Telecom Service Access & Subscription Specification, V1.0

This OMG document replaces the draft adopted specification and the submission document telecom/2000-05-03. It is an OMG Final Adopted Specification, which has been approved by the OMG board and technical plenaries, and is currently in the finalization phase. Comments on the content of this document are welcomed, and should be directed to *issues@omg.org* by November 30, 2000.

You may view the pending issues for this specification from the OMG revision issues web page *http://www.omg.org/issues/*; however, at the time of this writing there were no pending issues.

The FTF Recommendation and Report for this specification will be published on January 15, 2001. If you are reading this after that date, please download the available specification from the OMG formal specifications web page.

OMG Adopted Specification

Telecommunications Service Access and Subscription (TSAS) Specification

Final Adopted Specification
October 2000

PATENT

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

NOTICE

The information contained in this document is subject to change without notice. The material in this document details an Object Management Group specification in accordance with the license and notices set forth on this page. This document does not represent a commitment to implement any portion of this specification in any company's products.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR PARTICULAR PURPOSE OR USE. In no event shall The Object Management Group or any of the companies listed above be liable for errors contained herein or for indirect, incidental, special, consequential, reliance or cover damages, including loss of profits, revenue, data or use, incurred by any user or any third party. The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Right in Technical Data and Computer Software Clause at DFARS 252.227.7013 OMG® and Object Management are registered trademarks of the Object Management Group, Inc. Object Request Broker, OMG IDL, ORB, CORBA, CORBAfacilities, CORBAservices, and COSS are trademarks of the Object Management Group, Inc.

X/Open is a trademark of X/Open Company Ltd.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the issue reporting form at *http://www.omg.org/library/issuerpt.htm*.

Contents

Contents

Contents

Preface

About the Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 800 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

Associated OMG Documents

The CORBA documentation set includes the following:

- *Object Management Architecture Guide* defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.
- *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.
- *CORBA Languages*, a collection of language mapping specifications. See the individual language emapping specifications.
- *CORBAservices: Common Object Services Specification* contains specifications for OMG's Object Services.
- *CORBAfacilities: Common Facilities Specification* includes OMG's Common Facility specifications.
- *CORBA Manufacturing*: Contains specifications that relate to the manufacturing industry. This group of specifications defines standardized object-oriented interfaces between related services and functions.
- *CORBA Med*: Comprised of specifications that relate to the healthcare industry and represents vendors, healthcare providers, payers, and end users.
- *CORBA Finance*: Targets a vitally important vertical market: financial services and accounting. These important application areas are present in virtually all organizations: including all forms of monetary transactions, payroll, billing, and so forth.
- *CORBA Telecoms*: Comprised of specifications that relate to the OMG-compliant interfaces for telecommunication systems.

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

OMG formal documents are available from our web site in PostScript and PDF format. To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404

Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Acknowledgments

The following companies submitted and/or supported parts of this specification:

- Alcatel
- AT&T
- Britich Telecommunications plc.
- Cicso Systems
- Deutsche Telekom AG
- GMD Fokus
- Hitachi
- Humboldt University
- IBM Telecommunications Industry
- KPN Royal Dutch Telecom
- Lucent Technologies
- Nippon Telegraph and Telephone (NTT) Corporation
- Nortel Networks
- Siemens AG
- Sprint
- Sun Microsystems

Description 1

This chapter introduces the key concepts used in this specification.

Contents

This chapter contains the following sections.

1.1 Motivation

Network operators have traditionally followed a network-centric approach to delivering scalable, reliable and economic services to consumers and enterprises. The basic functions that are required to support services such as 800 numbers, call waiting and personal numbering have been under the exclusive control of the network operators. Enterprises and service providers wishing to offer value-added solutions, such as call centers, have had to rely on an edge-of-network approach and have been denied access to useful information and capabilities within the network.

The disadvantages of this separation are significant in today’s marketplace. Network operators employing a network-centric approach are unlikely to have the resources and flexibility necessary to respond to the specialized requirements of different customer

markets. Similarly, solution providers adopting an edge of network approach, while they may have the flexibility required for customizing services, are unable to gain the efficiency of using in-network functions and information. The architecture of the Telecommunication Service Access and Subscription (TSAS) specification combines the benefits of the network centric approach of economies of scale with the flexibility of the edge of network approach.

The set of interfaces contained within this specification provide the domain facilities through which network operators can offer 3rd party enterprises secure access to the capabilities of the network. Capabilities such as call control and user location can be offered (through their own interfaces) or by 3rd party value-added services and solutions.

Of course this approach is not only applicable to providing access to embedded network capabilities. It can also be used for a wide range of commercial models supporting customer-to-business or business-to-business relationships for eCommerce and the Application Service Provider market in general. Provision of functions for billing and payment can be easily integrated.

It is not within the scope of this specification to restrict the breadth of [component] services that could be offered by TSAS. This specification is technically aligned with that of the Parlay Group [Ref http://www.parlay.org]. Consequently the service interfaces specified in the Parlay API [Parlay API specification 2.0] can be offered using this specification.

1.2 Roles and Domains

Three different domains are defined for TSAS as shown in Figure 1-1: Consumer Domain, Retailer Domain, and Service Provider Domain.

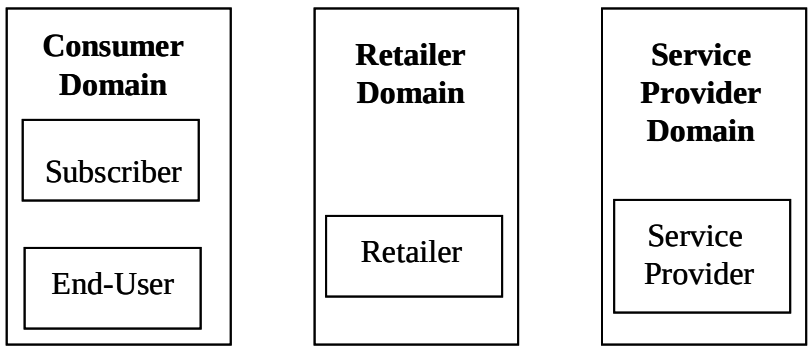

Figure 1-1 TSAS Domains

The domains are strongly correlated to *roles*, which will be explained in the following text.

In the *Consumer Domain* two kinds of roles are defined, the *end-user* role and the *subscriber* role. Typically end-users can be private households or any kind of company. The *end-user* is the one that makes use of the service while the *subscriber* holds the contract with the retailer and subscribes to services for its users. This can be depicted with a very common example: A company - the subscriber - has a subscription contract with a telephony provider. In the contract the rights of the different employees are defined - the employees are the end-users. In the case of a private household the subscriber and end-user role are identical.

Within the Retailer Domain the retailer role is defined. The *retailer* provides an integrated view of services to the end-user or subscriber. A major point of value added services offered by retailers is the unified management of services, in particular in terms of subscription facilities. Retailers thus act as middlemen for service providers and present a single point of contact to end-users and subscribers. This is an analogy to the notion of one-stop-shopping in a supermarket. Retailers have to ensure the ease and quality of service access.

Within TSAS the retailer is giving end-users a single point of contact for all their service needs. Additionally, the retailer enables end-users to customize and personalize services that they use by providing facilities to configure and select services incorporating personal preferences.

A prerequisite for service provisioning is a contractual relationship between service providers and the retailer. For service access a contractual relationship must exist between the subscriber and the retailer. No direct contractual relationship is required between end-user and service provider since the retailer mediates between both.

In general, the retailer:

- manages contracts for end-users and service providers,
- locates matches between user requirements and service provider subscription offers, and finally,
- enables the interaction between end-user and service provider.

In the Service Provider Domain the *service provider* role is defined. It offers its services to the end-user (or subscriber) through a retailer, or in other words, it supports the retailer with services. In addition, the retailer allows service providers to reach a larger number of potential end-users. The services that are provided by the service provider can be service logic or content, or both. The service provider can also be compared to a wholesaler.

The TSAS specification is a domain facility enabling end-users to access telecommunication services according to their own wishes. In addition, the specification describes how services can be retailed on behalf of service providers, which in turn offer their services to the retailer.

1.3 User Provider Relationship

TSAS offers mechanisms to establish and release authenticated connections between different domains; therefore, each domain provides interfaces to do so. TSAS uses the terms *user* and *provider* instead of the *client* and *server* terminology, which would be misleading in a number of situations. The user is the role directed to use the interface and provider is the role providing the interface, which is shown in Figure 1-2.

The active role is always the *user role* that initiates the access whereas the passive role is the *provider*, responding to a request.

For a single interaction between two domains request - response user and provider are situated in different *domains*. The domain boundaries are usually based on natural affinities between objects, such as network topology, business stakeholder, or geographical area. In a single scenario more than one user - provider relation may exist (for example, it is possible to have a chain in which a single party acts as a provider in one direction and as user in the other direction).

This may be illustrated by the following example. There is a chain of end-user, retailer, and service provider. The retailer offers services to the end-user, which are realized by the service provider. In the relationship between end-user and retailer the end-user is a user, the retailer a provider. In the relationship between retailer and service provider the retailer is a user and the service provider is a provider.

Note that the definitions in this section imply that the terms *user* and *provider*, *end-user*, and *service provider* have different meanings.

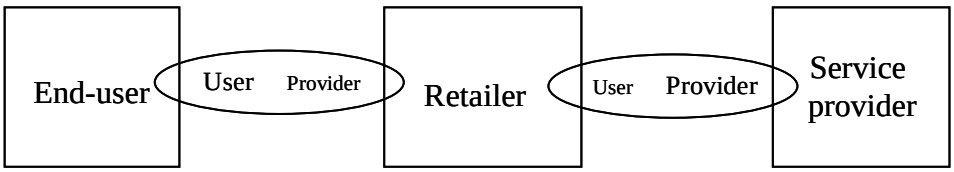

Figure 1-2 use of generic user and provider roles

1.4 Sessions

The usage of services that are implemented taking into account the TSAS framework can be structured in different *sessions*. These are used for grouping specific activities between user and provider. The TSAS specification distinguishes between two different sessions:

- An a*ccess session* is used to establish an authenticated binding between two domains, which in TSAS is between the consumer domain and the retailer domain or between the retailer domain and the service provider domain. It maintains the state about a user's attachment to a provider and about its involvement in services. An access session hence represents the context through which the end-user can access services. The general access session concept also supports all aspects of mobility, that means ubiquitous access by an end-user to the services, irrespective of the terminal being used and the point of attachment to the network.

- A *service session* represents a single activation of a service. It can relate multiple end-users of the service so that they can interact with each other. Moreover, end-users can share resources such as documents or white boards. An end-user may be involved in many services at the same time although it has accessed the retailer only once. The state of a service session is always kept by the service provider (not by the retailer).

Generally, a service cannot be used without having an active access session. Closing the connection between end-user and retailer, or retailer and service provider respectively, will end an access session and also terminate all currently used services.

1.5 Segments

The operations offered by TSAS are grouped in interfaces. The interfaces in turn build *segments*: named sets of interfaces (including so-called callback interfaces) that can be exchanged in one synchronous operation invocation. A segment may consist of two sets of interfaces: one dedicated set for each domain as shown in Figure 1-3.

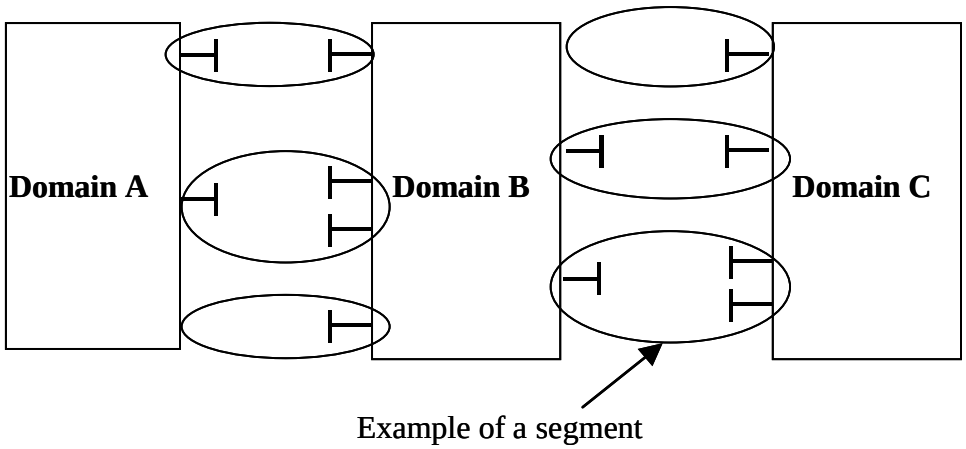

Figure 1-3 Domains, Segments and Interfaces

One TSAS segment is mandatory: the core segment that handles the initial access phase between different domains. This covers the possibility to perform an authentication protocol, and access to services an end-user may wish to use. In addition, it offers the possibility to gain access to other segments supported by the provider.

