of participants, type modifications
exception e_PaSBSetupError
   {
        t_PaSBSetupErrors errorCode;            // Error
        TINACommonTypes::t_ElementId     problemEl;     // Element causing problem
                // Valid for appropriate error codes, e.g. Invalid particpant
                // Invalid SB (returns given SBId), invalid participant type
   };
```

```
enum t_PaSBOperationErrors
   {
        PaSBOper_InvalidSBId,                   // Unknown SB
        PaSBOper_InvalidSBOp,                   // Invalid op for this SB
        PaSBOper_InvalidParticipantId,          // Unknown participant
        PaSBOper_SuspendedParticipant,          // Suspended participant
        PaSBOper_CriteriaNotMet,                // Success criteria not met
        PaSBOper_CommsNotAvailable,             // Supporting communications
        PaSBOper_InsufficientResources,         // No resources for activation
        PaSBOper_CommunicationFailure           // Comms unable to meet request
   };


// Exception on state change ops (delete, activate, deactivate)
exception e_PaSBOperationError
   {
        t_PaSBOperationErrors errorCode;        // Error
        TINACommonTypes::t_ElementId     problemEl;     // Element causing problem
                // Valid for appropriate error codes, e.g. Invalid particpant
                // Invalid SB (returns given SBId)
   };


enum t_PaSBQueryErrors
   {
        PaSBQuery_InvalidSBId,                  // Unknown SB
        PaSBQuery_InvalidSBOp,                  // Invalid op for this SB
        PaSBQuery_InvalidElementId,             // Unknown element:
                                                // sfep or pariticpant
        PaSBQuery_UnknownMediaType,             // Unknown media type
        PaSBQuery_IncompatibleParameters,       // Incompatible params:
        PaSBQuery_SuspendedParticipant          // Suspended participant
   };


// Exception on query type operations
exception e_PaSBQueryError
   {
        t_PaSBQueryErrors errorCode;    // Error
        TINACommonTypes::t_ElementId     problemEl;     // Element causing problem
                // Valid for appropriate error codes, e.g. Invalid particpant
                // Invalid SB (returns given SBId)
   };

// Exception on asynchronous handling of operation
exception e_NoSynchronousReqResp
   {
        TINAStreamCommonTypes::t_RequestId reqId; // Request Identifier for later response
   };


// DESCRIPTION:
//   i_ProviderPaSBReq interface
//   The participant oriented stream binding request interface
//   Allows session parties to create a stream binding, modify
//   its configuration, remove the stream binding, and make general
//   queries on stream bindings.
//
//   This interface supports a participant oriented stream binding
//   model.  As a result, operations are generally in terms of:
//   stream binding type
//   media types
//   participants (session members who have a relation with the SB)
//   stream binding identifiers (subsequent operations from creation)
// AUTHOR:
//   Stephanie Hogg
// CREATION DATE:
//  20/08/97
// UPDATES:
//
```

```
// #include "PaSBCommonTypes.idl"
// #include "PaSBProviderErrorTypes.idl"

interface i_ProviderPaSBReq
{
        // This interface provides control for all stream bindings
        // to which the requester has access.

        // Establish a stream binding, in terms of type, media, & participants
        void addProviderPaSBReq(
                in TINACommonTypes::t_PartyId reqId, // Requesters id
                in TINAStreamCommonTypes::t_SBType reqType,//
 type identifier: overall type
                in TINASBCommSCommonTypes::t_MediaDescList media,// Media of assoc flow
                        // additional overall type info,&/or implicit flow types
                        // Eg. SB type = multimedia, flow types = video, audio
                        // Additional flow type parameters, such as:
                        // QoS: service level qos related to media types
                        // Eg. for audio: CD, FM stereo, AM, phone, mobile
                in TINAPaSBTypes::t_ParticipantDescList reqMembers,
                        // List of participants, each participant described by:
                        // id, type, control SR, and role (source/sink/etc.),
                        // media parameters modifying overal media requirements
                        // success & recovery criteria
                        // initial administrative state state
                in TINAStreamCommonTypes::t_SFEPServDescList requesterSIs,
                        // Optional: if 0 SFEPs, requester is not participating
                in TINAStreamCommonTypes::t_SBSuccessCriteria criteria,
                        // What is necessary for this operation to succeed?
                in TINAStreamCommonTypes::t_SBRecover recActions,
                        // Actions on failure to recover stream binding
                        // Success of recovery criteria
                in boolean wait, // wait for reply flag: true = wait
                out TINAStreamCommonTypes::t_SBBindState status)        //
 Details of stream binding
                raises ( TINAUsageCommonTypes::e_UsageError,
                          e_PaSBSetupError, e_NoSynchronousReqResp );

        // Remove a stream binding
        void deleteProviderPaSBReq(
                in TINACommonTypes::t_PartyId reqId, // Requesters id
                in TINAStreamCommonTypes::t_SBId sbId, // stream binding id
                in boolean wait, // wait for reply flag: true = wait
                out TINAStreamCommonTypes::t_SBBindState status)//
 Details of stream binding
                raises ( TINAUsageCommonTypes::e_UsageError,
                          e_PaSBOperationError, e_NoSynchronousReqResp );

        // Add participants to a stream binding
        void addParticipantsProviderPaSBReq(
                in TINACommonTypes::t_PartyId reqId, // Requesters id
                in TINAStreamCommonTypes::t_SBId sbId, // stream binding id
                in TINAPaSBTypes::t_ParticipantDescList reqMembers, //
 List of participants
                in TINAStreamCommonTypes::t_SFEPServDescList requesterSIs, //
 Optional requester SFEPs
                in boolean wait, // wait for reply flag: true = wait
                out TINAStreamCommonTypes::t_SBBindState status) //
 Details of stream binding
                raises ( TINAUsageCommonTypes::e_UsageError,
                          e_PaSBSetupError, e_NoSynchronousReqResp );

        // Remove participants from a stream binding
        void deleteParticipantsProviderPaSBReq(
                in TINACommonTypes::t_PartyId reqId, // Requesters id
                in TINAStreamCommonTypes::t_SBId sbId, // stream binding id
                in boolean all, // All participants or listed participants?
                in TINAPaSBTypes::t_ParticipantIdList reqMembers,
                        // list of participants to be removed, valid all=false
                in boolean wait, // wait for reply flag: true = wait
                out TINAStreamCommonTypes::t_SBBindState status) //
```

```
Details of stream binding
                raises ( TINAUsageCommonTypes::e_UsageError,
                         e_PaSBOperationError, e_NoSynchronousReqResp );

        // Activate participants in a stream binding/ activate implicit
        // flows specified by type
        void activateParticipantsProviderPaSBReq(
                in TINACommonTypes::t_PartyId reqId, // Requesters id
                in TINAStreamCommonTypes::t_SBId sbId, // stream binding id
                in boolean all, // All participants or listed participants?
                in TINAPaSBTypes::t_ParticipantIdList reqMembers,
                        // list of members to be activated, valid if all=false
                in boolean allFlows,    // All sub types or listed sub types?
                in TINASBCommSCommonTypes::t_MediaDescList reqFlows,
                        // Media types to be activated, valid allFlows = false
                in boolean wait, // wait for reply flag: true = wait
                out TINAStreamCommonTypes::t_SBBindState status) //
Details of stream binding
                raises ( TINAUsageCommonTypes::e_UsageError,
                         e_PaSBOperationError, e_NoSynchronousReqResp );

        // Deactivate participants in a stream binding/ deactivate implicit
        // flows specified by type
        void deactiviateParticipantsProviderPaSBReq(
                in TINACommonTypes::t_PartyId reqId, // Requesters id
                in TINAStreamCommonTypes::t_SBId sbId, // stream binding id
                in boolean all, // All participants or listed participants?
                in TINAPaSBTypes::t_ParticipantIdList reqMembers,
                        // list of members to be deactivated, valid all=false
                in boolean allFlows,    // All sub types or listed sub types?
                in TINASBCommSCommonTypes::t_MediaDescList reqFlows,
                        // Media types to be deactivated, valid allFlows=false
                in boolean wait, // wait for reply flag: true = wait
                out TINAStreamCommonTypes::t_SBBindState status) //
Details of stream binding
                raises ( TINAUsageCommonTypes::e_UsageError,
                         e_PaSBOperationError, e_NoSynchronousReqResp );

        // Modify the stream binding by adding, modifying, or removing
        // media for the overall stream binding or specified participants
        void modifyParticipantsProviderPaSBReq(
                in TINACommonTypes::t_PartyId reqId, // Requesters id
                in TINAStreamCommonTypes::t_SBId sbId, // stream binding id
                in boolean all,// All participants or listed participants?
                in TINAPaSBTypes::t_ParticipantIdList reqMembers,//
                        // list of participants modify, valid if all=false
                in TINASBCommSCommonTypes::t_MediaDescList newTypes,// Media types to add
                in TINASBCommSCommonTypes::t_MediaDescList oldTypes, //
Media types to be removed
                in TINASBCommSCommonTypes::t_MediaChangeDescList modTypes,//
Media types to modify
                in TINAStreamCommonTypes::t_SFEPServDescList requesterSIs, //
Optional requester SFEPs
                in boolean wait, // wait for reply flag: true = wait
                out TINAStreamCommonTypes::t_SBBindState status) //
Details of stream binding
                raises ( TINAUsageCommonTypes::e_UsageError,
                         e_PaSBSetupError, e_NoSynchronousReqResp );

        // Modify the recovery and participation criteria
        void modifyCritieriaProviderPaSBReq(
                in TINACommonTypes::t_PartyId reqId, // Requesters id
                in TINAStreamCommonTypes::t_SBId sbId, // stream binding id
                in TINAStreamCommonTypes::t_SBSuccessCriteria criteria,// For SB
                in TINAStreamCommonTypes::t_SBRecover recActions, //
Actions recover stream binding
                in TINAPaSBTypes::t_PCriteriaList newPCriteria)
                        // List of participants with their new criteria
                raises ( TINAUsageCommonTypes::e_UsageError,
                         e_PaSBSetupError );
```

```
        // Notify the stream binding of a single participants relation to
        // the stream binding. (Not sure if needed...)
        void notifyProviderPaSBReq(
                in TINACommonTypes::t_PartyId reqId, // Requesters id
                in TINAStreamCommonTypes::t_SBId sbId, // stream binding id
                in TINAStreamCommonTypes::t_SFEPServDescList myStatus)
                raises ( TINAUsageCommonTypes::e_UsageError,
                         e_PaSBQueryError );

        // Allows a SB member to withdraw SFEPs (or SIs) from a stream binding
        //   this causes the rerunning of the binding algorithm
        void withdrawSFEPsProviderPaSBReq(
                in TINACommonTypes::t_PartyId reqId, // Requesters id
                in TINAStreamCommonTypes::t_SBId sbId, // stream binding id
                in TINASBCommSCommonTypes::t_SFEPNameList fepList, //
 SFEPs to withdraw (local only)
                out TINAStreamCommonTypes::t_SBBindState status) //
 Details of stream binding
                raises ( TINAUsageCommonTypes::e_UsageError,
                         e_PaSBSetupError, e_NoSynchronousReqResp);

        // Allows a SB member to register new SFEPs with a stream binding
        //   this causes the rerunning of the binding algorithm
        void registerSFEPsProviderPaSBReq(
                in TINACommonTypes::t_PartyId reqId, // Requesters id
                in TINAStreamCommonTypes::t_SBId sbId, // stream binding id
                in TINAStreamCommonTypes::t_SFEPServDescList fepList,//
 SFEPs to register with SB
                out TINAStreamCommonTypes::t_SBBindState status) //
 Details of stream binding
                raises ( TINAUsageCommonTypes::e_UsageError,
                         e_PaSBSetupError, e_NoSynchronousReqResp);

        // Rebind: allows a SB to rebind if external session factors
        //   affecting stream binding membership have occurred.
        void rebindProviderPaSBReq(
                in TINACommonTypes::t_PartyId reqId, // Requesters id
                in TINAStreamCommonTypes::t_SBId sbI, // Stream binding id
                out TINAStreamCommonTypes::t_SBBindState status) //
 Details of stream binding
                raises ( TINAUsageCommonTypes::e_UsageError,
                         e_PaSBSetupError, e_NoSynchronousReqResp);

        // List stream bindings in the service session
        void listProviderPaSBReq(
                in TINACommonTypes::t_PartyId reqId, // Requesters id
                in boolean all, // All stream bindings?: set true if all
                in TINACommonTypes::t_ElementIdList particpants,
                                // List stream bindings with these participants
                out TINAStreamCommonTypes::t_SBIdList sbList ) // List of SB ids
                raises ( TINAUsageCommonTypes::e_UsageError,
                         e_PaSBQueryError );

        // Get information on a particular stream binding
        void getInfoProviderPaSBReq(
                in TINACommonTypes::t_PartyId reqId, // Requesters id
                in TINAStreamCommonTypes::t_SBId sbId, // Stream binding id
                out TINAPaSBTypes::t_SBDesc thisSB) // Description of a stream binding
                raises ( TINAUsageCommonTypes::e_UsageError,
                         e_PaSBQueryError );
};


};

#endif // TINAScsSSMProviderPaSBUsage_IDL
```

2.25 TINAScsSSMProviderVotingUsage

```
// FILE: TINAScsSSMProviderVotingUsage.idl
//
// IDL for Service Session Manager
//
// Author: Per Fly Hansen (Tele Danmark)
// Creation date: September 23rd, 1997
// Modification date: September 23rd, 1997
//

#ifndef TINAScsSSMProviderVotingUsage_IDL
#define TINAScsSSMProviderVotingUsage_IDL

#include "TINACommonTypes.idl"
#include "TINAUsageCommonTypes.idl"

module TINAScsSSMProviderVotingUsage {

enum t_VoteResponse {
        NoVote,
        Agree,
        Disagree,
        Abstain
};

typedef short t_VoteValue;      // use is Service dependant
                                // could be used to weight the response,
                                // but effect is service dependant

struct t_Vote {
        t_VoteResponse response;
        t_VoteValue value;                      // may be ignored
};

enum t_VoteErrorCode {
        UnknownVoteError,
        VoteTooLate
};

exception e_VoteError {
        t_VoteErrorCode errorCode;
};

interface i_ProviderVotingReq {

        void voteReq(
                  in TINACommonTypes::t_PartyId reqId,
                  in TINAUsageCommonTypes::t_IndId indId,
                  in t_Vote vote
        ) raises (
                  TINAUsageCommonTypes::e_UsageError,
                  TINAUsageCommonTypes::e_IndError,
                  e_VoteError
        );
  };      // interface i_VotingReq
};
#endif /* TINAScsSSMProviderVotingReq_IDL */
```

2.26 TINAScsUSMInit

```
// FILE: TINAScsUSMInit.idl
//
// VERSION: 1
// DATE 21 August 97
//
// IDL for Usage Sesion Manager
// for the TINA- SCS
//
```

```
// AUTHOR: Martin Yates
//
// COMMENTS:
//
// MODIFICATIONS: PFH Sep. 24th 97
//
//
//

#ifndef TINAScsUSMInit_IDL
#define  TINAScsUSMInit_IDL

#include "TINACommonTypes.idl"
// #include "TINAUsageCommonTypes.idl" //PFH
// #include "TINAAccessCommonTypes.idl" //PFH

module TINAScsUSMInit{


/** properties to be interpreted by the USM, when initialising a service
* defined Property names:
* None defined at present
**/

typedef TINACommonTypes::t_PropertyList t_InitialSSProperties;

exception e_InitSSPropertyError {
        // use the errorCodes as for e_PropertyError
        TINACommonTypes::t_PropertyErrorStruct propertyError;
};


enum t_InitialiseUSMErrorCode {
        CannotCreateInterfaces,
        BindtoSSMFailed,
        initializeUnknownError
};


exception e_InitUSMError {
        t_InitialiseUSMErrorCode errorCode;

};

        interface i_Init {

                /
** Initialises the USM with session and service properties and return IRs that are newly in
stantiated
                **/
                void initialise(
                        in t_InitialSSProperties initProperties,
                        in TINACommonTypes::t_PartyId partyId, //PFH
                          /* Interfaces on UA and SSM: */ //PFH
                        in TINACommonTypes::t_InterfaceList infs, //PFH
                        out TINACommonTypes::t_InterfaceList initialUSMInfs,
                        out TINACommonTypes::t_ParticipantSecretId secretId //PFH
                ) raises(
                        e_InitSSPropertyError,
                        e_InitUSMError
                );

        }; //i_Init

};  //module TINAScsUSMInit

#endif   // TINAScsUSMInit_IDL
```

2.27 TINAScsUSMIntra

```
// FILE: TINAScsUSMIntra.idl
//
// VERSION: 1
// DATE 21 August 97
//
// IDL for Usage Sesion Manager
// for the TINA- SCS
//
// AUTHOR: Martin Yates
//
// COMMENTS:
//
// MODIFICATIONS:
//
//
//

#ifndef TINAScsUSMIntra_IDL
#define  TINAScsUSMIntra_IDL

#include "TINACommonTypes.idl"
#include "TINAUsageCommonTypes.idl"
#include "TINAAccessCommonTypes.idl"
#include "TINAProviderAccess.idl"
#include "TINAScsAmcObject.idl"
#include "TINAScsMgmtCtxt.idl"

/**
*       This module defines a subset of USM interfaces internal to the domain
*       supporting the USM.  but which are not specified in Ret feature sets or
*       their equivalents between the USM and SSM.
*       Interfaces here include: Accounting, Management Context, Resume, Session Contol
**/

module TINAScsUSMIntra{


// t_InitialSSProperties properties:
//      properties to be interpreted by the USM, when initialising a service
// defined Property names:
//      None defined at present
typedef TINACommonTypes::t_PropertyList t_InitialSSProperties;

exception e_initSSPropertyError {
        // use the errorCodes as for e_PropertyError
        TINACommonTypes::t_PropertyErrorStruct propertyError;
};

exception e_StartServiceSSPropertyError {
        // use the errorCodes as for e_PropertyError
        TINACommonTypes::t_PropertyErrorStruct propertyError;
};


enum t_SessionControlErrorCode {
        PartyIdInvalid,
        SessionIdInvalid,
        PartyDeniedAuthority,
        ControlUnavailable,
        SecurityContextNotSatisfied,
        sessionUnknownError
};

exception e_SessionControlError {
        t_SessionControlErrorCode errorCode;
};

enum t_InitialiseUSMErrorCode {
```

```
        CannotCreateInterfaces,
        BindtoSSMFailed,
        initializeUnknownError
};

exception e_initUSMError {
        t_InitialiseUSMErrorCode errorCode;
};

//INTERFACE DEFINITIONS START HERE

        /** Session control of the participation and session lifecycle, typical client UA.
        **/

        interface i_SessionCtrl {

                /
** Ends the entire session for all participants, requires authorisation. When
                        successful results in Ret interfaces of the USM becoming invalid.
                **/

                void endSession (
                        in TINACommonTypes::t_SessionId sessionId,
                        in TINACommonTypes:: t_PartyId Party_Id,
                        out any                 accountInfo
                ) raises (
                        e_SessionControlError
                );

                /** Ends only the participation of the party represented by the USM.  When
                        sucessful results in Ret interfaces of the USM becoming invalid
                **/

                void endMyParticipation (
                        in TINACommonTypes::t_SessionId sessionId,
                        in TINACommonTypes:: t_PartyId Party_Id,
                        out any                 accountInfo
                ) raises (
                        e_SessionControlError
                );

                /
** Suspends the entire session for all all participants. When sucsessful results
                 in Ret interfaces of the USM becoming invalid and a new IR capable of
                 resuming participation.
                **/

                void suspendSession (
                        in TINACommonTypes::t_SessionId         sessionId,
                        in TINACommonTypes:: t_PartyId          Party_Id,
                        out  TINACommonTypes::t_InterfaceList  resumeItfRef,
                        out any                 accountInfo
                ) raises (
                        e_SessionControlError
                );

                /
**Suspends the participation of the party represented by the USM. When sucsessful results
                 in Ret interfaces of the USM becoming invalid and a new IR capable of
                 resuming participation.
                **/

                void suspendMyParticipation (
                        in TINACommonTypes::t_SessionId         sessionId,
                        in TINACommonTypes:: t_PartyId          Party_Id,
                        out  TINACommonTypes::t_InterfaceList  resumeItfRef,
                        out any                                 accountInfo
                ) raises (
                        e_SessionControlError
                );
```

```
        }; //i_SessionCtrl

        /** Initialises the USM with session and service properties and return its
        * interface references when newly instantiated. Client is SF.
        **/


        /** Allows the SSM to fully resume a suspended service session by resuming the USMs
        **/

         interface i_Resume {

                /
**Resumes the USM after the entire session has been suspended and is called
                ONLY on the USM representing the user that has issued the resume request.
                The other USM in the session are reactivated by using sessionResuming()
                **/

                void resumeSession (
                        in TINACommonTypes::t_PartyId partyId,
                        in  TINACommonTypes::t_SessionId userSessionId,
                        in TINACommonTypes::t_InterfaceStruct ssmItfs,
                        in TINAProviderAccess::t_ApplicationInfo app
                ) raises (
                        TINAProviderAccess::e_SessionError,
                        TINAProviderAccess::e_ApplicationInfoError
                );

                void sessionResumed (
                        in TINACommonTypes::t_PartyId partyId,
                        in  TINACommonTypes::t_SessionId userSessionId,
                        in TINACommonTypes::t_InterfaceStruct usmItfs,
                        in TINAProviderAccess::t_ApplicationInfo app,
                        out TINAAccessCommonTypes::t_SessionInfo sessionInfo
                ) raises (
                        TINAProviderAccess::e_SessionError,
                        TINAProviderAccess::e_ApplicationInfoError
                );

                /
**  Resume participation of a specific party in an active service session.
                **/

                void resumeMyParticipation (
                        in TINACommonTypes::t_PartyId partyId,
                        in  TINACommonTypes::t_SessionId userSessionId,
                        in TINACommonTypes::t_InterfaceStruct ssmItfs,
                        in TINAProviderAccess::t_ApplicationInfo app
                )
                raises (
                        TINAProviderAccess::e_SessionError,
                        TINAProviderAccess::e_ApplicationInfoError
                );