The other segments can be selected at runtime after an (optional) negotiation phase. Currently these additional segments are defined and described in this chapter and chapter 2: The Invitation Segment, the Context Segment, the Access Control Segment, the Service Discovery Segment, the Session Control Segment, the End-user Customization Segment, the Service Provider Administration Segment, the End-user Administration Segment, and the End-user customization segment They all offer additional service independent functionality.

The usage of optional segments may be tailored for a certain purpose. Segments are self-contained, there exist no dependencies between segments. This eases use of some segments in a certain context, and allows adding additional segments in the future.

The optional segments (also called Service Access Segments and Subscription Segments) are available during an access session only, as described in Section 1.2, “Roles and Domains,” on page 1-2. Its operations allow the end-user or subscriber, retailer, and service provider to interact during an access session in the respective roles of user or provider across domains.

Segments can be requested or supported by the involved domains, depending on the required functionality. Each of these segments can be selected independently of the others. Once selected, however, the segment implementation must use the specifications of this document.

1.6 Security

TSAS uses (mutually) authentication mechanisms between two domains, between the end-user of the consumer domain and the retailer domain, and between the retailer domain and the service provider domain respectively. For authentication either CORBA security can be used or the authentication interface defined in Section 2.2.2, “Authentication Interface,” on page 2-6. Once authenticated, the other optional segments can be used without further authentication for each segment. As a result of the authentication, references of interfaces are available between domains and remain available as long as the relationship resulting from authentication is valid.

Core Segment 2

Contents

This chapter contains the following sections.

2.1 Overview

The core segment is mandatory and defines the interfaces which are used in the initial phase between different domains. This covers the first point of contact to access a provider, the possibility for user and provider to perform an authentication protocol, the access to services they wish to use, and access to other segments supported by the provider.

In TSAS a user contacts a provider to access services offered by the provider. To access these services, the user is required to invoke authentication procedures with the provider before it is able to access services. The use of the terms *user* and *provider* is made according to their definition in the previous chapter.

TSAS defines:

- The first point of contact for a user to access a provider.
- The authentication operations for the user and provider to perform an authentication procedure.
- The user access to services they wish to use.

- The user access to other segments supported by the provider.

The process by which the user accesses the provider has been separated into 3 phases:

1. Initial Contact
2. Authentication
3. Access to the provider's services and segments

Within the core, segment interfaces are defined and within these interfaces operations are defined to enable the user to progress through each of these phases. An overview of these interfaces and operations is given in Figure 2-1.

.

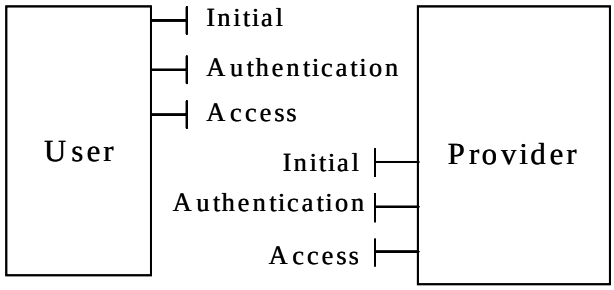

Figure 2-1 Core Interfaces

Initial - This interface allows a user to initiate an authentication procedure and to request access to the provider domain. This initiates an access session; the concept of access session is explained in Section 1.4, "Sessions," on page 1-4. The operations provided are:

- **initiate_authentication()** - allows the user to initiate an authentication procedure.
- **request_access()** - allows the user to request the provider to initiate an access session. If successful the user gains access to an interface for accessing services and other segments offered by the provider.

Authentication - This interface allows a user to proceed through an authentication procedure. It provides the following operations:

- **select_auth_method()** - for selecting the authentication procedure.
- **authenticate()** - to perform the authentication. (It can be invoked several times to complete the authentication procedure).
- **abort_authentication()** - to abort the authentication procedure.

Access - This interface allows an authenticated user to access services and other segments offered by the provider. The interface provides the following operations:

Table 2-1

Operation	Description
list_available_services()	Lists all services that are available at the retailer. The services are scoped using property lists. The operation returns sufficient information for the user to select a service, then start a service
select_service()	To select the service to be provided, and provide configuration information.
start_session()	To start a service session.
sign_service_agreement()	Used to start a service session when non-repudiation of the request to start the session is required.
end_access()	To end the access session.
end_session()	To end service sessions.
get_segment()	To set-up a segment.
list_segments()	To list the segments that are available from the provider.
release_segments()	To release segments.

2.2 Initial Contact and Authentication

Before a user can retrieve information about services offered by a provider, or use these services, they need to contact the provider, and perform an authentication procedure. Figure 2-2 on page 2-4 shows the sequence of operations on the **Initial** and **Authentication** interfaces, for the user to contact the provider, and authenticate. The user then gains access to the **Access** interface to retrieve information on services, use services, and use other interfaces offered by the provider.

- (Before diagram) - User gains a reference to the **Initial** interface of the provider. This may be gained through a URL, an Application Support Broker, a stringified object reference, etc.
- User may invoke **initiate_authentication** on the **Initial** interface. This 'starts' the authentication of the user and provider. The operation allows the user and provider to swap references to the **Authentication** interface. There is the possibility to choose between different authentication types. Here the TSAS authentication type is used also shown in Figure 2-2 the mutual authentication in brackets.
- User invokes **select_auth_method** on the provider's **Authentication** interface. The user identifies to the provider the authentication methods that it can use. Upon return, the provider selects the mechanism that it wishes the user to use.

- User invokes authenticate on the **Authentication** interface, in accordance with the authentication protocol selected. The authenticate operation contains an opaque parameter for the user to fill with data appropriate for the selected authentication protocol. This is the challenge parameter for the provider. The provider is able to 'decode' this parameter, and produce an appropriate response, based upon the challenge data, according to the authentication protocol. This response data is returned to the user in the response parameter. This operation identifies the user unequivocally to the provider.
- The response data is decoded by the user. Depending upon the response data and the selected authentication protocol, the user may need to produce some additional challenge data to the provider. If this is necessary, then the user makes repeated calls using authenticate Authentication. This process continues until the response data indicates that the authentication protocol is complete, and the user and provider are satisfied that they have authenticated each other. If either side is not satisfied with the authentication, they may call the **abortAuthentication** operation to abort the authentication protocol.
- Once user and provider are authenticated, the user invokes the **requestAccess** operation on the **Initial** interface. This operation allows the user to select the type of access that they require. If they select ACCESS, then a reference to the **Access** interface is returned.

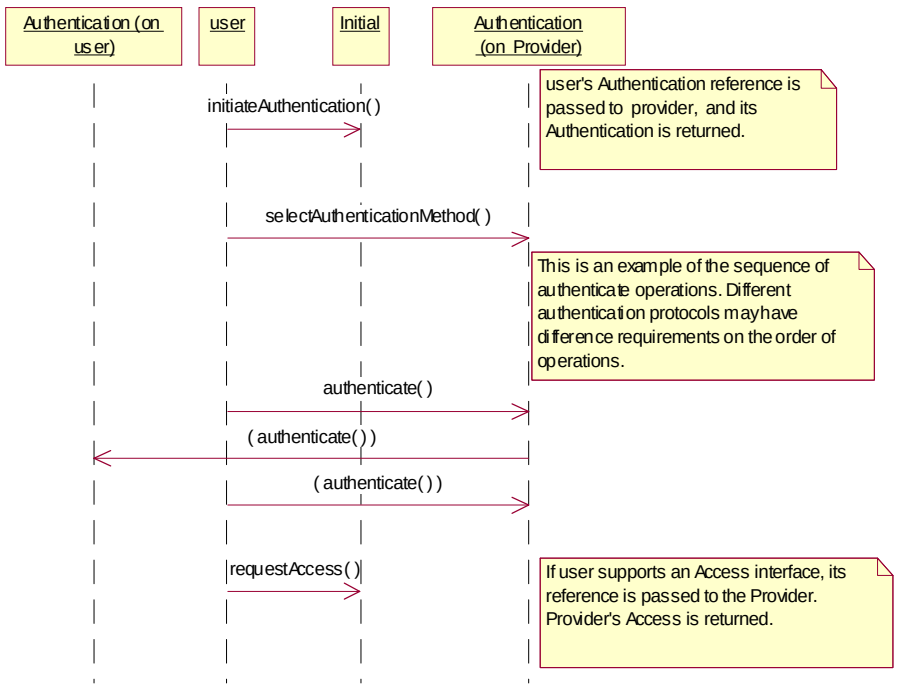

Figure 2-2 Sequence diagram for initial access and authentication

2.2.1 Initial interface

The user gains a reference to the **Initial** interface for the provider that it wishes to access. This may be gained through a URL, an Application Support Broker, a stringified object reference, etc. At this stage, the user has no guarantee that this is a reference to a valid provider.

The user uses this interface to identify himself to the provider and to initiate the authentication process. The **Initial** interface supports the **initiate_authentication** operation to allow the authentication process to take place. It also supports the **request_access** operation to gain access to the provider after the authentication has completed successfully.

2.2.1.1 initiate_authentication()

```
void initiate_authentication (
        in AuthDomain  user_domain,
        in AuthType  auth_type,
       out AuthDomain provider_domain)
    raises (
        DomainError,
        AuthError );
```

The user uses this method to initiate the authentication process. **user_domain** is an identifier for the user's domain that starts the authentication process. It is a structure of the type **AuthDomain** that contains a **DomainId** and an interface reference. The **DomainId** is used to identify the user to the provider (see **authenticate()** on **Authentication** interface). If the **DomainId** is unknown to the provider, an exception DomainError is raised by the provider. The interface reference is a reference to an **Authentication** interface at the user domain that can be invoked by the provider to perform the authentication procedure.

auth_type identifies the type of authentication mechanism requested by the user. It provides users and providers with the opportunity to use an alternative to the **TSAS Authentication** interface (for example, CORBA Security). This authentication process may be specific to the TSAS provider. The **TSAS Authentication** provided by the authentication interface is the default authentication method.

If the CORBA Security Service is supported by both the user and the provider, then it may be used to mutually authenticate the user and the provider. The operation of the CORBA security service is out of the scope of TSAS. If it is used to provide authentication of the parties, then the CORBA_SECURITY value is used for the **auth_type** attribute, and no further authentication is required.

However, if the CORBA Security Service is not supported by both parties, and if further authentication is required, then the **TSAS Authentication** interface can be used. It is obtained by filling the **auth_type** attribute with the value **TSAS_AUTHENTICATION**.

The operation delivers **provider_domain**, an identifier of the provider domain. Similar to the **user_domain**, it is a structure of the type **AuthDomain** that contains a **DomainId** and an interface reference. The **domainId** is used to identify the provider to the user. The interface reference is a reference to an **Authentication** interface at the provider domain.

2.2.1.2 *request_access()*

```
request_access (
        in AccessType access_type,
        in Object user_access,
        out Object provider_access)
    raises ( AccessError );
```

The user uses this method to gain access to the provider by means of an access session. This operation must be invoked only after user and provider are authenticated. If this method is called before the user and the provider have successfully completed the authentication process, then the request fails and an exception AccessError is raised.

access_type identifies the type of access interface requested by the user. Providers can define their own access interfaces to satisfy user requirements for different types of access. If the user requests ACCESS, then the **TSAS Access** interface is returned. **TSAS Access** is the default access method. Depending on the requested AccessType, the access interface with the corresponding type is returned (see below).

user_access provides the reference for the provider to call the access interface of the user. If the interface reference does not correspond to the type expected, due to the value of **access_type**, an exception AccessError is raised by the provider.

The returned object provides the reference for the user to call the access interface of the provider.

2.2.2 *Authentication Interface*

Once the user has made initial contact with the provider, authentication of the user and provider may be required. The user may be required to authenticate with the provider before it will be able to use any of the other interfaces supported by the provider. Invocations on other interfaces may fail until authentication has been successfully completed.

TSAS supports several authentication methods. TSAS also defines its own generic authentication mechanism. If the user wants to use the TSAS generic authentication, then it uses the **initiate_authentication** operation on the provider's **Initial** interface as described above, with **auth_type** parameter set to **TSAS_AUTHENTICATION**. The reference returned is the **TSAS Authentication** interface. This interface can be used to support an authentication procedure.

1. The user invokes the **select_auth_method** operation on the provider's **Authentication** interface. This includes the authentication capabilities of the user (that is, the authentication procedures known by the user application). The provider

then chooses an authentication procedure based on the authentication capabilities of the user and the provider. If the user is capable of handling more than one authentication procedure, then the provider chooses one option, the **selected_cap**. In some instances, the authentication capability of the user may not fulfill the demands of the provider, in which case, the authentication will fail.