        }; // i_Resume

        /** Receives of accounting events pushed from SSM. The USM forms a part of
         accounting management ladder in the retailer domain.
        **/

        interface i_AccountingPush : TINAScsAmcObject :: i_AmcLadderElement {
                //no additional operations defined
        }; //i_AccountingPush


        /** Controls the delivery of USM generated accounting events push to
        *accouting interface on the UA. Client UA
        **/

        interface i_AccountingPushMgmt: TINAScsAmcObject :: i_AccObjectManagement {
                // no additional operations defined
```

```
        }; //i_AccountingPushMgmt



        /** added by Revision 1.0.1
        * The interface is immature and scenarios that show how the interfaces is used
        * in the context of session wide components has not yet been determined.
        * Treat with caution.
        **/

        interface i_MgmtCtxt {

                /** bind the component to a context for the session which comprises the
                * list of management contexts and their respective properties and values
                * session explicity passed in the operation.
                **/
                boolean bindList (
                        in TINAScsMgmtCtxt::t_MgmtCtxtList contexts
                ) raises (
                        TINAScsMgmtCtxt::e_usmMgmtCtxt
                );

                /
** bind the component to a context for the session which is identified by an ID
                * which resolves elsewhere to a list of management contexts.
                **/
                boolean bindID (
                         in TINAScsMgmtCtxt::t_MgmtCtxtID contexts
                ) raises (
                        TINAScsMgmtCtxt::e_usmMgmtCtxt
                );

                /** unbind the component to a context for the session which comprises the
                *       list of management contexts and their respective properties and val
ues
                *       session explicity passed in the operation.
                **/

                boolean unbindList (
                        in TINAScsMgmtCtxt::t_MgmtCtxtList contexts
                ) raises (
                        TINAScsMgmtCtxt::e_usmMgmtCtxt
                );

                /
** unbind the component to a context for the session which is identified by an ID
                *       which resolves elsewhere to a list of management contexts.
                **/

                boolean unbindID (
                    in TINAScsMgmtCtxt::t_MgmtCtxtID contexts
                ) raises (
                        TINAScsMgmtCtxt::e_usmMgmtCtxt
                );

                /
** REbind the component to a previously used context for the session which comprises the
                *       list of management contexts and their respective properties and val
ues
                *       session explicity passed in the operation.
                **/
                boolean rebindList (
                    in TINAScsMgmtCtxt::t_MgmtCtxtList contexts
                ) raises (
                        TINAScsMgmtCtxt::e_usmMgmtCtxt
                );

                /
** Rebind the component to a previously used context for the session which is identified by
 an ID
```

```
                *       which resolves elsewhere to a list of management contexts.
                **/
                boolean rebindID(
                    in TINAScsMgmtCtxt::t_MgmtCtxtID contexts
                ) raises (
                        TINAScsMgmtCtxt::e_usmMgmtCtxt
                );

        };      // i_MgmtCtxt


};  //module TINAScsUSMIntra

#endif   // TINAScsUSMIntra_IDL
```

2.28 TINAScsUSMPartyBasicExtUsage

```
// FILE: TINAScsUSMPartyBasicExtUsage.idl
//
// VERSION: 1
// DATE 21 August 97
//
// IDL for Usage Sesion Manager
// for the TINA- SCS
//
// AUTHOR: Martin Yates
//
// COMMENTS:
//
// MODIFICATIONS:
//
//
//

#ifndef TINAScsUSMPartyBasicExtUsage_IDL
#define  TINAScsUSMPartyBasicExtUsage_IDL

#include "TINACommonTypes.idl"
#include "TINAUsageCommonTypes.idl"
#include "TINAAccessCommonTypes.idl"
#include "TINAPartyBasicExtUsage.idl"

/**
*       This module defines interfaces supported by the USM and typically used by the SSM
*       component. The interfaces are used when the USM/
ssUAP also support the PartyBasicExtUsage
*       Feature set across Ret.
**/

module TINAScsUSMPartyBasicExtUsage{

        interface i_PartyBasicExtReq {
        };

        // YUCK not sure about next one
        interface i_PartyGetInterfaces {
        };
};

  //module TINAScsUSMPartyBasicExtUsage_IDL

#endif   // TINAScsUSMPartyBasicExtUsage
```

2.29 TINAScsUSMPartyControlSRUsage

```
// FILE: TINAScsUSMPartyControlSRUsage.idl
//
// VERSION: 1
```

```
// DATE 21 August 97
//
// IDL for Usage Sesion Manager
// for the TINA- SCS
//
// AUTHOR: Martin Yates
//
// COMMENTS:
//
// MODIFICATIONS:
//
//
//

#ifndef TINAScsUSMPartyControlSRUsage_IDL
#define  TINAScsUSMPartyControlSRUsage_IDL

#include "TINACommonTypes.idl"
#include "TINAUsageCommonTypes.idl"
#include "TINAAccessCommonTypes.idl"
#include "TINAPartyControlSRUsage.idl"
#include "TINAControlSRTypes.idl"
#include "TINAScsCommonTypes.idl"


/**
*       This module defines interfaces supported by the USM and typically used by the SSM
*       component. The interfaces are most likely to be used when the USM/
ssUAP also support
*       the PartyControlSRUsage feature set across Ret.
**/

module TINAScsUSMPartyControlSRUsage{

interface i_PartyControlSRInd {

        void setControlInd (
                in TINAScsCommonTypes::t_PartyIdListHandler partyIdList,
                in TINAUsageCommonTypes::t_IndId indId,
                in TINAControlSRTypes::t_ControlInfo controlInfo
        ) raises (
                TINAUsageCommonTypes::e_PartyDomainError,
                TINAUsageCommonTypes::e_PartyError,
                TINAScsCommonTypes::e_PartyIdListError
        );

};

interface i_PartyControlSRInfo {

oneway  void setControlInfo (
                in TINAScsCommonTypes::t_PartyIdListHandler partyIdList,
                in TINAControlSRTypes::t_ControlInfo controlInfo
        ) raises (
                TINAScsCommonTypes::e_PartyIdListError
        );

};

};  //module TINAScsUSMPartyControlSRUsage

#endif   // TINAScsUSMPartyControlSRUsage_IDL
```

2.30 TINAScsUSMPartyMultipartyIndUsage

```
// FILE: TINAScsUSMPartyMultipartyIndUsage.idl
//
// VERSION: 1
// DATE 21 August 97
```

```
//
// IDL for Usage Sesion Manager
// for the TINA- SCS
//
// AUTHOR: Martin Yates
//
// COMMENTS:
//
// MODIFICATIONS:
//
//
//

#ifndef TINAScsUSMPartyMultipartyIndUsage_IDL
#define  TINAScsUSMPartyMultipartyIndUsage_IDL

#include "TINACommonTypes.idl"
#include "TINAUsageCommonTypes.idl"
#include "TINAAccessCommonTypes.idl"
#include "TINAPartyMultipartyUsage.idl"
#include "TINAPartyMultipartyIndUsage.idl"
#include "TINAScsCommonTypes.idl"



/**
*       This module defines interfaces supported by the USM and typically used by the SSM
*       component. The interfaces are most likely to be used when the USM/
ssUAP also support
*       the PartyMultipartyIndUsage feature set across Ret.
**/

module TINAScsUSMPartyMultipartyIndUsage{


   interface i_PartyMultipartyInd{

        void operationCancelled (
                in TINAScsCommonTypes::t_PartyIdListHandler indPartyList, /
** target partyIds **/
                in TINAUsageCommonTypes::t_IndId indId
        ) raises (
                TINAScsCommonTypes::e_PartyIdListError,
                TINAUsageCommonTypes::e_IndError
        );

        void modifyPartyTypeInd (
                in TINAScsCommonTypes::t_PartyIdListHandler indPartyList, /
** target partyIds **/
                in TINAUsageCommonTypes::t_IndId indId,
                in TINACommonTypes::t_PartyId reqPartyId,       /** originating party **/
                in TINAUsageCommonTypes::t_PartyDetails partyDetails
        ) raises (
                TINAScsCommonTypes::e_PartyIdListError,
                TINAUsageCommonTypes::e_PartyError
        );


        void endSessionInd (
                in TINAScsCommonTypes::t_PartyIdListHandler indPartyList, /
** target partyIds **/
                in TINAUsageCommonTypes::t_IndId indId,
                in TINACommonTypes::t_PartyId reqPartyId /** originating party **/
        ) raises (
                TINAScsCommonTypes::e_PartyIdListError,
                TINAUsageCommonTypes::e_PartyError
        );

        void endPartyInd (
                in TINAScsCommonTypes::t_PartyIdListHandler indPartyList, /
** target partyIds **/
```

```
                in TINAUsageCommonTypes::t_IndId indId,
                in TINACommonTypes::t_PartyId reqPartyId, /** originating party **/
                in TINACommonTypes::t_PartyId partyId
        ) raises (
                TINAScsCommonTypes::e_PartyIdListError,
                TINAUsageCommonTypes::e_PartyError
        );

        void suspendSessionInd (
                in TINAScsCommonTypes::t_PartyIdListHandler indPartyList, /
** target partyIds **/
                in TINAUsageCommonTypes::t_IndId indId,
                in TINACommonTypes::t_PartyId reqPartyId /** originating party **/
        ) raises (
                TINAScsCommonTypes::e_PartyIdListError,
                TINAUsageCommonTypes::e_PartyError
        );

        void resumeSessionInd(
                in TINAScsCommonTypes::t_PartyIdListHandler indPartyList, /
** target partyIds **/
                in TINAUsageCommonTypes::t_IndId indId,
                in TINACommonTypes::t_PartyId reqPartyId /** originating party resuming**/
        ) raises (
                TINAScsCommonTypes::e_PartyIdListError,
                TINAUsageCommonTypes::e_PartyError
        );

        void suspendPartyInd (
                in TINAScsCommonTypes::t_PartyIdListHandler indPartyList, /
** target partyIds **/
                in TINAUsageCommonTypes::t_IndId indId,
                in TINACommonTypes::t_PartyId reqPartyId, /** originating party **/
                in TINACommonTypes::t_PartyId partyId
        ) raises (
                TINAScsCommonTypes::e_PartyIdListError,
                TINAUsageCommonTypes::e_PartyError
        );

        void resumePartyInd (
                in TINAScsCommonTypes::t_PartyIdListHandler indPartyList, /
** target partyIds **/
                in TINAUsageCommonTypes::t_IndId indId,
                in TINACommonTypes::t_PartyId partyId  /** originating party resuming**/
        ) raises (
                TINAScsCommonTypes::e_PartyIdListError,
                TINAUsageCommonTypes::e_PartyError
        );

        void joinSessionInd (

                in TINAScsCommonTypes::t_PartyIdListHandler indPartyList, /
** target partyIds **/
                in TINAUsageCommonTypes::t_IndId indId,
                in TINACommonTypes::t_PartyId reqPartyId,               /
** originating party **/
                in TINACommonTypes::t_UserDetails userDetails
        ) raises (
                TINAScsCommonTypes::e_PartyIdListError,
                TINAUsageCommonTypes::e_PartyError,
                TINACommonTypes::e_UserDetailsError
        );

        void inviteUserInd (
                in TINAScsCommonTypes::t_PartyIdListHandler indPartyList, /
** target partyIds **/
                in TINAUsageCommonTypes::t_IndId indId,
                in TINACommonTypes::t_UserDetails userDetails,
                in TINACommonTypes::t_PartyId reqPartyId               /
** originating party **/
        ) raises (
```

```
                TINAScsCommonTypes::e_PartyIdListError,
                TINAUsageCommonTypes::e_PartyError,
                TINACommonTypes::e_UserDetailsError
        );

        void announceSessionInd (
                in TINAScsCommonTypes::t_PartyIdListHandler indPartyList, /
** target partyIds **/
                in TINAUsageCommonTypes::t_IndId indId,
                in TINACommonTypes::t_PartyId reqPartyId,              /
** originating party **/
                in TINACommonTypes::t_AnnouncementProperties announcement
        ) raises (
                TINAScsCommonTypes::e_PartyIdListError,
                TINAUsageCommonTypes::e_PartyError,
                TINAUsageCommonTypes::e_AnnouncementError
        );


   }; //i_PartyMultipartyInd

};  //module TINAScsUSMPartyMultipartyIndUsage

#endif   // TINAScsUSMPartyMultipartyIndUsage_IDL
```

2.31 TINAScsUSMPartyMultipartyUsage

```
/** FILE: TINAScsUSMPartyMultipartyUsage.idl    */
/**                                             */
/** VERSION: 1                                  */
/** DATE 21 August 97                           */
/**                                             */
/** IDL for Usage Sesion Manager                */
/** for the TINA- SCS                           */
/**                                             */
/** AUTHOR: Martin Yates                        */
/**                                             */
/** COMMENTS:                                   */
/**                                             */
/** MODIFICATIONS:                              */
/**                                             */
/**                                             */
/**                                             */

#ifndef TINAScsUSMPartyMultipartyUsage_IDL
#define  TINAScsUSMPartyMultipartyUsage_IDL

#include "TINACommonTypes.idl"
#include "TINAUsageCommonTypes.idl"
#include "TINAScsCommonTypes.idl"


/**
*       This module defines interfaces supported by the USM and typically used by the SSM
*       component. The interfaces are most likely to be used when the USM/
ssUAP also support
*       the PartyMultipartyIndUsage feature set across Ret.
**/

module TINAScsUSMPartyMultipartyUsage {

interface i_PartyMultipartyExe {

        /
** In the following operations strictly the excetion need only be invalid Party_id **/

        void modifyPartyTypeExe (
                in TINAUsageCommonTypes:: t_PartyType newType
        ) raises (
```

```
                TINAUsageCommonTypes:: e_PartyError,    /
** error in modification request  **/
                TINAScsCommonTypes:: e_PartySessionError
        );


        void endSessionExe (
                in TINACommonTypes:: t_PartyId partyId
        ) raises (
                TINAScsCommonTypes:: e_PartyIdError,
                TINAScsCommonTypes:: e_PartySessionError
        );

        void endPartyExe (
                in TINACommonTypes:: t_PartyId partyId
        ) raises (
                TINAScsCommonTypes:: e_PartyIdError,
                TINAScsCommonTypes:: e_PartySessionError
        );

        void suspendSessionExe (
                in TINACommonTypes:: t_PartyId partyId,
                in TINACommonTypes:: t_InterfaceList resumeIR
        ) raises (
                TINAScsCommonTypes:: e_PartyIdError,
                TINAScsCommonTypes:: e_PartySessionError
        );

        void suspendPartyExe (
                in TINACommonTypes:: t_PartyId partyId,
                in TINACommonTypes:: t_InterfaceList resumeIR
        ) raises (
                TINAScsCommonTypes:: e_PartyIdError,
                TINAScsCommonTypes:: e_PartySessionError
        );


};      /** interface i_PartyMultipartyExe  **/

/** The following mirror Ret but have been changed to raise user defined exceptions,
        because the interactions between SSM and USM are more critical than those across Re
t -
        which sere defined as oneway operations. The operations have been designed so that
        a single 'USM' could handle multiple parties (PD_USSs) if it were implemented this
way.
        This was considered less restictive on implementation options.
**/

interface i_PartyMultipartyInfo {


        void modifyPartyTypeInfo (
                in TINAScsCommonTypes:: t_PartyIdListHandler indPartyList, /
** target partyIds **/
                in TINACommonTypes:: t_PartyId partyId
        ) raises (
                TINAScsCommonTypes:: e_PartyIdListError,
                TINAUsageCommonTypes:: e_PartyError
        );


        void endPartyInfo (
                in TINAScsCommonTypes:: t_PartyIdListHandler indPartyList, /
** target partyIds **/
                in TINACommonTypes:: t_PartyId partyId
        ) raises (
                TINAScsCommonTypes:: e_PartyIdListError,
                TINAUsageCommonTypes:: e_PartyError
        );
```

```
        void suspendPartyInfo (
                in TINAScsCommonTypes:: t_PartyIdListHandler indPartyList, /
** target partyIds **/
                in TINACommonTypes:: t_PartyId partyId
        ) raises (
                TINAScsCommonTypes:: e_PartyIdListError,
                TINAUsageCommonTypes:: e_PartyError
        );


        void resumePartyInfo (
                in TINAScsCommonTypes:: t_PartyIdListHandler indPartyList, /
** target partyIds **/
                in TINACommonTypes:: t_PartyId partyId
        ) raises (
                TINAScsCommonTypes:: e_PartyIdListError,
                TINAUsageCommonTypes:: e_PartyError
        );


        void joinSessionInfo (
                in TINAScsCommonTypes:: t_PartyIdListHandler indPartyList, /
** target partyIds **/
                in TINAUsageCommonTypes:: t_PartyDetails partyDetails,
                in TINACommonTypes:: t_UserDetails userDetails
        ) raises (
                TINAScsCommonTypes:: e_PartyIdListError,
                TINAUsageCommonTypes:: e_PartyError,
                TINACommonTypes:: e_UserDetailsError
        );


        void inviteUserInfo (
                in TINAScsCommonTypes:: t_PartyIdListHandler indPartyList, /
** target partyIds **/
                in TINACommonTypes:: t_UserDetails userDetails,
                in TINAUsageCommonTypes:: t_InvitationId invitationId
        ) raises (
                TINAScsCommonTypes:: e_PartyIdListError,
                TINACommonTypes:: e_UserDetailsError
        );

        void inviteReplyInfo (
                in TINAScsCommonTypes:: t_PartyIdListHandler indPartyList, /
** target partyIds **/
                in TINAUsageCommonTypes:: t_InvitationId invitationId,
                in TINACommonTypes:: t_InvitationReply reply
        ) raises (
                TINAScsCommonTypes:: e_PartyIdListError
        );

        void announceSessionInfo (
                in TINAScsCommonTypes:: t_PartyIdListHandler indPartyList, /
** target partyIds **/
                in TINACommonTypes:: t_AnnouncementProperties announcement
        ) raises (
                TINAScsCommonTypes:: e_PartyIdListError,
                TINAUsageCommonTypes:: e_AnnouncementError
        );


};      /** interface i_PartyMultipartyInfo  */


};  /**module TINAScsUSMPartyMultipartyUsage  */

#endif   /** TINAScsUSMPartyMultipartyUsage_IDL  */
```

2.32 TINAScsUSMPartyPaSBIndUsage

```
#ifndef TINAScsUSMPartyPaSBIndUsage_IDL_
#define TINAScsUSMPartyPaSBIndUsage_IDL_

// FILE:
//   TINAScsUSMPartyPaSBIndUsage.idl
// DESCRIPTION:
//  TINA Participant Oriented Stream Binding with Indications
//  Feature Set Party Module
//
// Interfaces and types needed to support the participant
// oriented stream binding extended feature set for users.
// This extended feature set supports indications.
// Note: no extra interfaces need be supported by the provider.
// AUTHOR:
//   Stephanie Hogg
//   Koki NAKASHIRO, Carlo Licciardi
// CREATION DATE:
//  06/11/97
// UPDATES:
//

#include "TINAUsageCommonTypes.idl"
#include "TINAPaSBTypes.idl"
#include "TINAPartyPaSBIndUsage.idl"
#include "TINAScsCommonTypes.idl"

module TINAScsUSMPartyPaSBIndUsage {


// DESCRIPTION:
//   i_PartyPaSBInd interface
//   The participant oriented stream binding indication interface
//   Allows the stream binding to provider to report requested
//   operations to stream members.
//   If using the session graph model, this should be all parties
//   who have an ownership control relatin with the stream binding

// #include "PaSBCommonTypes.idl"
// #include "TINAUsageCommonTypes.idl"
// #include "PaSBIndErrorTypes.idl"


interface i_PartyPaSBInd
{
        // Request to establish a stream binding
        void addPartyPaSBInd(
                in TINAScsCommonTypes::t_PartyIdListHandler indPartyList, /
** target partyIDs **/
                in TINAUsageCommonTypes::t_IndId indId,
                        // Indication Id: used in voting to identify
                        // what is being voted on (i.e. this op)
                in TINAStreamCommonTypes::t_RequestId reqId,
                        // Request Id: identifies responses to op
                in TINAStreamCommonTypes::t_SBType reqType,//
 type identifier: overall type
                in TINASBCommSCommonTypes::t_MediaDescList media,//
 Overall media of assoc flow
                in TINAPaSBTypes::t_ParticipantIdList reqMembers,
                        // List of participants, each participant described
                in TINAStreamCommonTypes::t_SBSuccessCriteria criteria,
                        // What is necessary for this operation to succeed?
                in TINAStreamCommonTypes::t_SBRecover recActions)
                        // Actions on failure to recover stream binding
                        // Success of recovery criteria
                raises ( TINAUsageCommonTypes::e_PartyDomainError,
                         TINAUsageCommonTypes::e_IndError,
                         TINAPartyPaSBIndUsage:: e_PaSBIndError,
                         TINAScsCommonTypes::e_PartyIdListError );
```

```
        // Request to remove a stream binding
        void deletePartyPaSBInd(
                in TINAScsCommonTypes::t_PartyIdListHandler indPartyList, /
** target partyIDs **/
                in TINAUsageCommonTypes::t_IndId indId,
                in TINAStreamCommonTypes::t_RequestId reqId,
                in TINAStreamCommonTypes::t_SBId sbId) // stream binding id
                raises ( TINAUsageCommonTypes::e_PartyDomainError,
                         TINAUsageCommonTypes::e_IndError,
                         TINAPartyPaSBIndUsage:: e_PaSBIndError,
                         TINAScsCommonTypes::e_PartyIdListError );