2. The user and provider interact to authenticate each other. Depending on the authentication capability selected, this procedure may consist of a number of interactions (for example, a challenge/response protocol). This authentication procedure is performed using the authenticate operation on the **TSAS Authentication** interface. Depending on the authentication capability selected, the procedure may require invocations on the **Authentication** interface supported by the provider; or on the **Authentication** interface supported by the user; or on both interfaces.

After the authentication procedure has been completed, the user can invoke the **request_access** operation on the **Initial** interface to gain access to the provider's services and other TSAS segments supported by the provider.

2.2.2.1 *select_auth_method()*

```
void select_auth_method (
        in AuthCapabilityList auth_caps,
        out AuthCapability selected_cap)
    raises (AuthError );
```

The user invokes the **selectAuthMethod** on the provider's **Authentication** interface to initiate the TSAS generic authentication process. This provides the authentication capabilities of the user to the provider. The provider then chooses an authentication method based on the authentication capabilities of user and provider. The operation returns the selected method (**selected_cap**). In some instances, the authentication capability of the user may not fulfil the demands of the provider, in which case the authentication will fail (the operation raises the exception Authentication Error).

- **auth_caps** is the means by which the authentication mechanisms supported by the user are conveyed to the provider. Examples for authentication capabilities may be (for example, bio ID techniques, chip cards, or username/password combinations).
- **selected_cap** is returned by the provider to indicate the mechanism preferred by the provider for the authentication process among the ones supported by the user that were specified in **authCaps**. If the value of the **selectedCap** returned by the provider is not understood by the user, it should be considered as an unrecoverable error ('panic') and the user should abort its application.

2.2.2.2 *authenticate()*

```
void authenticate (
        in AuthCapability selected_cap,
        in string challenge,
        out string response)
```

raises (AuthError);

The user and provider use this operation to authenticate each other. It returns a response string. This operation is used according to the authentication procedure, selected by the **selected_cap** parameter (returned by **select_auth_method()**). This procedure may consist of a number of messages (for example, a challenge/ response procedure). The values of the challenge and response parameters are defined by the authentication procedure. The challenge is used to identify a user uniquely. It may contain a userId or a certificate, which can identify the user by a distinguished name conforming to X.509 v3.

An AuthError exception is raised if the **selected_cap** does not correspond to the **selected_cap** returned by **select_auth_method()**. An AuthError exception is also raised if the challenge data does not correspond to the procedure selected (that is, the challenge data cannot be decrypted according to that method).

The response attribute provides the response of the provider to the challenge data of the user in the current sequence. The response will be based on the challenge data, according to the procedure selected by **select_auth_method ()**.

2.2.2.3 abort_authentication()

void abort_authentication ( )
raises (AuthError);

The user uses this method to abort the authentication process. This method is invoked if the user no longer wishes to continue with the authentication process (for example, if the provider responds incorrectly to a challenge). If this method has been invoked, calls to the **request_access** operation on the **Initial** interface will raise the AccessError exception until the user has been properly authenticated. It contains no attributes.

2.3 Access

Once a user has been authenticated with a provider an access session is established. The user now can gain access to the services and other segments offered by the provider.

The user invokes the **request_access** operation on the **Initial** interface with the required accessType. If it requests ACCESS, then a reference to the **Access** interface is returned. (TSAS Providers can define their own access interfaces to satisfy user requirements for different types of access). The user also provides the provider with a reference to its ‘callback’ interface to allow the TSAS provider to initiate interactions during the access session. If the user has requested ACCESS, then it must provide a reference to an interface that will be used for the authentication procedure.

The **Access** interface allows the user to access services offered by the provider and to gain references to other segments. Segments are defined by TSAS in chapter 3. The sequence for accessing the segments is given in Figure 2-3. Segments are accessed by using the **list_segments()**, **get_segment()**, and **release_segments()** operations.

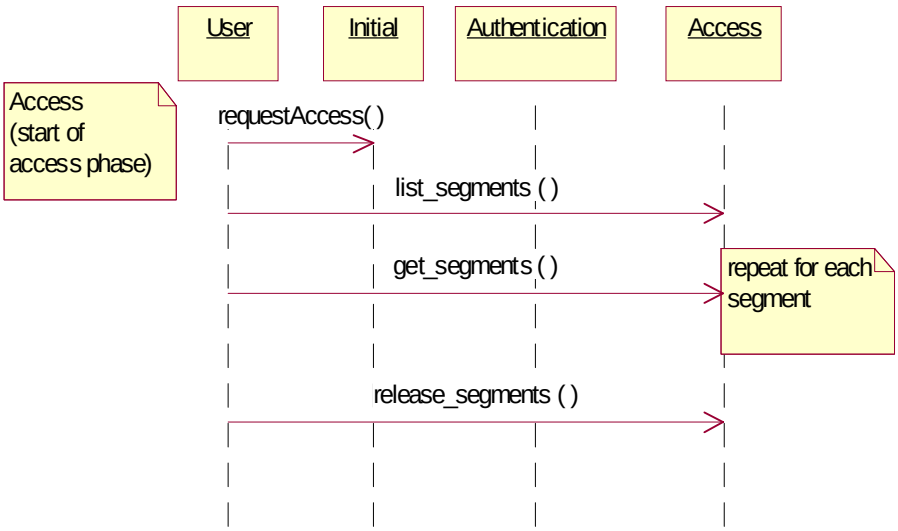

Figure 2-3 Sequence diagram for access segments

list_segments() may be used for getting informed which segments are currently available for a user. With **get_segment ()** a single segment will be returned. This operation needs to be called separately for every segment which shall be used. When segments are not needed anymore, they can be released with **release_segments()**.

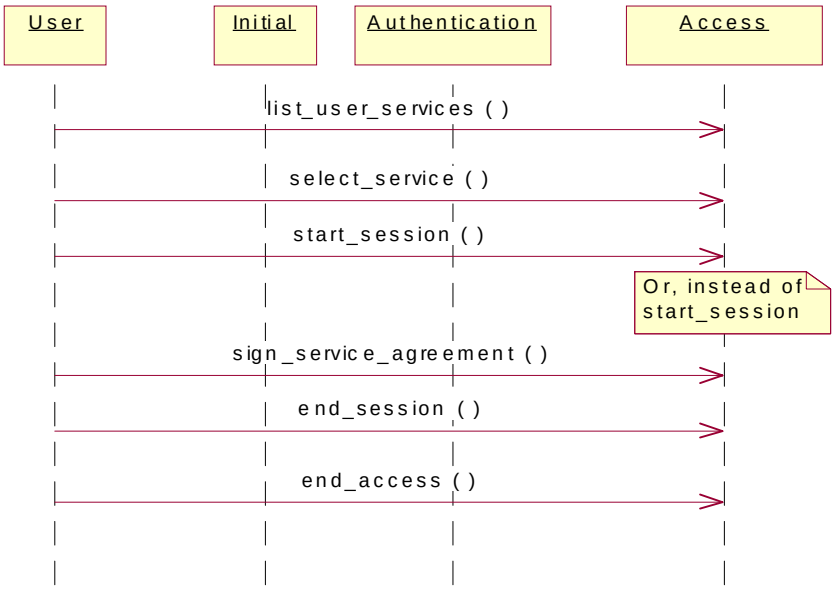

Figure 2-4 Sequence diagram for accessing services

The user uses **list_available_services()** to retrieve the **ServiceId** of the service they wish to use. The **select_service()** operation is used to inform the provider that the user wishes to use the service. Then the **start_session()** operation is used to initiate the session and return an interface reference to the service. Alternatively, the **sign_service_agreement()** operation can be used when non-repudiation of the request to initiate the session is required. A service session can be ended by using the **end_session** operation. As a result, the interfaces offered by the service are not longer available to the user. The complete process is described in more detail in the following section.

The **end_access** operation is used to end the user's access session with the provider. After it is invoked, the user will no longer be authenticated with the provider. The user will not be able to use the references to any of the provider interfaces gained during the access session. Any calls to these interfaces will fail.

The **Access** interface is also offered by the user to the provider to allow it to initiate interactions during the access session.

2.3.1 Access Interface

During an authenticated access session the user will be able to select and access services. In order to use a service, the user must be authorized to use the service having establishing a service agreement.

Service agreements can be concluded using either off-line or on-line mechanisms. Off-line agreements will be gained outside of the scope of TSAS interactions and so are not described here. However, users can make use of service agreements that are made off-line. Some providers may only offer off-line mechanisms to conclude service agreements. On-line service agreements may be concluded by using other TSAS provider interfaces, such as the interfaces defined by the subscription segments.

After a service agreement has been established between the user and the provider, the user will be able to make use of this agreement to access a service. The user can use the operations on the **Access** interface to:

- list the services which it can use,
- select the service it wish to use with some specific service properties, and
- to start the service session.

The **list_available_services()** operation is used to provide a list of services, which the user can use. The user can specify a list of properties that the service must match in order to scope the range of services returned.

The **select_service()** operation is used to identify the service that the user wishes to use. A list of service properties initializes the service and a service token is returned.

The user starts the service session by using the **start_session()** operation. This operation uses the service token to identify the service, with specific service properties, from which to create a new service session. The operation returns a **SessionInfo** structure that contains the **SessionId**, **SessionPropertyList**, and an **InterfaceList** with references to interfaces offered by the service session implementation.

Alternatively, after the service has been selected, the **sign_service_agreement()** operation can be used to start the service session. This operation is used when the user and provider wish to have non-repudiation for the request to start the service. The **sign_service_agreement()** operation allows the user to sign the service agreement for this service confirming their identity to the provider.

2.3.1.1 list_available_services()

```
void list_available_services (
        in ListedServiceProperties desired_properties,
        out ServiceList service_list)
            raises (
        PropertyError,
        ListError );
```

The **list_end_user_services()** returns a list of the services that are immediately available to the user. It can be noted that the list that is returned can contain services to which the user is already subscribed, as well as services that are (momentarily) offered for free (for which no subscription is required, see section 6 for details on subscription)

The **desired_properties** parameter can be used to scope the list of services. **desired_properties** identifies the properties that the services must match. For example, such a property can indicate that the services returned in the list must all be

currently available. **ListedServiceProperties** also defines whether a service must match one, all or none of the properties (see **MatchProperties** in section Section 5.1.1, “Properties and Property Lists,” on page 5-1). Currently no specific property names and values have been defined for **ListedServiceProperties** (‘available’ or ’subscribed’ would be a good example though), and so its use is service provider specific.

The list of services that matches the **desired_properties** is returned in the ServiceList. This is a sequence of **ServiceInfo** structures which contain the **ServiceId**, **UserServiceName** (consumer’s name for the service), and a sequence of service properties, **ServicePropertyList**. The **ServiceId** is associated with a specific service when the service is subscribed.

The value of **Service_id** is unique among all the available services, bur may be different for different users. The **service_id** value persists for the lifetime of the contractual relation between user and provider concerning this service.

Currently no specific property names and values have been defined for **ServicePropertyList**, and so its use is service provider specific.

If the **desired_properties** parameter is wrongly formatted, or provides an invalid property name or value, the PropertyError exception should be raised. Property names that are not recognized can be ignored if **desired_properties** require that only some, or none of the properties are matched. If the service list is unavailable because the retailer’s services are not available, then the operation should raise a ListError exception with the ListUnavailable error code.

The operation delivers a list of the services which the user may use. It can be noted that the list that is returned can contain services that are offered for free (for which no subscription is required).

2.3.1.2 select_service()

```
void select_service (
    in ServiceId service_id,
    in ServicePropertyList service_properties,
    out ServiceToken service_token)
 raises (
    ServiceError,
    PropertyError );
```

This operation is used by the user to identify the service that they wish to use.

service_id identifies the service required. It may be gained by using **list_available_servicesuser_services** or by some other means. The **service_id** is unique among all the available services. The **service_id** value persists for the lifetime of the contractual relation between user and provider concerning this service.

If the **service_id** is not recognized, then a ServiceError exception is raised with an InvalidServiceId error code. If the user is not allowed to use this service, a ServiceError exception is raised with a ServiceAccessDenied error code.

service_properties are a list of the service properties that the service instance should support. (These properties are used to initialize the service instance.) If a service property is not recognized by the provider, a PropertyError exception is raised.

The returned **service_token** is a free format text token returned by the provider, which can be used to start a service session with the selected service properties. This token contains provider specific information relating to the service agreement. The user is not intended to be able to 'decode' or understand the service token. The user merely offers the service token when they wish to gain a reference to the service session (either using **start_session**, or **sign_service_agreement**.) The **ServiceToken** may have a limited lifetime. If the lifetime of the **ServiceToken** expires, a method accepting the **ServiceToken** will raise a ServiceError with an InvalidServiceToken error code. **ServiceToken**s will not be accepted if the access session has been terminated (that is, the user or provider invokes the **end_access** operation on the other's **Access** interface).