        // Request to add participants to a stream binding
        void addParticipantsPartyPaSBInd(
                in TINAScsCommonTypes::t_PartyIdListHandler indPartyList, /
** target partyIDs **/
                in TINAUsageCommonTypes::t_IndId indId,
                in TINAStreamCommonTypes::t_RequestId reqId,
                in TINAStreamCommonTypes::t_SBId sbId, // stream binding id
                in TINAPaSBTypes::t_ParticipantIdList reqMembers) // alt: id list
                        // List of participants, each participant described
                raises ( TINAUsageCommonTypes::e_PartyDomainError,
                         TINAUsageCommonTypes::e_IndError,
                         TINAPartyPaSBIndUsage:: e_PaSBIndError,
                         TINAScsCommonTypes::e_PartyIdListError );

        // Request to remove participants from a stream binding
        void deleteParticipantsPartyPaSBInd(
                in TINAScsCommonTypes::t_PartyIdListHandler indPartyList, /
** target partyIDs **/
                in TINAUsageCommonTypes::t_IndId indId,
                in TINAStreamCommonTypes::t_RequestId reqId,
                in TINAStreamCommonTypes::t_SBId sbId, // stream binding id
                in boolean all, // All participants or listed participants?
                in TINAPaSBTypes::t_ParticipantIdList reqMembers)
                raises ( TINAUsageCommonTypes::e_PartyDomainError,
                         TINAUsageCommonTypes::e_IndError,
                         TINAPartyPaSBIndUsage:: e_PaSBIndError,
                         TINAScsCommonTypes::e_PartyIdListError );

        // Request to activate by participants (and opt. implicit flows)
        void activateParticipantsPartyPaSBInd(
                in TINAScsCommonTypes::t_PartyIdListHandler indPartyList, /
** target partyIDs **/
                in TINAUsageCommonTypes::t_IndId indId,
                in TINAStreamCommonTypes::t_RequestId reqId,
                in TINAStreamCommonTypes::t_SBId sbId, // stream binding id
                in boolean all, // All participants or listed participants?
                in TINAPaSBTypes::t_ParticipantIdList reqMembers,
                        // list of members to activate, valid if all=false
                in boolean allFlows,    // All sub types or listed sub types?
                in TINASBCommSCommonTypes::t_MediaDescList reqFlows)
                        // Media types to activate, valid if allFlows=false
                raises ( TINAUsageCommonTypes::e_PartyDomainError,
                         TINAUsageCommonTypes::e_IndError,
                         TINAPartyPaSBIndUsage:: e_PaSBIndError,
                         TINAScsCommonTypes::e_PartyIdListError );

        // Request to deactivate by participants (and opt. implicit flows)
        void deactiviateParticipantsPartyPaSBInd(
                in TINAScsCommonTypes::t_PartyIdListHandler indPartyList, /
** target partyIDs **/
                in TINAUsageCommonTypes::t_IndId indId,
                in TINAStreamCommonTypes::t_RequestId reqId,
                in TINAStreamCommonTypes::t_SBId sbId, // stream binding id
                in boolean all,// All participants or listed participants?
                in TINAPaSBTypes::t_ParticipantIdList reqMembers,
                        // list of members to deactivate, valid if all=false
                in boolean allFlows,    // All sub types or listed sub types?
                in TINASBCommSCommonTypes::t_MediaDescList reqFlows)
```

```
                        // Media types to deactivate, valid if allFlows=false
                raises ( TINAUsageCommonTypes::e_PartyDomainError,
                         TINAUsageCommonTypes::e_IndError,
                         TINAPartyPaSBIndUsage:: e_PaSBIndError,
                         TINAScsCommonTypes::e_PartyIdListError );

        // Request to modify participation
        void modifyParticipantsPartyPaSBInd(
                in TINAScsCommonTypes::t_PartyIdListHandler indPartyList, /
** target partyIDs **/
                in TINAUsageCommonTypes::t_IndId indId,
                in TINAStreamCommonTypes::t_RequestId reqId,
                in TINAStreamCommonTypes::t_SBId sbId, // stream binding id
                in boolean all,// All participants or listed participants?
                in TINAPaSBTypes::t_ParticipantIdList reqMembers,//
                        // list of participants to modify, valid if all=false
                in TINASBCommSCommonTypes::t_MediaDescList newTypes,//
 Flow types to add or modify
                in TINASBCommSCommonTypes::t_MediaDescList oldTypes, //
 Media types to be removed
                in TINASBCommSCommonTypes::t_MediaChangeDescList modTypes)//
 Media types to modify
                raises ( TINAUsageCommonTypes::e_PartyDomainError,
                         TINAUsageCommonTypes::e_IndError,
                         TINAPartyPaSBIndUsage:: e_PaSBIndError,
                         TINAScsCommonTypes::e_PartyIdListError );

        // Modify the recovery and participation criteria
        void modifyCritieriaPartyPaSBInd(
                in TINAScsCommonTypes::t_PartyIdListHandler indPartyList, /
** target partyIDs **/
                in TINAUsageCommonTypes::t_IndId indId,
                in TINAStreamCommonTypes::t_RequestId reqId,
                in TINAStreamCommonTypes::t_SBId sbId, // stream binding id
                in TINAStreamCommonTypes::t_SBSuccessCriteria criteria,// For SB
                in TINAStreamCommonTypes::t_SBRecover recActions, //
 Actions recover stream binding
                in TINAPaSBTypes::t_PCriteriaList newPCriteria)
                        // List of participants with their new criteria
                raises ( TINAUsageCommonTypes::e_PartyDomainError,
                         TINAUsageCommonTypes::e_IndError,
                         TINAPartyPaSBIndUsage:: e_PaSBIndError,
                         TINAScsCommonTypes::e_PartyIdListError );

        // Request from a participant to withdraw SFEPs from SB
        void withdrawSFEPsPartyPaSBInd(
                in TINAScsCommonTypes::t_PartyIdListHandler indPartyList, /
** target partyIDs **/
                in TINAUsageCommonTypes::t_IndId indId,
                in TINAStreamCommonTypes::t_SBId sbId, // stream binding id
                in TINASBCommSCommonTypes::t_SFEPNameList fepList) //
 SFEPs to withdraw (local only)
                raises ( TINAUsageCommonTypes::e_PartyDomainError,
                         TINAUsageCommonTypes::e_IndError,
                         TINAPartyPaSBIndUsage:: e_PaSBIndError,
                         TINAScsCommonTypes::e_PartyIdListError );

        void registerSFEPsPartyPaSBInd( // Allows a particpant register SFEPs
                in TINAScsCommonTypes::t_PartyIdListHandler indPartyList, /
** target partyIDs **/
                in TINAUsageCommonTypes::t_IndId indId,
                in TINAStreamCommonTypes::t_SBId sbId, // stream binding id
                in TINAStreamCommonTypes::t_SFEPServDescList fepList)//
 SFEPs to register with SB
                raises ( TINAUsageCommonTypes::e_PartyDomainError,
                         TINAUsageCommonTypes::e_IndError,
                         TINAPartyPaSBIndUsage:: e_PaSBIndError,
                         TINAScsCommonTypes::e_PartyIdListError );

        void rebindPaSBFSInd(
                in TINAScsCommonTypes::t_PartyIdListHandler indPartyList, /
```

```
** target partyIDs **/
                in TINAUsageCommonTypes::t_IndId indId,
                in TINAStreamCommonTypes::t_RequestId reqId,
                in TINAStreamCommonTypes::t_SBId sbI) // Stream binding id
                raises ( TINAUsageCommonTypes::e_PartyDomainError,
                         TINAUsageCommonTypes::e_IndError,
                         TINAPartyPaSBIndUsage:: e_PaSBIndError,
                         TINAScsCommonTypes::e_PartyIdListError );
};


};

#endif
       // _TINA_PARTY_PASB_IND_FS_IDL_
```

2.33 TINAScsUSMPartyPaSBUsage

```
#ifndef TINAScsUSMPartyPaSBUsage_IDL_
#define TINAScsUSMPartyPaSBUsage_IDL_


// FILE:
//   TINAScsUSMPartyPaSBUsage.idl
// DESCRIPTION:
//  TINA Participant Oriented Stream Binding Feature Set
//  Party Module
//
// Interfaces and types needed to support the participant
// oriented stream binding feature set for users.
// AUTHOR:
//   Stephanie Hogg
//   Koki NAKASHIRO, Carlo Licciardi
// CREATION DATE:
//  06/11/97
// UPDATES:
//

#include "TINAPaSBTypes.idl"
#include "TINAPartyPaSBUsage.idl"
#include "TINAScsCommonTypes.idl"

module TINAScsUSMPartyPaSBUsage {

// #include "PaSBCommonTypes.idl"
// #include "PaSBPartyErrorTypes.idl" $$s001$$

// DESCRIPTION:
//   Participant oriented stream binding party error types
//   The error codes and exception types associated with the
//   party part of the normal particiapant oriented stream binding.

// #include "StreamCommonTypes.idl"


// DESCRIPTION:
//   i_PartyPaSBExe interface
//   The participant oriented stream binding exe interface
//   Makes exe requests on stream binding members.
//   The set of exe requests is simpler than the full set of
//   stream binding requests as the same set of exes can support
//   a number of request operations.
//
//   The exe interface allows a provider to request a session member
//   join - become a stream binding member
//   leave - end participation in the stream binding
//   modify - modify participation  i.e. change MediaTypes supported
//   modify criteria - modify their local success and recover criteria
```

```
// #include "PaSBCommonTypes.idl"
// #include "PaSBPartyErrorTypes.idl"

interface i_PartyPaSBExe
{
        // This interface stream binding responses and notifications to
        // parties or resources which are partipants in a stream binding

        // Join: Request to a participant to join the SBFS and
        // return supporting SI reference and information
        void joinPartyPaSBExe(
                in TINACommonTypes::t_PartyId partyId, /** target partyId **/
                in TINAStreamCommonTypes::t_SBId sbId, // ID stream binding to join
                in TINAStreamCommonTypes::t_SBType reqType,// Overall type
                in TINASBCommSCommonTypes::t_MediaDescList media,// Overall req of flows
                in TINAPaSBTypes::t_ParticipantIdList others,// other members id
                in TINAPaSBTypes::t_ParticipantDesc reqParticipation,
                        // Specification of required participation
                        // for this session member
                in TINAStreamCommonTypes::t_RequestId reqId,//
 op id used for later info ops.
                out TINAStreamCommonTypes::t_SFEPServDescList participantSIs)
                        // Description of type: include SI ref and
                        // description, deviation from requested type
                raises ( TINAUsageCommonTypes::e_PartyDomainError,
                         TINAPartyPaSBUsage::e_PaSBPartySetupError,
                         TINAScsCommonTypes::e_PartyIdError );

        // Leave: Request to a participant to leave the SBFS
        void leavePartyPaSBExe(
                in TINACommonTypes::t_PartyId partyId, /** target partyId **/
                in TINAStreamCommonTypes::t_SBId sbId, // ID stream binding to leave
                in TINAStreamCommonTypes::t_RequestId reqId) //
 op id used for later info ops.
                raises ( TINAUsageCommonTypes::e_PartyDomainError,
                         TINAPartyPaSBUsage::e_PaSBPartyExeError,
                         TINAScsCommonTypes::e_PartyIdError );

        // Modify: Request to a change participation, including QoS,
        // Supported media types (large scale changes)
        void modifyPartyPaSBExe(
                in TINACommonTypes::t_PartyId partyId, /** target partyId **/
                in TINAStreamCommonTypes::t_SBId sbId, // ID stream binding
                in TINASBCommSCommonTypes::t_MediaDescList newTypes, //
 Media types to add or modify
                in TINASBCommSCommonTypes::t_MediaDescList oldTypes, //
 Media types to be removed
                in TINASBCommSCommonTypes::t_MediaChangeDescList modTypes,//
 Media types to modify
                in TINAStreamCommonTypes::t_RequestId reqId,//
 op id used for later info ops.
                out TINAStreamCommonTypes::t_SFEPServDescList participantSIs) //
New participation info
                raises ( TINAUsageCommonTypes::e_PartyDomainError,
                         TINAPartyPaSBUsage::e_PaSBPartySetupError,
                         TINAScsCommonTypes::e_PartyIdError );

        // Change administrative status (i.e. activate/deactivate
        void changeStatePartyPaSBExe(
                in TINACommonTypes::t_PartyId partyId, /** target partyId **/
                in TINAStreamCommonTypes::t_SBId sbId, // ID stream binding
                in TINASBCommSCommonTypes::t_AdministrativeState state, //
 New state for participant
                in boolean allFlows,    // All sub types or listed sub types?
                in TINASBCommSCommonTypes::t_MediaDescList reqFlows,
                        // Media types to be deactivated, valid allFlows=false
                in TINAStreamCommonTypes::t_RequestId reqId)//
 op id used for later info ops.
                raises ( TINAUsageCommonTypes::e_PartyDomainError,
                         TINAPartyPaSBUsage::e_PaSBPartyExeError,
                         TINAScsCommonTypes::e_PartyIdError );
```

```
        // Change participation criteria (success and recovery)
        void modifyCriteriaPartyPaSBExe(
                in TINACommonTypes::t_PartyId partyId, /** target partyId **/
                in TINAStreamCommonTypes::t_SBId sbId, // ID stream binding
                in TINAPaSBTypes::t_ParticipantCriteria newPCriteria)
                raises ( TINAUsageCommonTypes::e_PartyDomainError,
                         TINAPartyPaSBUsage::e_PaSBPartySetupError,
                         TINAScsCommonTypes::e_PartyIdError );
};

// #include "i_PartyPaSBInfo.idl" $$s001$$

//   i_PartyPaSBInfo interface
//   The participant oriented stream binding information interface
//   Allows status reports on a synchronous operations,
//   the distribution of SIs (in a very simple way),
//   the informs on the widthdrawals of SIs (and other elements)
//   and the notification of communication errors
///  Based on the general stream interface.
// AUTHOR:
//   Stephanie Hogg
// CREATION DATE:
//  20/08/97
// UPDATES:
//

// #include "i_GeneralStreamInfo.idl"

// DESCRIPTION:
//   i_GeneralStreamInfo interface
//   A general stream information interface
//   Allows status reports on a synchronous operations,
//   the distribution of SIs (in a very simple way),
//   the informs on the widthdrawals of SIs (and other elements)
//   and the notification of communication errors
// AUTHOR:
//   Stephanie Hogg
// CREATION DATE:
//  20/08/97
// UPDATES:
//

// #include "StreamCommonTypes.idl"

interface i_GeneralStreamInfo {

        // Report unexpected error with binding during normal operations
        oneway void notifyGSInfo(
                in TINASBCommSCommonTypes::t_Notification event);

        // Update on error
        oneway void notifyUpdateGSInfo(
                in TINASBCommSCommonTypes::t_NotifyIdentifier changedEvent,// identifier
                in TINASBCommSCommonTypes::t_StatusInfo eventChange); //
 Changed parameters

        // cancellation of previous error:
        // for those notification for which this is relevent
        oneway void notifyCancelGSInfo(
                in TINASBCommSCommonTypes::t_NotifyIdentifier changedEvent);
};

interface i_PartyGeneralStreamInfo : i_GeneralStreamInfo {

        // Confirms success of an asynchronous operation
        // identified by reqId
        oneway void confirmPartyGSInfo(
                in TINAScsCommonTypes::t_PartyIdListHandler infoPartyList, /
** target partyIDs **/
                in TINAStreamCommonTypes::t_RequestId reqId,
```

```
                in TINAStreamCommonTypes::t_RequestType reqType,
                in TINAStreamCommonTypes::t_SBBindState info);

        // Reports failure of an asynchronous operation
        // identified by reqId
        oneway void failurePartyGSInfo(
                in TINAScsCommonTypes::t_PartyIdListHandler infoPartyList, /
** target partyIDs **/
                in TINAStreamCommonTypes::t_RequestId reqId,
                in TINAStreamCommonTypes::t_RequestType reqType,
                in TINAStreamCommonTypes::t_FailureCode  error, // cause of failure
                in boolean additionalInfo, // Additional info flag
                in TINAStreamCommonTypes::t_ReqProblem  reqProblem);
                                // Elements causing problem (optional)
                                // Only valid if additional info flag set

        // Distributes SIs to SB members
        // SIs group SFEPs and identify the associated participant
        // this a useful way of distributing SFEP data
        oneway void SIDistribPartyGSInfo(
                in TINAScsCommonTypes::t_PartyIdListHandler infoPartyList, /
** target partyIDs **/
                in TINAStreamCommonTypes::t_SBId sbId,
                in TINAStreamCommonTypes::t_SIDescList newSIs);
                // This allows participants to know which SIs are available
                // and who they belong to.

        // Distributes SFEPs to SB members
        // Alternative to the above: could be useful if SI is already known
        oneway void SFEPDistribPartyGSInfo(
                in TINAScsCommonTypes::t_PartyIdListHandler infoPartyList, /
** target partyIDs **/
                in TINAStreamCommonTypes::t_SBId sbId,
                in TINAStreamCommonTypes::t_SFEPServDescList newSFEPs);
                // This allows participants to know which SFEPS are available

        // Notify withdrawal of elements to an SB
        // Elements could be SFEPs, SIs, SFCs, SB members
        oneway void notifyWithdrawnElementsPartyGSInfo(
                in TINAScsCommonTypes::t_PartyIdListHandler infoPartyList, /
** target partyIDs **/
                in TINAStreamCommonTypes::t_SBId sbId,
                in TINACommonTypes::t_ElementIdList gone);
};

interface i_PartyPaSBInfo : i_GeneralStreamInfo
{
        // No additional operations or attributes
        // Identified (at least not yet...)
};


};

#endif
       // _TINA_PARTY_PASB_FS_IDL_
```

2.34 TINAScsUSMPartyVotingUsage

```
// FILE: TINAScsUSMPartyVotingUsage.idl
//
// VERSION: 1
// DATE 21 August 97
//
// IDL for Usage Sesion Manager
// for the TINA- SCS
//
// AUTHOR: Martin Yates
//
```

```
// COMMENTS:
//
// MODIFICATIONS:
//
//
//

#ifndef TINAScsUSMPartyVotingUsage_IDL
#define  TINAScsUSMPartyVotingUsage_IDL

#include "TINACommonTypes.idl"
#include "TINAUsageCommonTypes.idl"
#include "TINAAccessCommonTypes.idl"
#include "TINAPartyVotingUsage.idl"

/**
*       This module defines interfaces supported by the USM and typically used by the SSM
*       component. The interfaces are most likely to be used when the USM/
ssUAP also support
*       the PartyVotingUsage feature set across Ret.
**/

module TINAScsUSMPartyVotingUsage{

        interface i_PartyVotingInfo {
        };

};  //module TINAScsUSMPartyVotingUsage

#endif   // TINAScsUSMPartyVotingUsage_IDL
```

2.35 PLATyToolsFix

```
#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

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

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 operationalState;          // $$s004$$
  t_UsageState usageState;                      // $$s004$$
  t_AdministrativeState administrativeState;    // $$s004$$
};

/* 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 */
  OffDuty,              // $$s004$$
                        /* 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
```

2.36 Security