2.3.1.3 start_session()

```
void start_session (
    in ServiceToken service_token
    in ApplicationInfo  app,
    out SessionInfo session_info)
raises (
    ServiceError );
```

This operation is used by the user to start a service session and is an alternative operation to **sign_service_agreement**. The service session corresponds to the service token (that is, the service session is a session of the service type), and has the service properties selected when the service token was generated (using **select_service()**).

service_token is returned by the provider in the call to **select_service()**. This token is used to identify the service type and service properties selected by the user. If the **service_token** is invalid, or has expired, a ServiceError exception is raised with an InvalidServiceToken error code.

The returned **session_info** is a structure containing information about the started service session instance. It includes the **SessionId**, **SessionPropertyList**, and a list of interfaces relating to the service session.

2.3.1.4 sign_service_agreement()

```
void sign_service_agreement(
    in ServiceToken service_token,
    in string agreement_text,
    in SigningAlgorithm signing_algorithm,
    out SignatureAndSessionInfo signature_session_info)
raises (
    ServiceError,
```

```
        ServiceAgreementError );
```

This operation is used by the user to request that the provider signs a service agreement before the user is allowed to use the service. The service agreement provides non-repudiation that the user requested to use the service and gain access to a service session.

service_token is the token returned by the provider in the call to **select_service()**. This token is used to identify the service type and service properties selected by the user. If the **service_token** is invalid, or has expired, a ServiceError exception is raised with an InvalidServiceToken error code.

agreement_text is the service agreement text that is to be signed by the provider.

signing_algorithm is the algorithm used to compute the digital signature of the service agreement.

Returned is a structure containing the digital signature of the provider for the **agreement_text** and the session information.

```
struct SignatureAndSessionInfo {
    string digital_signature;
    SessionInfo session_info;
};
```

The **digital_signature** is a signed version of a hash of the service token and agreement text. The mechanism to compute the digital signature is given by **signing_algorithm**.

session_info is a structure containing information about the service session. It includes the **SessionId**, **SessionPropertyList**, and a list of interfaces relating to the service session.

2.3.1.5 end_access()

```
void end_access (
    in EndAccessPropertyList end_access_properties )
raises ( PropertyError );
```

This operation is used to end the user's access session with the provider. The user requests that its access session is ended. After it is invoked, the user will no longer be authenticated with the provider. The user will not be able to use the references to any of the provider interfaces gained during the access session. Any calls to these interfaces will fail.

end_access_properties is a **PropertyList** defining the actions to be taken by the provider in ending the access session (for example, the **end_access_properties** may define the action to be taken if **end_access()** is called while there are active service sessions). If the properties are invalid, a PropertyError exception is raised.

2.3.1.6 *end_session()*

```
void end_session (
    in SessionId session_id )
raises ( SessionError );
```

This operation is used to end a service session. After it is invoked, the service session associated with the **SessionID** will have ended and will not be accessible to the user, (that is, the user will no longer be able to use any of the references to the session's usage interfaces).

session_id identifies the session to end. If the **session_id** is invalid, a SessionError exception is raised with an InvalidSessionId error code.

2.3.1.7 *list_segments()*

```
void list_segments (
    out SegmentIdList segment_ids );
```

This operation is used to list the segments offered by the provider. Segments other than this core segment are optional, and so only a subset of the segments defined by TSAS may be offered by a provider. The **segment_ids** returned by this operation only include segment identifiers to segments that are offered by this provider and are available to this user.

2.3.1.8 *get_segment()*

```
void get_segment (
        in SegmentId segment_id,
        in InterfaceList user_refs,
        out InterfaceList provider_refs )
    raises (
        SegmentError,
        InterfaceError );
```

This operation is used to establish a segment between the user and the provider.

- **segment_id** identifies the segment to be established. The segment defines a number of interface to be offered by the user and the provider. If the **segment_id** is invalid, the provider raises a SegmentError exception with an InvalidSegmentId error code.
- **user_refs** is a list of interfaces supported by the user. It must include references to all interfaces of the types which are required for this segment on the user side. If a required interface is missing from the list, a SegmentError exception is raised with a RequiredSegmentInterfaceNotSupplied error code, and the interface name is returned. If an interface is not part of the segment interfaces, a SegmentError exception is raised, with an InvalidSegmentInterface error code, and the interface name is returned.

A list of interfaces supported by the segment is returned. It must include references to interfaces of the types which must be supported by the provider for this segment.

2.3.1.9 release_segment()

```
void release_segments (
    in SegmentIdList segment_ids )
raises (
    SegmentError );
```

This operation is used to release segments that have been established between a user and provider. Once a segment is released, the interfaces associated with the segment cannot be used.

segment_ids is a list of segment identifiers of segments to be released. If a segment identifier is invalid, a SegmentError exception is raised with an InvalidSegmentId error code.

Service Access Segments 3

Contents

This chapter contains the following sections.

3.1 Overview

This chapter describes segments that are defined for controlling the access between domains and for controlling the access to services. In the scope of the ‘Telecommunication Service Access and Subscription’ (TSAS), these inter-domain accesses and services access take place on the one hand between the consumer domain and the retailer domain, and on the other hand between the retailer domain and the service provider domain.

The service access segments address two types of access functionality:

- functionality dedicated to the control of (inter-domain) access sessions, which in turn is specialized into the control of the access between the
  - consumer domain and the retailer domain, and
  - the retailer domain and the service provider domain.
- functionality related to accessing services, for which the consumer domain invokes the retailer domain, and the retailer invokes in its turn one or more service providers. These service providers support the actual services.

The functions dedicated to accessing domains consist of retrieving a list of active access sessions and a facility to terminate active access sessions.

The functions dedicated to accessing services are:

- set up of the default context required for the control of access sessions and control of service sessions,
- discovery of service offerings, including the retrieval of detailed service description information,
- listings of service sessions and services, and
- control of service sessions from the access session (e.g., resume, invite, join, notify changes, etc.).

Some of the service access segments define two asymmetric interfaces, one to be supported by the user domain and the other to be supported by the provider domain. In that case the interface name will include ‘user’ or ‘provider’ in order to avoid confusion.

Figure 3-1 illustrates the various interfaces offered by each domain.

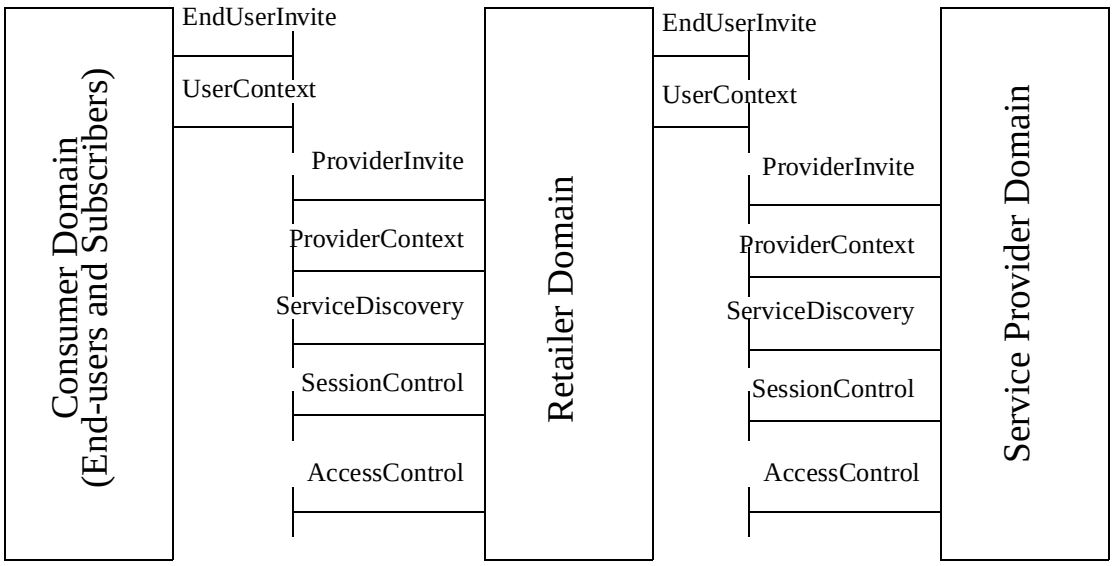

Figure 3-1 Interfaces supported by the TSAS domains

3.2 Service Access Segment Interfaces

This section globally describes the service access segments, their interfaces, and their operations in a generic fashion (that is, for the generic roles of user and provider). This generic specification can be re-used for the specific cases of, on the one hand end-user and retailer, and on the other hand retailer and service provider.

The segments available for use during an access session are:

Invitation segment

It allows the control of invitations and announcements. It defines two interfaces:

1. **EndUserInvite** - This interface is used by the service provider to notify the end-user (via its retailer) of invitations to join service sessions.
2. **ProviderInvite** - This interface allows a known user to get a list of session invitations and session announcements and to join these sessions, and to reply to invitations.

Context segment

It allows the control of configuration (context) information. It defines two interfaces:

1. **UserContext** - This interface is used by the provider within an access session to access user configuration information.
2. **ProviderContext** - This interface allows a known user to set configuration information at the provider side, and to verify the current settings by retrieving that same information when required.

Access control segment

It provides supplementary functionality for access session control. It defines one interface:

1. **AccessControl** - This interface allows a known user to get a list of running access sessions and to end one or more of them.

Service discovery segment

It supports functionality helping to learn about (new) services. It defines one interface:

1. **ServiceDiscovery** - This interface allows a known user to get a list of subscribed services, to discover new services, and to get supplementary information about services.

Session control segment

It provides functionality for service session control. It defines one interface:

1. **SessionControl** - This interface allows a known user to get a list of running service sessions and to resume service sessions or participation in service sessions (when these have been suspended), and to end service sessions.

Access session information segment

It allows a user to receive information over all its access sessions with this provider. It defines no interface but rather uses either the CORBA **CosNotification** service or the CORBA **CosEvent** service.

Service session information segment

It allows an end-user to receive information over all its service sessions (possibly across several access session). It defines no interface but rather uses either the CORBA **CosNotification** service or the CORBA **CosEvent** service.

These segments, interfaces, and the operations they provide are described below.

3.2.1 Base Interface

Since it must be possible to release any segment that is set up from within the segment, all the interfaces defined for the service access segments and for the subscription segments inherit from a base interface that defines an operation **release_segment()**. This base interface is defined in the common types (see Section 5.6.1, “Base Interface,” on page 5-10).

3.3 Invitation Segment

The invitation segment defines two interfaces, the **EndUserInvite** interface and the **ProviderInvite** interface.

EndUserInvite Interface

The **EndUserInvite** interface allows the service provider to send invitations to join a service session during an end-user’s access session with its retailer.

- **invite_user()** - allows the provider to invite the user to join a service session. A session description and sufficient information to join the session is available in the parameter list. The session can only be joined using the **join_session_with_invitation()** operation on the **ProviderInvite** interface.
- **cancel_invite_user()** - allows the provider to inform the user that an invitation previously sent to the user has been cancelled.

ProviderInvite Interface

The **ProviderInvite** interface allows a known user to get a list of session invitations and session announcements and to join these sessions, and to reply to invitations. It provides the following operations:

- **list_session_invitations()** - lists the invitations to join a service session that have been sent to the user.
- **list_session_announcements()** - lists the service sessions with have been announced. It can be scoped by some announcement properties.

- **join_session_with_invitation()** - allows the user to join a service session to which he has been invited.
- **join_session_with_announcement()** - allows the user to join a service session which has been announced.
- **reply_to_invitation()** - 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).

As will be shown in the scenario examples for the invitation segment, the **UserId** (user identity) must be exchanged between the Retailer and the Service Provider. However, the anonymity of the end-user can still be guaranteed by the following two facts:

- The value used for **UserId** between end-user and retailer, and between retailer and service provider, can differ (however the publicly known one is the one used by the Service Provider to reach an end-user).
- With a **UserId** value as above, the service provider cannot contact the end-user without transiting through the retailer (or at least local mechanisms can be foreseen to ensure that).

3.3.1 *EndUserInvite Interface*

```
interface EndUserInvite: SegmentBase
{
};
```

The **EndUserInvite** interface allows a service provider to invite an end-user (via its retailer) to join a service session, and to cancel pending invitations when required. This interface is returned as a result of the **Core::Access::get_segment()** operation establishing this segment.

3.3.1.1 *invite_end_user()*

```
void  invite_end_user (
        in  SessionInvitation invitation,
        out InvitationReply reply
) raises (
        InvitationError
);
```

The **SessionInvitation** and **InvitationReply** parameters are defined according to the Internet Engineering Task Force working group MMUSIC, (Multimedia Multiparty Session Control) draft standard ‘Session Initiation Protocol.’ This operation allows a service provider to invite an end-user (via its retailer) to join a service session. It can only be used during an access session. The service provider sends the invitation to the appropriate retailer, and the retailer to the consumer domain(s) at which the end-user can be reached.

SessionInvitation describes the service session to which the end-user has been invited and provides an **InvitationId** to identify this invitation when joining. It does not give interface references to the session, nor any information that would allow the end-user to join the service session outside the context of an access session with its retailer.

An **InvitationReply** is returned that allows the end-user to inform the retailer of the action it will take regarding the invitation (for more details, see Section 5.2.2, "Invitations and Announcements," on page 5-4).

The end-user may join the service session described by the invitation from within this access session, or it may establish another access session with this retailer. The same **InvitationId** will refer to this invitation in both access sessions. The end-user should use the operation **join_session_with_invitation()** from the **ProviderInvite** interface of this invitation segment. Note that the service session cannot be joined without an access session with the retailer.

3.3.1.2 cancel_invite_end_user()

```
void cancel_invite_end_user (
        in UserId invitee_id,
        in InvitationId id
    ) raises (
        InvitationError
);
```

This operation allows a service provider to cancel an invitation to join a service session that has been sent to an end-user.

InvitationId is used to determine the invitation to be cancelled. **InvitationId**s are unique across all access sessions with the same service provider.

If the **InvitationId** list is unknown to the consumer domain (receiving the **cancel_invite_end-user** on behalf of its end-user), then the operation should raise an InvitationError exception with the InvalidInvitationId error code. It is possible to receive a **cancel_invite_end_user** before a corresponding **invite_end_user**. This operation should raise the exception anyway.

3.3.1.3 Scenarios

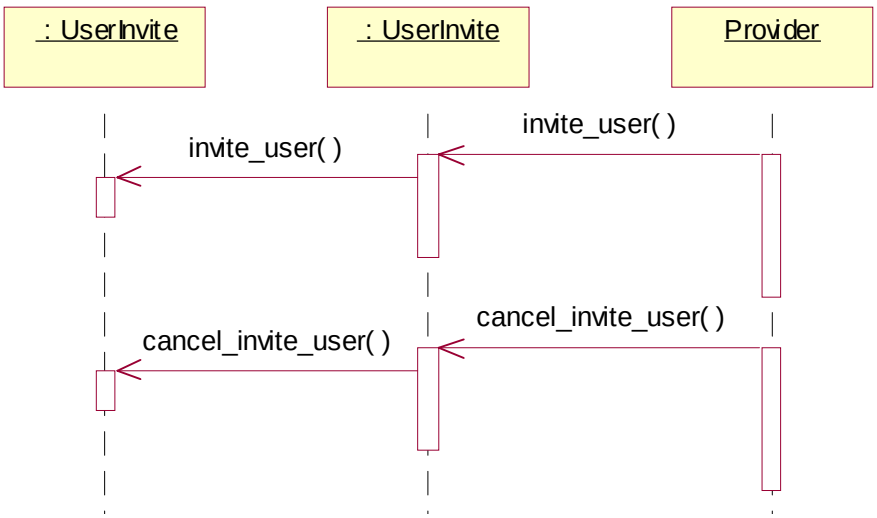

Figure 3-2 Invitation Segment - EndUserInvite Diagram

3.3.2 ProviderInvite Interface

```
interface ProviderInvite: SegmentBase
{
};
```

The **ProviderInvite** interface allows a known end-user to retrieve information related to announced service sessions or to invitations meant for that user. The end-user can use this interface to reply to invitations or request to join announced service sessions or service sessions it has been invited to. This interface is returned as a result of the **Core::Access::get_segment()** operation establishing this segment.

3.3.2.1 list_session_invitations()

```
void  list_session_invitations (
        out InvitationList invitations
) raises (
        ListError
);
```

The **list_session_invitations()** returns a list of the invitations to join a service session, which have been sent to the end-user through this retailer.

The **InvitationList** returned by this operation is a sequence of **SessionInvitation** structures:

```
struct SessionInvitation {
        InvitationId id;
```

```
        UserId invitee_id;
        SessionPurpose purpose;
        InvitationReason reason;
        InvitationOrigin origin;
};
```

- **id** - identifies the particular invitation. It uniquely identifies this invitation from others for this end-user at this retailer (other end-users with this retailer may have invitations with the same id). This **id** is used in **join_session_with_invitation()** (see below) to join the session referred to by this invitation.
- **invitee_id** - is the user id of this end-user. This information is not strictly necessary here as the user id is known in the access session (in which this invitation segment is established). However, it is included to make the structure more re-usable, and to allow the recipient to check that the invitation was for him.
- **purpose** - is a string containing the purpose of the session.
- **reason** - is a string containing the reason this end-user has been invited to join this session.
- **origin** - is a structure containing the **user_id** of the end-user that requested that the invitation was sent to this end-user, and the **session_id** of the service session to join.

If the invitation list is not available, then the operation should raise the ListError with the ListUnavailable error code.

3.3.2.2 list_session_announcements()

```
void  list_session_announcements (
        in AnnouncementSearchProperties  desired_properties,
        out AnnouncementList announcements
    ) raises (
        PropertyError,
        ListError
);
```

The **list_session_announcements()** returns a list of the session announcements that have been announced through this retailer. As the retailer plays the role of a one-stop-shop to the end-user, this list of announcements can be a collection of lists from several service providers that are in contact with this retailer. The service sessions are announced either by requests from session participants (service provider specific), or due to properties established at service session start-up. The process by which sessions are announced is not defined by TSAS. However, this operation is provided in order to allow an end-user to request a list of service sessions that have been announced. The announcements may be scoped in order to restrict the distribution of the announcement to particular groups. This operation returns a list of announcements that match the **desired_properties**, as specified by the end-user.

The **desired_properties** parameter can be used to scope the list of announcements. **AnnouncementSearchProperties** identifies the properties that the announcements must match. (See **MatchProperties** in Section 5.1.1, “Properties and Property Lists,” on page 5-1). Currently no specific property names and values have been defined for **AnnouncementSearchProperties**, and so its use is service provider specific.

The returned **AnnouncementList** is a list of announcements available to the end-user and matching the **desired_properties**. This is a sequence of **SessionAnnouncement** structures that contain the properties of the announcement, (that is, **AnnouncementProperties**). Currently no specific property names and values have been defined for AnnouncementProperties, and so its use is service provider specific.

If the **desired_properties** parameter is wrongly formatted, or provides an invalid property name or value, the PropertyError exception is raised. Property names that are not recognized can be ignored if **desired_properties** requires that only some, or none of the properties are matched.

If an announcement list is not available, then the operation should raise the ListError, with the ListUnavailable error code.

3.3.2.3 join_session_with_invitation()

```
void  join_session_with_invitation (
        in InvitationId  invitation_id,
        in ApplicationInfo  app,
        in JoinPropertyList  join_properties,
        out SessionInfo session_info
    ) raises (
        SessionError,
        InvitationError,
        ApplicationInfoError,
        PropertyError
);
```

The **join_session_with_invitation()** allows the end-user to join an existing service session, for which it has received an invitation.

- **invitation_id** - is the identifier of the invitation. The invitation, kept by the retailer, contains sufficient information for the retailer to contact the service session at the service provider's domain, and request that the end-user be allowed to join the service session.
- **app** - is an **ApplicationInfo** structure containing information on the end-user application that will be used to interact with the service session. It provides an application name, version, serial number, license number, and a list of properties:

```
struct ApplicationInfo {
        string name;
        string version;
        string serial_num;
        string licence_num;
```

```
        PropertyList properties;
};
```

join_properties is a **PropertyList**. It can contain information related to the end-user that is requesting to join the session, such as for example a motivation for joining. Currently no specific property names and values have been defined for **JoinPropertyList**, and so its use is service provider specific.

A **SessionInfo** is returned as a result of a successful joining. It is a structure that contains information that allows the end-user to refer to this service session using other operations on this interface. It also contains information for the usage part of the session, including the interface references to interact with the service session (service provider specific).

The exception SessionError is raised if the service session refuses to allow the end-user to join it.

The exception InvitationError is raised if the **invitation_id** is invalid.

The exception ApplicationInfoError is raised if there are unknown or invalid values for **ApplicationInfo**, or if the application is incompatible with the type of service being joined.

If the **join_properties** parameter is wrongly formatted, the PropertyError exception is raised.

3.3.2.4 *join_session_with_announcement()*

```
void  join_session_with_announcement (
        in AnnouncementId announcement_id,
        in ApplicationInfo app,
        in JoinPropertyList  join_properties,
        out SessionInfo session_info
    ) raises (
        SessionError,
        AnnouncementError,
        ApplicationInfoError,
        PropertyError
);
```

The **join_session_with_announcement()** allows the end-user to join an existing service session, for which the end-user has discovered an announcement. The service session announcements are obtained by using the **list_session_announcements** operation on the same interface, or in a number of other ways that are not described by TSAS (can be retailer and service provider specific), including through a specialized service session.

- **announcement_id** - is the identifier of the announcement. The announcement information forwarded by the retailer, contains sufficient information for the retailer to contact the service session at the service provider domain, and request that the end-user be allowed to join the service session.
- **ApplicationInfo** and **SessionInfo**: same as above.

- **join_properties**: same as above.

The exceptions SessionError, ApplicationInfoError, and PropertyError: same as above.

The exception AnnouncementError is raised if the **announcement_id** is invalid.

3.3.2.5 *reply_to_invitation()*

```
void reply_to_invitation (
        in InvitationId invitation_id,
        in InvitationReply reply
    ) raises (
        InvitationError,
        InvitationReplyError
);
```

The **reply_to_invitation()** allows the end-user to reply to a received invitation. This has two purposes. The first one is to enable the end-user to reply if it did not get the invitation when it was issued, and it had to be stored and momentarily be replied to by the retailer. The second possibility is for the end-user to reply with a different reply code than the one used in the case the issued invitation was originally received by the end-user and replied to. This latter possibility, however, should only be used in 'exception' scenarios.

This operation is used by the end-user to inform the service provider about its reaction (its reply) to the invitation. The end-user can use one or more **reply_to_invitation()** invocations to indicate ‘busy,’ then ‘ringing,’ then to indicate its intention to join, or not, the session, or to indicate a different location to look for the end-user. This operation is not used to join service sessions, the **join_session_with_invitation()** and **join_session_with_announcement()** must be used for that purpose. In order not to confuse the service session that issued the invitation, it is recommended not to use multiple **reply_to_invitation()** to the same invitation.

- **invitation_id** - is the identifier of the invitation.
- **reply** - is a structure which contains the information about the end-user's reply. For details see section Section 5.2.2, “Invitations and Announcements,” on page 5-4.

The exception InvitationError is raised if the **invitation_id** is invalid. The exception InvitationReplyError is raised if there is an error in the reply.

3.3.2.6 Scenarios

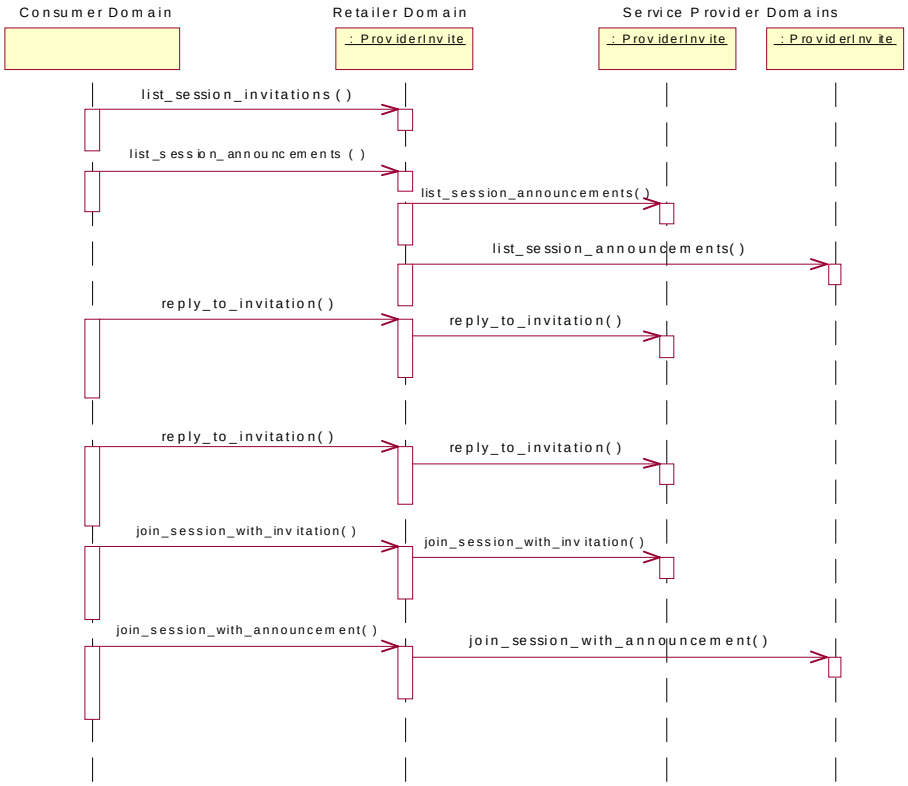

Figure 3-3 Invitation Segment - ProviderInvite Diagram

- The invitations are stored in the retailer domain, so that the **list_session_invitation()** does not need to be invoked on the service provider domains.
- The announcements should not be stored in the retailer domain because these should be checked only upon request. As shown in the scenario the end-user in the consumer domain invokes the **list_session_announcements()** on its retailer only once. The retailer acting as a one-stop shop to the end-user can invoke several service provider domains to collect session announcements and compile them in a single list that is returned to the end-user.
- The end-user can reply to an invitation without requesting to join the corresponding service immediately. Several replies can be sent on the same invitation, but it is recommended that only one be used.
- The requests to join the service session, with invitation or announcement, must be forwarded by the retailer to the appropriate service provider.