```
#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
```

Annex 3. Subscription Information Model

The subscription management information model contains all the information required to handle end users, subscribers and subscription life cycle. It is closely related to service life cycle management as consumers are subscribed to services that are deployed and operational.

The model is divided in several fragments:

- *Subscription Information Model*: represents the entities and relationships required to manage subscriptions, subscribers and end-users in a retailer domain.
- *Subscription Business model*: representing the relationships between retailers, subscribers and end-users.
- *Service Profile model*: representing the relationship between service life-cycle management information objects (service description and service template), subscription management information objects (subscription profile and SAG service profile) and user customization information objects (user service profile). It gives an idea on the process followed to define the user service profile that will be applied to a particular service session.

The model includes the following main entities:

- *Subscriber*: It represents an entity (a person or organization) that signs a contract with the retailer for the provision of a (set of) service(s). This class contains all the information related to the subscriber that is independent from the services it is subscribed to. It includes subscriber identification, address, contact points, etc.
- *Subscription Assignment Group* (SAG): It represents a group of users, terminals or NAPs (Network Access Points) associated to, and defined by, a subscriber who share a common service profile (the SAG service profile).
- *Subscription Contract*: It represents the agreement for the provision of a service to a subscriber. It describes the terms of the agreement. This class does not describe service specific information but the conditions of the contract for the provision of the given service. This can include payment mode, bank account information, etc.
- *Service Description*: It represents the service specification produced by a standardization body, industrial forum or group of companies. It described a particular service type.
- *Service Template*: It describes the generic information and behavioral characteristics of a service instance (of a specific service type) as offered by a service provider. The description is done following the service description corresponding to the implemented service type.
- *Subscription Service Profile*: It represents the tailoring of a Service Template to the specific requirements and needs of a subscriber.
- *SAG Service Profile*: It is a customization of a subscription profile for a SAG.
- *Service Contract*: It describes the terms of the provision of a specific service to a subscriber. It collects the subscription and SAG service profiles, defining the agreed service characteristics, the generic ones for the provider and the specific ones for each SAG, respectively.

3.1 Subscription Business Model

Figure 3-1 is a part of the business model representing the relationships between the stakeholders involved in a subscription, namely subscribers, retailers and end-users.

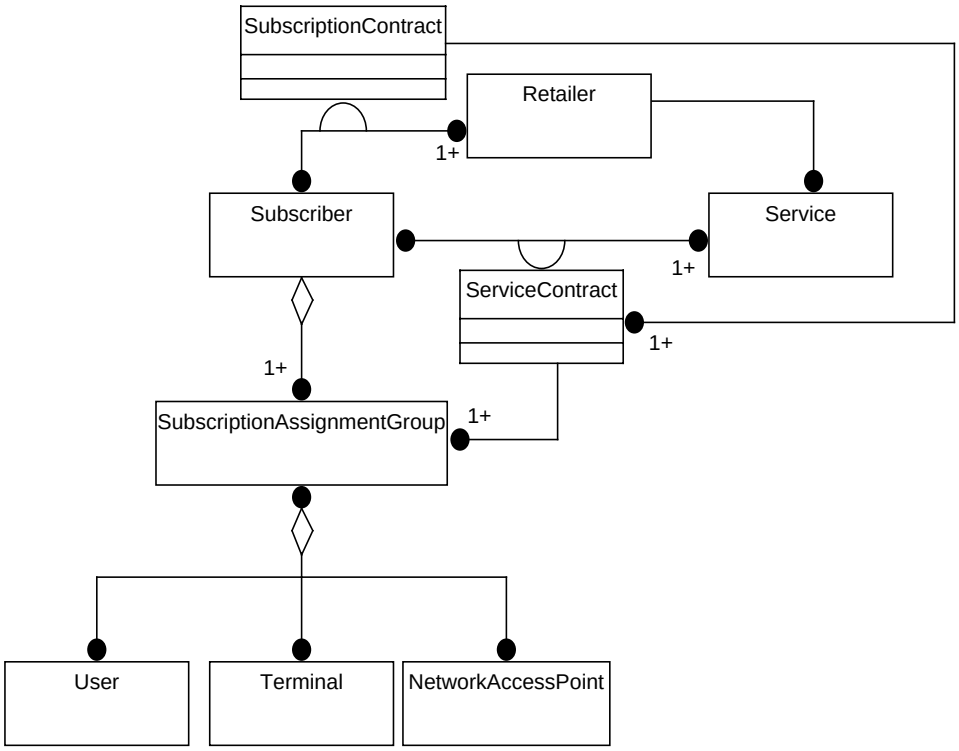


Figure 3-1. Subscription Business Model

A *Subscriber* may sign *Subscription Contracts* with a number of *Retailers*. These contracts represent the service independent part of the agreement (v.g., payment mode, subscription period, etc.). Of course, subscribers sign a subscription contract with a retailer to receive one or more *Services*. For each of the subscribed services, the retailer may allow the subscriber to set a number of conditions (service characteristics), that compose the *Service Contract*. The service contract details the terms of the provision of a specific service to a particular subscriber. A number of entities -*Users*, *Terminals* or *NAPs*- can be associated to a subscriber. If a subscriber does not want to grant the same service characteristics to all these entities, he can group them in different *Subscription Assignment Groups*, each one with a different service profile (SAGServiceProfile). By default, at least one SAG including all the users (terminals or NAPs) is considered for each subscriber.

3.2 Subscription Management Information Model

This model represents the information required to handle subscriptions, subscribers and end-users in a retailer domain.

Figure 3-2 represents mainly the relationship between a service and a subscriber, described in terms of a number of service profiles (service template, subscription profile and SAG service profile).

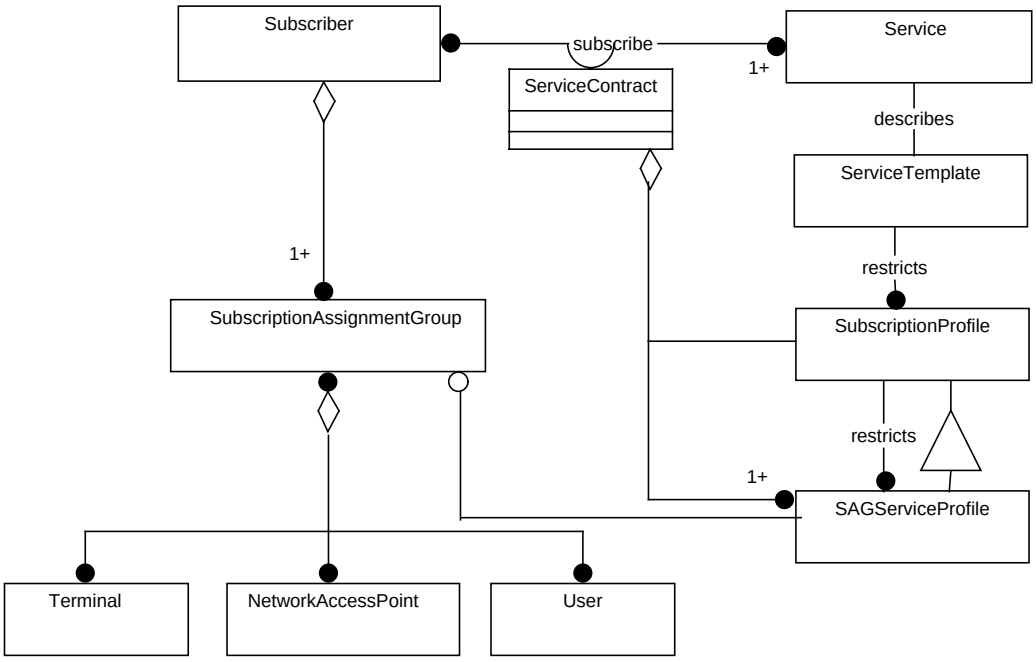


Figure 3-2. Subscription Management Information Model

A *Subscriber subscribes* to a number of *Services*; at least one to be considered as such. The agreed *Service Contract* defines the conditions of the service provision for each of the service subscriptions. A *Service Template describes* the characteristics of the *Service* provided by the retailer. The retailer may give the subscriber the option to select specific service parameters to apply to all its associated entities[1] -*Subscription Profile*- or to a group of them -*SAG Service Profile*-, reducing the alternatives (*restricts*) given in the service template. These profiles are the main part of the service contract.

A set of entities, *Users*, *Terminals* or *NAPs*, can be associated to a subscriber. The subscriber may not want to grant all of them with the same service characteristics (or privileges). For this reason, the subscriber can group them in a set of *Subscription Assignment Group*s (SAG) and assign a particular profile (*SAG Service Profile*) to each group.

3.3 Service Profile Definition

This diagram represents the relationship between the different profiles used in the service architecture. A service profile is used to describe the characteristics of the provision of a service to a customer.

The ones on top of the hierarchy describes the characteristics associated to a service type (*Service Description*) and the ones related to a specific service instance[2] (*Service Template*). Both are handled by the Service LifeCycle Management component and are part of the service type management functionality.

1. Users, terminals or network access points.
2. By service instance we mean a particular implementation of a service type by a service provider.

The *Subscription Profile* represents the tailoring of a service template to the requirements and needs of a subscriber, while the *SAG Service Profile* is a customization of the former for a specific SAG, reflecting the fact that in some subscriber organizations not all the users are granted with the same service characteristics. These profiles are defined by the subscriber at subscription time and handled by the Subscription Management components.

Finally, in some cases the end-user is granted with customization capabilities and is allowed to define its own *User Service Profile*, derived and constrained by the SAG Service Profile corresponding to the SAG the user is assigned to. This is not part of the service contract and thus not a responsibility of the subscription management components. It is handled by the User Agent as part of the user profile management functionality.

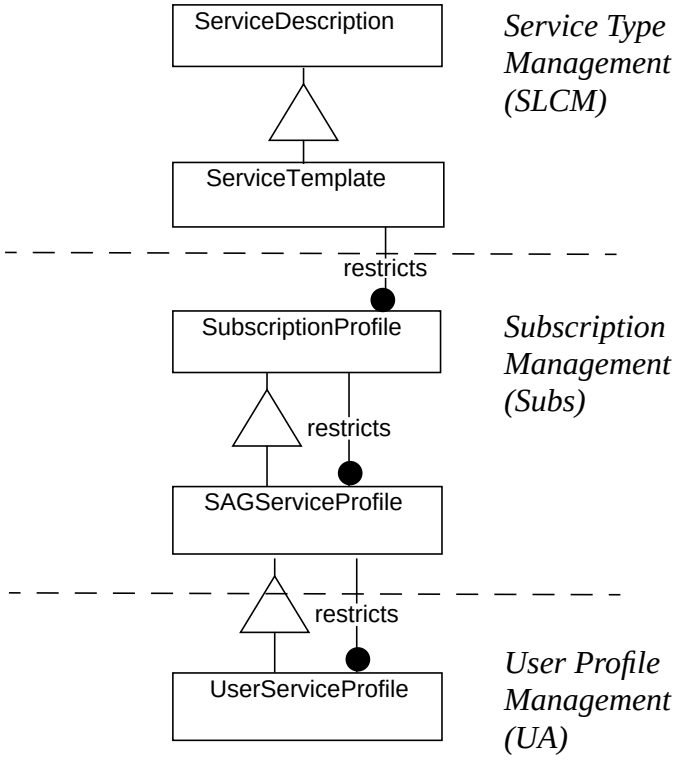


Figure 3-3. Service Profiles.

3.4 Service Dependencies

In some cases, the provision of a service requires the use of other services. This is the case of complex services that are composed of a number of basic services. In this case, the subscription to the compound service will require the definition of the service characteristics (set up of a Service Contract) for the constituent (dependent) services.

In some other cases, this may not be an operational matter but a marketing one. For instance, a retailer may decide to offer a set of services as a package and not to allow their independent subscription. In this example, the services are mutually dependent from a contractual viewpoint.

Figure 3-4 represents how a subscriber may be required to agree on a set of service contracts when

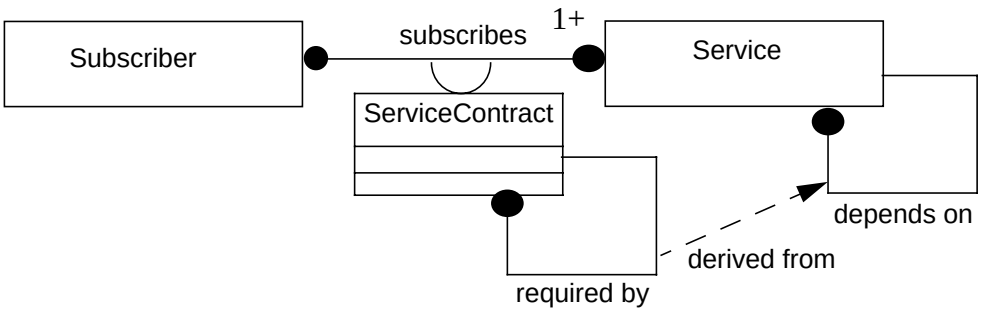


Figure 3-4. Impact of Service Dependencies on Service Contracts

he subscribes to a service that *depends on* some other services. A service contract dependency is *derived from* the service dependency.

Annex 4. Suggested Decomposition for the Subscription Management Component

In this annex, an internal decomposition for the subscription management component is described. Although, it is not prescriptive it is strongly recommended as it provides a simple and natural way of handling the two main concepts in a subscription database: subscribers and service contracts. The reason for not making it prescriptive is just the idea that it is likely that most of the retailers will wish to keep their already existing subscription databases while connecting them to new service systems like the one provided by TINA. For this reason, only the interfaces to other TINA service components and the interfaces for online subscription services are prescribed.

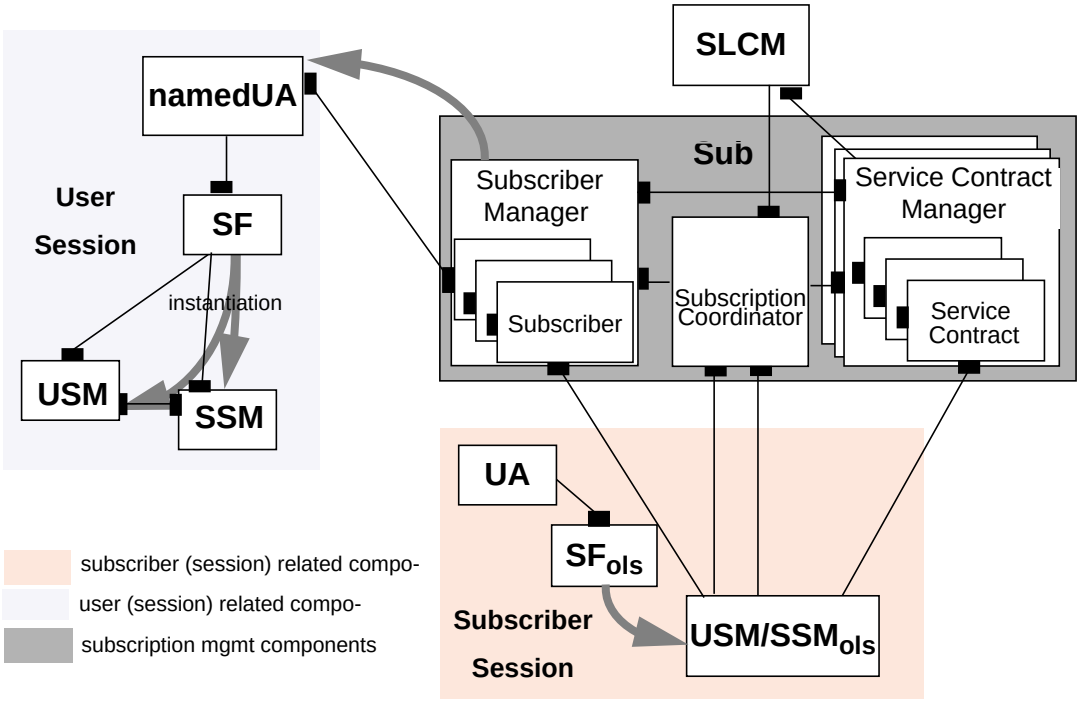


Figure 4-1. Relationship with other components in the SA.

4.1 Subscriber Manager (SubM)

It manages the subscribers' lifecycle.

The *Subscriber Manager (SubM)* offers two internal (to Sub components) interfaces:

- `i_subSCooManagement`: to create, delete and query about subscribers, and
- `i_subSCMNotify`: to notify about new service contracts or changes in the already signed ones.

SubM maintains a *Subscriber Object (SubO)* per subscriber. This SubO offers two external interfaces:

- `i_SubscriberInfoQuery`: to query subscriber information, and
- `i_SubscriberInfoMgmt`: to update subscriber information.

4.1.1 i_SubscriberInfoQuery

:This interface is offered to the access components (NamedUA, PeerA) to allow them to retrieve the subscription information associated to a particular user, the one that the access components is representing.

The most important operations in this interface are:

- **`listServices`**(): returns the list of services the user is subscribed to, indicating which ones are available with the current terminal configuration.