3.4 Context Segment

The context segment defines two interfaces, the **UserContext** interface and the **ProviderContext** interface.

The **UserContext** interface allows the provider to gain information about the user domain's configuration, and applications.

- **get_user_ctxt()** - allows the provider to retrieve information about the user domain's configuration.

The **ProviderContext** interface allows a known user to set configuration information at the provider domain side.

It provides the following operations:

- **set_user_ctxt()** - allows the user to inform the provider about interfaces in the user domain, and other user domain information. (for example, user applications available in the user domain, operating system used).
- **get_user_ctxts()** - allows the user to retrieve one or more sets of configuration (context) information that has been stored in the provider domain.

3.4.1 UserContext Interface

```
interface UserContext: SegmentBase
{
};
```

This interface allows the provider to gain information about the user domain's configuration and applications. This interface could also be extended, (for example, to allow a provider to ask more specific questions about the user domain). This interface is returned as a result of the **Core::Access::get_segment()** operation establishing this segment.

3.4.1.1 get_user_ctxt()

```
void get_user_ctxt(
        out UserCtxt user_ctxt );
```

This operation allows the provider to receive all the information about the user domain's configuration that the user accepts the provider to have access to. In particular it can be used by the service provider to gain access, via the retailer, to information related to the end-user's consumer domain, such as terminal and applications used. The operation returns the **UserCtxt** structure that contains a property list enabling provider specific information to be included. See Section 5.5, "User Context Information," on page 5-8.

3.4.1.2 Scenario

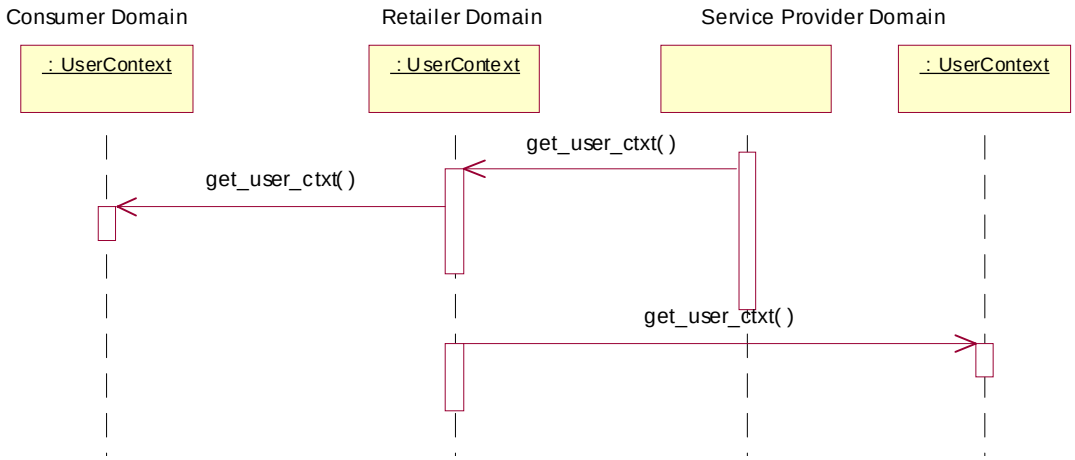

Figure 3-4 Context Segment - User Context Diagram

Usually the request for user context information is initially issued by a service provider and forwarded by the retailer to the end-user. However, as is explained above, the user context relates to the general user/provider scenario: for example, the service provider can be logged on the retailer domain as a user.

3.4.2 ProviderContext Interface

```
interface ProviderContext: SegmentBase
{
};
```

The **ProviderContext** interface allows a known user to set context information related to its domain. This interface is returned as a result of the **Core::Access::get_segment()** operation establishing this segment.

3.4.2.1 set_user_ctxt()

```
void set_user_ctxt (
            in UserCtxt user_ctxt
) raises (
            UserCtxtError
);
```

The **set_user_ctxt()** allows the user to inform the provider about the configuration of the consumer domain. In the particular case of the end-user, it can inform the service provider, via the retailer, of user applications available in the consumer domain, operating systems, etc.

user_ctxt - is a structure containing consumer domain configuration information and possibly interfaces (in the list of properties).

If there is a problem with **user_ctxt**, then UserCtxtError should be raised with the appropriate error code.

3.4.2.2 *get_user_ctxts()*

```
void  get_user_ctxts (
            in SpecifiedUserCtxt ctxt,
            out UserCtxtList user_ctxts
) raises (
            UserCtxtError,
            ListError
);
```

This operation allows the user to retrieve information about user contexts that have been registered with the provider.

ctxt - is a union specifying which contexts information must be returned, namely all, only the current ones, or a list of specified ones.

The returned **UserCtxtList** is a list of structures each containing user domain configuration information and properties such as, possibly, interfaces. This operation will raise a UserCtxtError exception with a UserCtxtNotAvailable error code if there is no context set. If there is a problem with **ctxt**, then UserCtxtError should be raised with the appropriate error code, and if the list is not available then the ListError is raised with the ListUnavailable error code.

3.4.2.3 *get_user_info()*

```
void  get_user_info(
            out UserInfo user_info );
```

The **get_user_info()** allows the user to request information about himself. This operation returns a **UserInfo** structure. This contains the user's UserId, its name, and a list of user properties. Currently no specific property names and values have been defined for **UserPropertyList**, and so its use is provider specific.

3.4.2.4 Scenarios

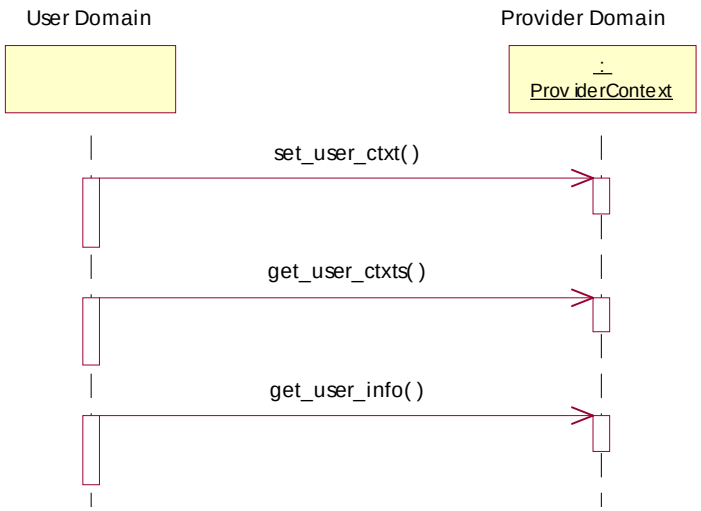

Figure 3-5 Context Segment - Provider Context Diagram

These scenarios are valid in the general case of a User domain accessing a Provider domain (end-user with retailer, retailer with provider, provider with retailer, or other domains to which the TSAS specifications would be applied).

3.5 Access Control Segment

The access control segment defines the **AccessControl** interface.

The **AccessControl** interface allows a known user to get a list of running access sessions, to end one or more of them, and to get the user information stored at that moment in the provider domain.

It provides the following operations:

- **list_access_sessions()** - allows the user in this access session to find out about other access sessions that he has with this provider. For example, an end-user is at work, but has an access session set up at home which runs an active security service session.
- **end_access_sessions()** - allows the user to end one or more specified access session(s). This can include the current one, or others, found using **list_access_sessions()**. The user can also specify some actions to be performed if there are active service sessions within the access session(s) to be ended.
- **get_user_info()** - gets the user's username and other properties.

3.5.1 AccessControl Interface

```
interface AccessControl: SegmentBase
{
```

```
};
```

The **AccessControl** interface allows a known user to control its access sessions. This interface is returned as a result of the **Core::Access::get_segment()** operation establishing this segment.

3.5.1.1 *list_access_sessions()*

```
void  list_access_sessions (
        out AccessSessionList as_list
    ) raises (
        ListError
);
```

The **list_access_sessions()** returns a list of access sessions. The list contains all the access sessions the user currently has established with this provider. It is a sequence of **AccessSessionInfo** structures, which consist of the **AccessSessionId**, **UserCtxtName**, and **AccessSessionPropertyList**. The last of these is a **PropertyList**. Currently no specific property names and values have been defined for **AccessSessionPropertyList**, and so its use is provider specific.

The information returned by this operation can be used by the user to find out which other access sessions are currently established, and to perform some operations on these access sessions as required, and as indicated below.

If the **AccessSessionList** is unavailable, because the user's access sessions are not available, then the operation should raise a ListError exception with the ListUnavailable error code.

3.5.1.2 *end_access_sessions()*

```
void end_access_sessions(
        in SpecifiedAccessSession as,
    ) raises (
        SpecifiedAccessSessionError
);
```

The **end_access_sessions()** allows the user to end one or more access session. The operation can end the current access session, a specified access session, or all access sessions (including the current one), through the use of the **SpecifiedAccessSession** parameter.

If as is wrongly formatted, or provides an invalid access session id, then the SpecifiedAccessSessionError exception should be raised.

3.5.1.3 Scenarios

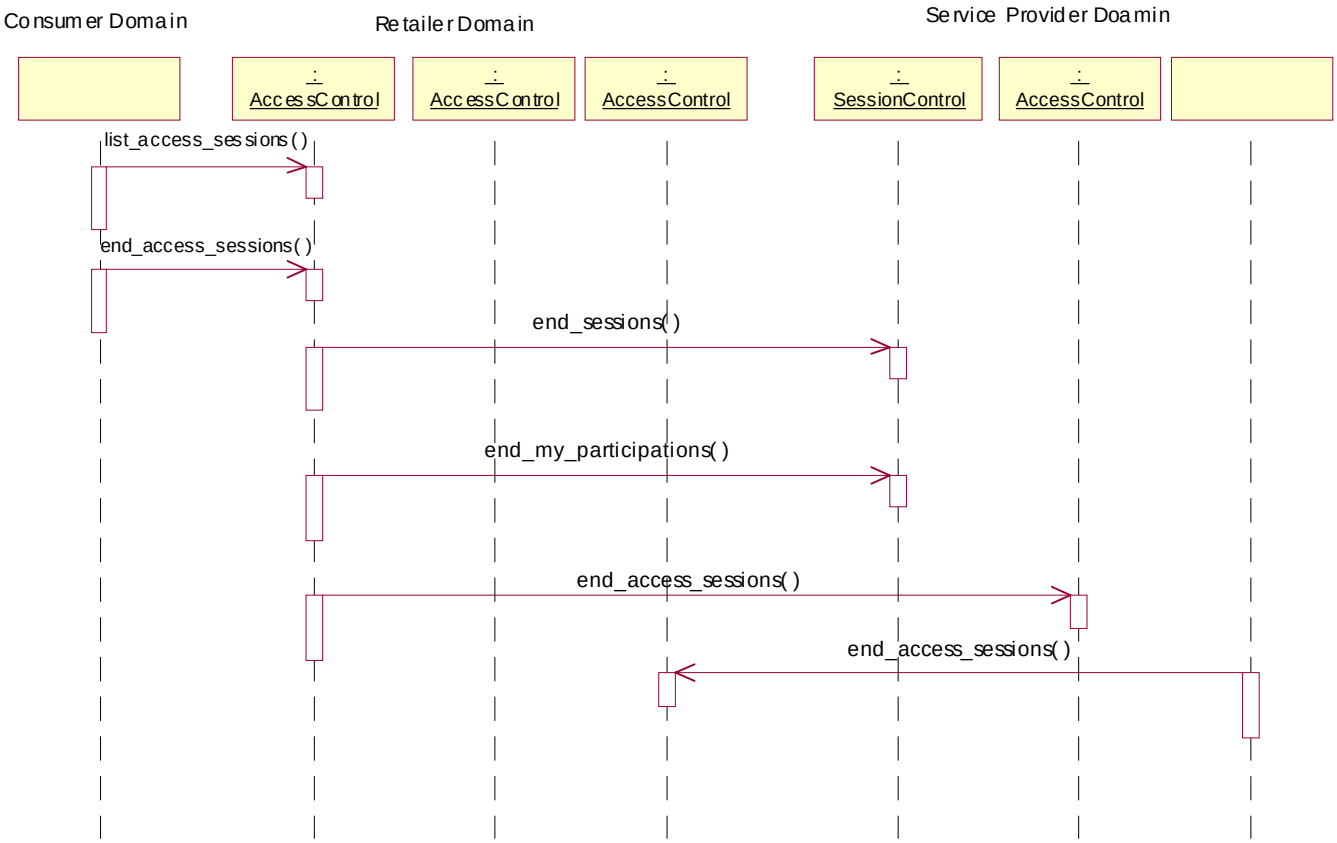

Figure 3-6 Access Control Segment Diagram

The end-user can invoke the retailer domain to list the running access sessions, and then end one or more of them (remote access sessions can be deleted as well).

As the retailers and service providers do access each other with the same access session mechanisms, the same **list_access_sessions()** and **end_access_sessions()** operations can be used to terminate these access sessions.

3.6 Service Discovery Segment

The service discovery segment defines the **ServiceDiscovery** interface.

The **ServiceDiscovery** interface allows a known user to get a list of subscribed services, to discover new services, and to get supplementary information on services.

It provides the following operations:

- **discover_services()** - lists all the services available via this retailer (and from the service providers). The user can scope the list by supplying some properties that the service should have, and a maximum number to return.
- **get_service_info()** - returns the service information for a particular service (identified in the invocation by its **service_id**). Similar information (**ServicePropertyList**) can be obtained with the **list_end_user_services** or **discover_services**, but the **get_service_info** is targeting on a single service and is independent from subscription.

3.6.1 ServiceDiscovery Interface

```
interface ServiceDiscovery: SegmentBase
{
};
```

TSASServiceDiscovery interface allows a known user to access information about its subscribed services, and to discover new services. This interface is returned as a result of the **Core::Access::get_segment()** operation establishing this segment.

3.6.1.1 discover_services()

```
void  discover_services(
        in DiscoverServiceProperties  desired_properties,
        in unsigned long  how_many,
        out ServiceList services
    ) raises (
        PropertyError,
        ListError
);
```

The **discover_services()** returns a list of the services available via this retailer.This operation is used to discover the services provided via the retailer, for use by the end-user. It can be scoped by the **desired_properties** parameter (see **MatchProperties** in Section 5.1.1, “Properties and Property Lists,” on page 5-1).

The retailer has the possibility to contact one or more service providers in order to fulfill the user's request. This takes place in a way totally transparent to the end-user. The retailer performs one or more invocations on one or more service providers and collects the information received from each service provider. This collected information is merged and provided to the end-user as one piece of information.

The list of retailer services matching the **desired_properties** is returned in services. This is a sequence of **ServiceInfo** structures which contain the **ServiceId**, **UserServiceName** (the end-users name for the service), and a sequence of service properties, **DiscoverServiceProperties**. Currently no specific property names and values have been defined for **DiscoverServiceProperties**, and so its use is service provider specific. Examples of **DiscoverServiceProperties** can be ‘free’ services, ‘comfort’ telephony services, ‘information retrieval’ services, ‘video on demand,’ ‘joint document editing,’ ‘payment,’ ‘calling card reload,’ etc.

The **how_many** parameter defines the number of **ServiceInfo** structures to return in the services parameter. The number of services shall not exceed that number.

If the **desired_properties** parameter is wrongly formatted, or provides an invalid property name or value, the PropertyError exception should be raised. Property names that are not recognized can be ignored if **desired_properties** requires that only some, or none of the properties are matched.

If the services list is unavailable, because the retailer's services are not available, then the operation should raise an ListError exception with the ListUnavailable error code.

3.6.1.2 *get_service_info()*

```
void  get_service_info (
          in ServiceId  service_id,
          in SubscribedServiceProperties  desired_properties,
          out ServicePropertyList service_properties
      ) raises (
          ServiceError,
          PropertyError
);
```

The **get_service_info()** returns information on a specific service, identified by the **service_id**. The **desired_properties** list can scope the information that is requested to be returned.

3.6.1.3 *Scenarios*

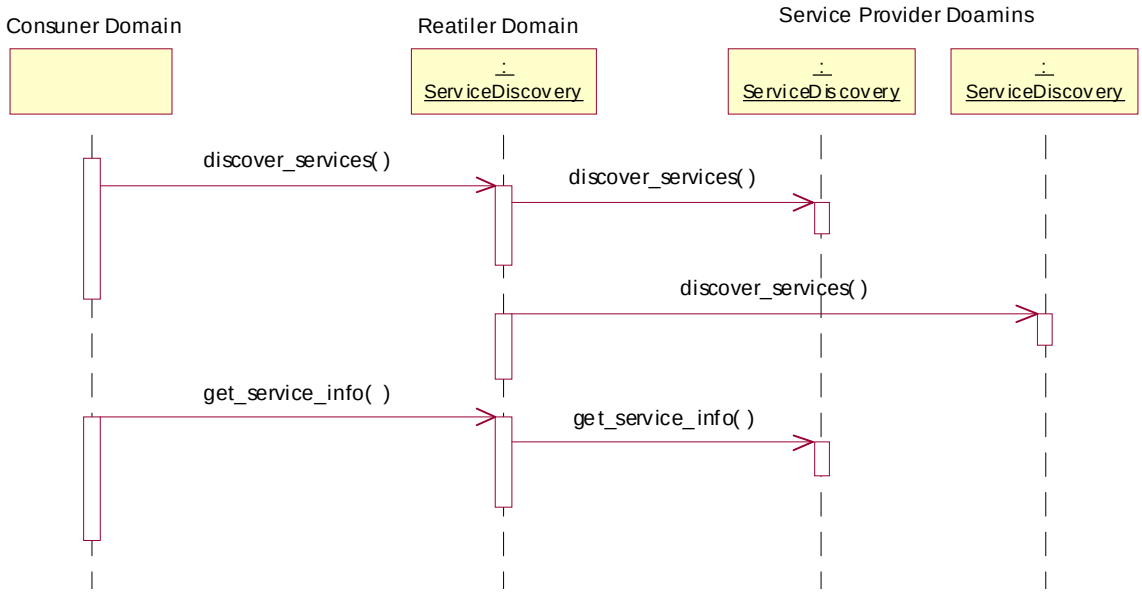

Figure 3-7 Service Discovery Segment Diagram

The **discover_services()** is invoked by the end-user on the retailer. The retailer subsequently invokes **discover_services()** on one or more service provider to fulfil the end-user's request. The retailer will return the compiled list of discovered services to the end-user.

3.7 Session Control Segment

The session control segment defines the **SessionControl** interface.

The **SessionControl** interface allows a known user to get a list of running service sessions and to resume service sessions or participation in service sessions (when these have been suspended).

It provides the following operations:

- **list_service_sessions()** - lists the service sessions of the user. The request can be scoped by the access session and session properties (for example, active, suspended, service type).
- **end_sessions()** - allows the user to end one or more service sessions.
- **end_my_participations()** - allows the user to end his participation in one or more service sessions, without ending the service session.
- **resume_session()** - allows the user to resume a service session.
- **resume_my_participation()** - allows the user to resume his participation in a service session.

3.7.1 SessionControl Interface

```
interface SessionControl: SegmentBase
{
};
```

The **SessionControl** interface allows a known user to list its running service sessions, and resume the suspended service sessions or the service sessions in which its participation has been suspended. This interface also provides an operation to end a list of service sessions, or to end the user's participation in a list of service sessions. This interface is returned as a result of the **Core::Access::get_segment()** operation establishing this segment.

3.7.1.1 list_service_sessions()

```
void  list_service_sessions (
        in SpecifiedAccessSession  sas,
        in SessionSearchProperties  desired_properties,
        out SessionList sessions
    ) raises (
        SpecifiedAccessSessionError,
        PropertyError,
        ListError
```

);

The **list_service_sessions()** returns a **SessionList** (list of the service sessions) which the end-user is involved in. This includes active and suspended sessions. The **sas** parameter scopes the list of sessions by the access session in which they are used. It can identify the current access session, a list of access sessions, or all access sessions. A session is associated with an access session if it is being used within that access session, or it has been suspended (or participation suspended), and was being used within that access session when it was suspended.

The **desired_properties** parameter can be used to scope the list of sessions. It identifies the properties that the sessions must match. It also defines whether a session must match one, all or none of the properties (see **MatchProperties** in Section 5.1.1, "Properties and Property Lists," on page 5-1). The following property names and values have been defined for **SessionSearchProperties**:

- name: "SessionState"
- value: SessionState

If a property in **SessionSearchProperties** has the name "SessionState," then the matching service session must have the same **SessionState** as given in the property value.

- name: "**UserSessionState**"
- value: **UserSessionState**

If a property in **SessionSearchProperties** has the name "UserSessionState," then the matching service session must have the same **UserSessionState** as given in the property value.

Other provider specific properties can also be defined in **desired_properties**.

The list of sessions matching the **desired_properties** and the access session **sas** are returned in sessions. This is a sequence of **SessionInfo** structures which define the **SessionId**, and a series of provider specific information, such as information on existing service session participants, references to service session control interfaces, etc. This information is provider specific, and consequently out of the scope of TSAS.

If sas is wrongly formatted, or provides an invalid access session id, then the SpecifiedAccessSessionError exception should be raised.

If the **desired_properties** parameter is wrongly formatted, or provides an invalid property name or value, the PropertyError exception should be raised. Property names that are not recognized can be ignored if **desired_properties** requires that only some, or none of the properties are matched.

If the sessions list is unavailable because the end-user's sessions are not known, then the operation should raise a ListError exception with the ListUnavailable error code.

3.7.1.2 end_sessions()

void end_sessions (

```
        in SessionIdList session_id_list
    ) raises (
        SessionError
);
```

The **end_sessions()** ends one or more service sessions, identified by **session_id_list**. The SessionError exception is raised if there is an unrecognized **session_id** in the list.

3.7.1.3 *end_my_participations()*

```
void end_my_participations (
        in SessionIdList session_id_list
    ) raises (
        SessionError
);
```

The **end_my_participations()** ends the user's participation in one or more service session identified by **session_id_list**, without ending the service session. The SessionError exception is raised if there is an unrecognized **session_id** in the list.

3.7.1.4 *resume_session()*

```
void  resume_session (
        in SessionId session_id,
        in ApplicationInfo app,
        out SessionInfo session_info
    ) raises (
        SessionError,
        ApplicationInfoError
);
```

The **resume_session()** resumes a service session. It is used on a service session that is suspended. The suspension and resuming operations are mainly used to obtain service session mobility. The service session can be resumed within an access session different from the one in which the service session was initially running, which possibly involves a different terminal as well. As the operation required to suspend a service session involves service session mobility, the mechanism required to suspend the service session might be service specific, and is therefore not provided by TSAS, but should be defined on one of the service specific interfaces.

session_id - is the identifier of the session to be resumed.

The **ApplicationInfo** is a structure containing information on the application which will be used to interact with the resumed service session. This application may be different to the user's original application that was used when the session was suspended, because as was said above, the service session can be resumed within a different access session using a different terminal and different applications. The structure of **ApplicationInfo** was explained in Section 3.3.2.3, "join_session_with_invitation()," on page 3-9".

The returned **SessionInfo** is a structure that contains information that allows the consumer domain to refer to this service session using other operations on this interface. It also contains information for the usage part of the session, including the interface references to interact with the service session (service provider specific).

The exception SessionError is raised if **session_id** is invalid, or if the session refuses to resume because of the user's session state, or if the user does not have permission.

The exception ApplicationInfoError is raised if there are unknown or invalid values for **ApplicationInfo**, or if the application is incompatible with the type of service being resumed.

3.7.1.5 resume_my_participation()

```
void  resume_my_participation (
            in SessionId session_id,
            in ApplicationInfo app,
            out SessionInfo session_info
) raises (
            SessionError,
            ApplicationInfoError
);
```

The **resume_my_participation()** resumes the end-user's participation in a service session. It is used on a session that the end-user has previously suspended his participation from. See above for more details on the suspend-resume mechanisms.

session_id - is the identifier of the service session to resume the user's participation.

app and the returned **SessionInfo**: same as above.