- **`getServiceProfiles`**(): returns the profile assigned to the user (SAG service profile) for a specific service or a list of services.
- **`checkServiceProfile`**(): It checks whether the User Service Profile (customized by the end user) is compatible with the subscribed profile (corresponding SAG Service Profile).

4.1.2 i_SubscriberInfoMgmt

This interface allows to add, modify or delete subscriber related information like associated entities (users, terminals or NAPs), subscription assignment groups and associations of service profiles to SAGs. From this interface, the client can access only to the information of a particular subscriber.

- **`createEntities`**(): it creates the entities specified as a parameter returning an identifier for each of them.
- **`deleteEntities`**(): it deletes the entities specified as a parameter. It removes any existing assignment to SAGs these entities could have.
- **`createSAGs`**(): it creates a (number of) SAG(s). A list of entities for every SAG can be specified. A SAG identifier is returned to ease further management.
- **`assignEntitiesToSAG`**(): It assigns a list of entities to a SAG.
- **`removeEntitiesFromSAG`**(): It removes a list of entities from a SAG.
- **`listEntities`**(): It returns the list of entities associated to the subscriber. If a (list of) SAG identifier(s) is specified, it returns only the users assigned to that(those) SAG(s).
- **`listSAGs`**(): It returns the list of SAGs (ids) for that subscriber.
- **`getSubscriberInfo`**(): It returns the information about the subscriber.
- **`setSubscriberInfo`**(): It modifies the information about the subscriber.
- **`listSubscribedServices`**(): It returns the list of services subscribed by the subscriber. If a user is specified, it returns the list of services granted to that specific user by the subscriber.

4.1.3 i_SubscriberLCMgmt

This interface is offered to the Subscription Coordinator (SCoo) to allow it to create, delete and list subscribers.

The most important operations in this interface are:

- **`createSubscriber`**(): creates a subscriber object and returns its interfaces.
- **`deleteSubscriber`**(): deletes a subscriber object.

- **`listSubscribers`**(): returns the list of subscribers. If a service identifier is specified, it returns the list of subscribers subscribed to that service.
- **`listUsers`**(): returns the list of users. If a service identifier is specified, it returns the list of users subscribed to that service.

4.1.4 i_ServiceContractInfoUpdate

This interface is offered to the Service Contract Manager (SCM) to allow it to notify a subscriber object about services contracted by its represented subscriber.

The most important operations in this interface are:

- **`notify`**(): notifies the SubO about a new, modified or cancelled service contract signed by the represented subscriber. It passes also a reference to the Service Contract Object in order to allow it to get further information.

4.2 Subscription Coordinator (SCoo)

This is the main control point of subscription management. It coordinates subscriber management and service contract management. When it receives a request to subscribe a new customer, it creates a Subscriber Object for this new customer (via the `i_subSCooManagement` interface in the SubM) and a set of Service Contracts (via the `i_subSCooManagement` interface in the SCM), one for each requested service.

It provides interfaces to clients to get the appropriate interface references (`i_InitialAccess`) to perform its corresponding subscription management operations.

An interface (`i_Subscribe`) is offered to allow clients to apply for or cancel service contracts or subscriptions to the retailer domain.

This component is also the point of contact with the SLCM. It is offering an interface (`i_ServiceNotify`) to this component to receive notifications about new available services or modification/withdrawal of existing ones.

All the interfaces offered by this component are external (client are not part of the set of subscription management components).

4.2.1 i_InitialAccess

It allows a client to request an interface to access to the subscription management functionality. In case the client is a UA, it returns a `i_SubscriberInfoQuery` interface reference and, in case it is a SSM_{ols}, a `i_Subscribe` interface reference. A terminate operation is provided to release the interfaces once they are not needed.

- **`init`**(): returns the list of interface references corresponding to the client that makes the request.
- **`terminate`**(): release the resources that could be allocated in the `init` operation.

4.2.2 i_Subscribe

It allows to create a subscription contract for a subscriber. These are the main operations in this interface:

- **`getReferences`**(): It returns the references to interfaces to modify subscriber info or service contract info.
- **`listServices`**(): It returns the list of services provided by the retailer.

- **`subscribe`**(): It allows to create a subscription contract with the retailer. As input parameters it has the subscriber information and a list of services the subscriber is willing to subscribe to. It returns a subscriber identifier, a reference to the subscriber information management interface (i_SubscriberInfoMgmtMgmt) and a list of interface references for contracting each of the specified services (i_subSMContractService).
- **`unsubscribe`**(): It allows a subscriber to delete a (list of) service contract(s) or the whole relationship with the retailer.
- **`contractService`**(): It subscribes a subscriber to a service and returns an interface reference where he can define the service contract (i_subSMContractService).
- **`listSubscribers`**(): It returns the list of subscribers. If a service Id is specified, it returns the list of subscribers for that service. Only accessible for a retailer operator.
- **`listServiceContracts`**(): It returns the list of service contracts. If a service Id is specified, it returns the list of service contracts for that service. If a subscriber is specified, it acts as the `listSubscribedServices` in the `i_SubscriberInfoMgmt` interface for that particular subscriber. Only accessible for a retailer operator.
- **`listUsers`**(): It returns the list of users for a specified service. Only accessible for a retailer operator.

Editor's note: Maybe, we should move the operations only accesible to retailer operators to a different interface. Any feedback about this is appreciated.

4.2.3 i_ServiceNotify

This interface allows SLCM to notify Sub about new services deployed and available for subscription and use, or about modification or withdrawal of existing ones.

- **`notify`**(): it notifies Sub about the deployment, upgrade or withdrawal of services in the network, so that Sub can have updated information about subscribable and available services.

4.3 Service Contract Manager (SCM)

The SCM controls the service contract lifecycle for a specific service instance. There will be as many SCMs as services is offering the retailer to the subscribers. A new SCM will be deployed every time a new service is deployed and made ready for usage and subscription. It offers an internal interface to the Subscription Coordinator (`i_ServiceContractLCMgmt`) to create, query and delete service contracts.

It maintains a *Service Contract Object (SCO)* for each service contract. This object offers two interfaces:

- `i_ServiceContractInfoMgmt`: it allows the modification and query of the service contract information. This is an external interface that allows the client to define the service contract.
- `i_ServiceContractInfoQuery`: it allows the retrieval of service contract information. It is an internal interface used by the Subscriber Object to get information about the subscribed services.

4.3.1 i_ServiceContractInfoMgmt

`i_ServiceContractInfoMgmt` provides functions to define and modify a service contract.

The main operations are:

- **`getServiceTemplate`**`()`: It returns the template for service profile definition.

- **`defineServiceContract`** (): It allows to define the service contract. This contract includes, amongst other contractual information, the set of service profiles composing the service contract, namely the subscription profile (applicable to all users) and the set of SAG service profiles (each one applicable to a SAG and consistent with the subscription profile). It returns a list of SAG profile identifiers to ease their future reference. It is used to define and redefine (modify) service contracts.
- **`defineServiceProfiles`** (): It allows to define a set of service profiles for the service contract, namely the subscription profile and the set of SAG service profiles. It returns a list of SAG profile identifiers to ease their future reference. It is used to define and redefine (modify) service profiles.
- **`deleteServiceProfiles`** (): It deletes a service profile removing the SAG that could be associated to it.
- **`getServiceContractInfo`** (): It returns the information related to the service contract. If a list of SAGs is specified, the set of associated SAG service profiles defined for those SAGs in the contract is returned.

4.3.2 i_ServiceContractInfoQuery

This interface provides functions to query information about a service contract. It is used by a Subscriber Object (SubO) to retrieve the subscriber's service profiles.

The main operation is:

- **`getServiceProfiles`**`()`: It returns the service profiles defined in the service contract. If a SAG identifier is specified, only the SAG Service Profile for that SAG is returned.

4.3.3 i_ServiceContractLCMgmt

This interface provides functions for creating, querying and deleting service contracts. It is used by the Subscription Coordinator (SCoo).

The main operation is:

- **`listServiceContracts`**`()`: It returns the list of service profiles signed. If a service identifier is specified, only the service contracts for that service are returned. If a subscriber is specified, using its account number, the list of service contracts for that subscriber is returned.
- **`createServiceContracts`**`()`: It creates a set of service contracts and returns the list of references to `i_SubInfoMangement` interfaces (one per service contract) required to manage them.
- **`deleteServiceContracts`**`()`: It deletes the set of service contracts specified as parameters.