The exceptions SessionError and ApplicationInfoError: same as above.

3.7.1.6 Scenarios

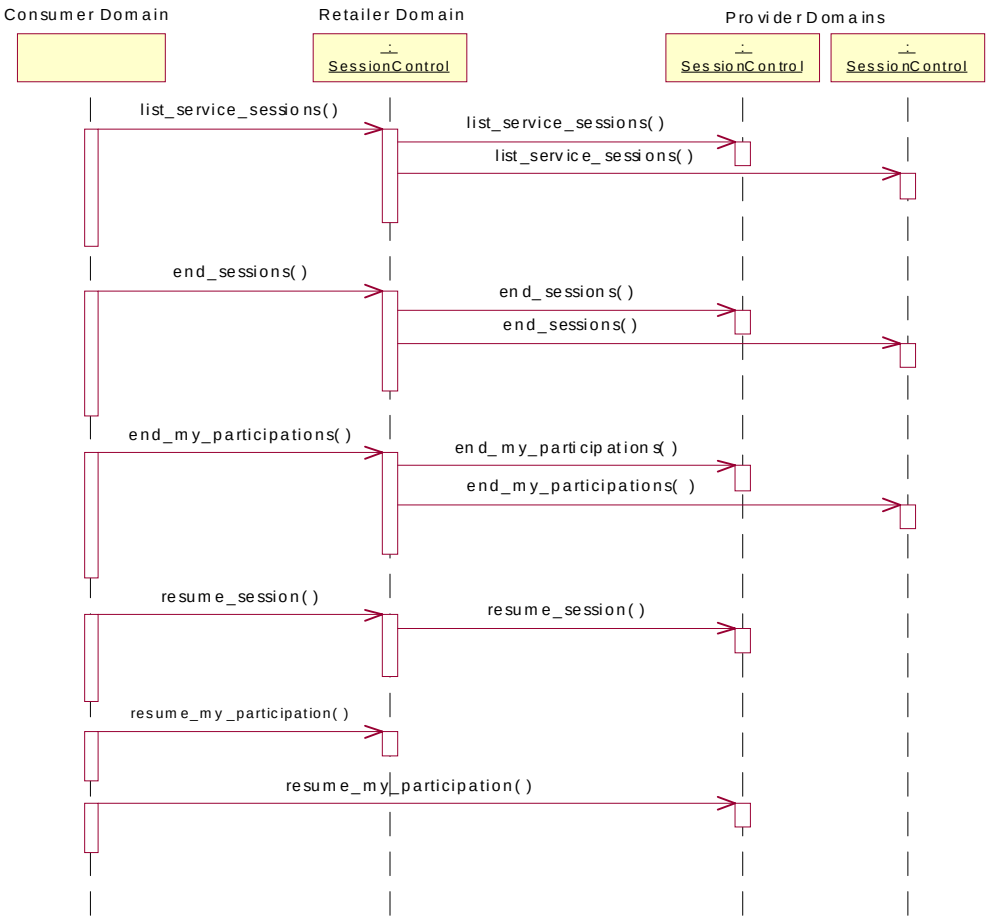

Figure 3-8 Session Control Segment Diagram

The **list_service_sessions()**, **end_sessions()** and **end_my_participations()** are invoked by the end-user on its retailer. The retailer will forward these invocations to one or more service providers, depending on the number of service providers involved with this request (the end-user can have more than one service session running in one access session, and each of these service sessions can be with a different service provider).

The **resume_session()** and **resume_my_participation()** operations are invoked by the end-user on its retailer for one single service session, and must be forwarded by the retailer to the appropriate service provider, as illustrated.

3.8 Access Session Information Segment

This segment is defined to allow a provider (in the general sense) to inform a user (in the general sense) of changes of state in other access sessions with the same user (for example, access sessions with the same user that are created or deleted). The user is only informed about access sessions it is involved in.

Since this segment only provides simple information in a unidirectional fashion, it does not define any interface but rather defines structures that can be used with either the CORBA **CosNotification** service or the CORBA **CosEvent** service to propagate the information. It is still provided as a separate segment in order to be able to activate or deactivate the access session information by activating or deactivating the segment.

The structures that are defined and their respective usage are listed below.

3.8.1 Access Session Information structures

3.8.1.1 NewAccessSessionInfo

```
struct NewAccessSessionInfo {
        AccessSessionInfo access_session;
};
```

This structure is used to inform the user that a new access session has been established. The **NewAccessSessionInfo** contains the **AccessSessionInfo** structure that contains the following information:

- The **AccessSessionId** of the new access session.
- The corresponding **UserCtxtName** so that the user can identify which domain the access session has been established from.
- The **AccessSessionPropertyList** that is a provider specific property list that can be used to provide more information on the access session.

3.8.1.2 EndAccessSessionInfo

```
struct EndAccessSessionInfo {
        AccessSessionId as_id;
};
```

This structure is used to inform the user that an access session has been ended (by the user). The **AccessSessionId** identifies which access session has ended.

3.8.1.3 CancelAccessSessionInfo

```
struct CancelAccessSessionInfo {
        AccessSessionId as_id;
};
```

This structure is used to inform the user that an access session has been cancelled by the provider. The **AccessSessionId** identifies which access session has been cancelled. This information differs from the **EndAccessSessionInfo** in that an access session is cancelled when the provider invokes the **end_access()** operation on the Access interface of the user, while a normal access session ending is done upon invocation by the user of the **end_access()** operation on the **Access** interface of the provider.

3.8.1.4 NewServicesInfo

```
struct  NewUserServicesInfo {
        ServiceList services;
};
```

This structure is used to inform the user that some new services are immediately available to this user (or subscriber). The **ServiceList** can contains identification of services to which the user has just been subscribed, as well as services that are (momentarily) offered for free (for which no subscription is required). In the case of newly subscribed services, the user may have recently subscribed to the services through a service in this or another access session, or a subscriber may have subscribed his users to a new service. The **ServiceList** is a sequence of **ServiceInfo** structures. The **ServiceInfo** structure contains the **ServiceId**, the **UserServiceName**, and a **ServicePropertyList**.

3.9 Service Session Information Segment

This segment allows a service provider to inform an end-user of changes of state in service sessions in which the end-user is involved. Information will be provided whenever a change to a service session affects the end-user, for example, a service session is suspended, but not when the change does not affect the end-user. Also, the information can be provided for all the service sessions involving the end-user, and not just those associated with this access session. This information helps the access session update the knowledge of the end-user involvement in service sessions.

Since this segment only provides simple information in a unidirectional fashion, it does not define any interface but rather defines structures that can be used with either the CORBA **CosNotification** service or the CORBA **CosEvent** service to propagate the information. It is still provided as a separate segment in order to be able to activate or deactivate the service session information by activating or deactivating the segment.

The structures that are defined and their respective usage are listed below.

3.9.1 Service Session Information Structures

3.9.1.1 NewSessionInfo

```
struct NewSessionInfo {
        SessionInfo session;
};
```

This structure is used to inform the end-user that a new service session has been started. The **SessionInfo** contains information about the new service session that has been started.

3.9.1.2 *EndSessionInfo*

```
struct EndSessionInfo {
        SessionId session_id;
};
```

This structure is used to inform the end-user that a service session has been ended. The **SessionId** identifies the ended service session.

3.9.1.3 *EndMyParticipationInfo*

```
struct EndMyParticipationInfo {
        SessionId session_id;
};
```

This structure is used to inform the end-user that its participation to a service session has been ended. The **SessionId** identifies the service session.

3.9.1.4 *SuspendSessionInfo*

```
struct SuspendSessionInfo {
        SessionId session_id;
};
```

This structure is used to inform the end-user that a service session has been suspended. The **SessionId** identifies the suspended service session.

3.9.1.5 *SuspendMyParticipationInfo*

```
struct  SuspendMyParticipationInfo {
        SessionId session_id;
};
```

This structure is used to inform the end-user that its participation to a service session has been suspended. The **SessionId** identifies the service session.

3.9.1.6 *ResumeSessionInfo*

```
struct  ResumeSessionInfo {
        SessionInfo session;
};
```

This structure is used to inform the end-user that a suspended service session has been resumed. The **SessionInfo** contains information about the suspended service session that has been resumed. The end-user may or may not have re-joined the service session, depending on whether it or another end-user resumed the service session.

3.9.1.7 *ResumeMyParticipationInfo*

```
struct  ResumeMyParticipationInfo {
        SessionInfo session;
};
```

This structure is used to inform the end-user that its (suspended) participation to a service session has been resumed. The **SessionInfo** contains information about the service session to which the end-user's participation has been resumed.

3.9.1.8 *JoinSessionInfo*

```
struct  JoinSessionInfo {
        SessionInfo session;
};
```

This structure is used to inform the end-user that it has joined a service session. The **SessionInfo** contains information about the service session that the end-user has joined.

Subscription Segments 4

Contents

This chapter contains the following sections.

4.1 Overview

As described previously in Chapter 3 the retailer mediates services on behalf of service providers to its end-users. The subscription segments offered by the retailer are structured according to the functionality they provide, (that is, management of entries and query interfaces), and for which roles (see section Section 3.2, “Service Access Segment Interfaces,” on page 3-3) they are used, as depicted in Figure 4-1. The subscription segments define interfaces for the consumer domain to be used by end-users and subscribers, and for the service provider domain to be used by service providers.

Subscription manages information about services and contractual relationships between end-user/subscriber and retailer and between service provider and retailer. Before the subscription segments can be used, the end-user/subscriber or the service provider have to access the retailer by establishing an access session as defined in Chapter 2.

All subscription segments described in this chapter provide a framework for the management of subscription information. The retailer can use them to either build an on-line subscription service or use the interfaces to administer its subscription related information. In general the management tasks for subscription encompass management of:

- subscriber related information concerning create, modify and delete subscriber entries.
- service contracts to create, modify or delete service contracts and assign or de-assign service contracts to users.
- user related information concerning the administration of user entries, user groups (subscription assignment groups) and service profiles.
- service templates to deploy, modify or withdraw a service offered by a service provider.

The subscriber administration segment provides interfaces for the management of subscriber related information and for queries related to the subscriber information. This segment is used by the subscriber.

With the end-user administration segment the subscriber can manage its users and user groups, called subscription assignment groups.

The service provider administration segment provides the management of service templates and is used by the service provider.

The end-user customization segment is used by the end-user to manage its personal preferences to be used for customization.

All subscription segments can only be obtained by using the **get_segment** operation, defined in the core segment (see Chapter 2).

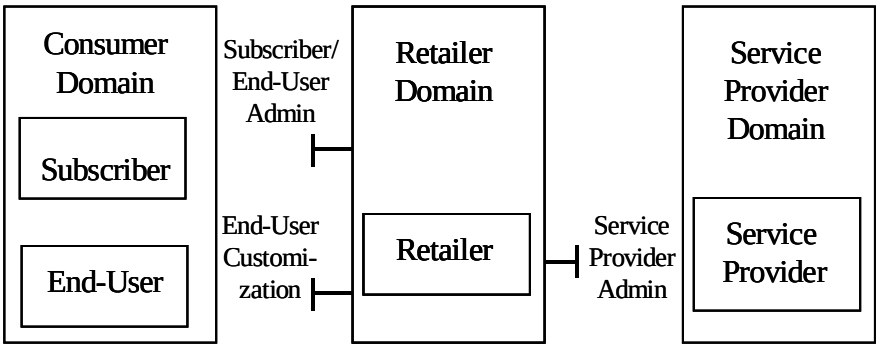

Figure 4-1 subscription Segments

4.2 Information Model

The information model describes the relationship between information objects that are relevant for the retailer to support subscription segments.

Subscribers play an important role by representing organizations or a single end-user which is going to sign *service contracts* for accessing services provided by a retailer. Signing a service contract gives the permission to use a service under the conditions described in the *service profile*.

If the subscriber represents an organization, it can also manage its *end-users* by creating end-user information objects and building groups of these users, called *subscription assignment groups*. The subscriber will authorize its end-users, groups of users or itself to access services for which it has signed the contract by associating *service profiles* with them. Each service contract contains a default service profile for the subscribed service. The subscriber can create new service profiles with respect to its contract and associate these service profiles with its end-users and user groups.

The services a retailer provides to the end-users can be services the retailer offers itself or services the retailer offers on behalf of service providers. In the latter case, the service provider also signs a contract with the retailer and registers its services at the retailer domain by using the service provider administration segments.

A *service type* defines the generic classification or category of a service that will be supported by a retailer. The services are deployed against a particular service type. The service types are created at the retailer before a particular service corresponding to that type can be deployed by a service provider. The management of service types, that is the creation, modification and deletion of service types, is retailer specific. In general, a retailer (administrator) would decide the types of services it wants to host in its domain. The attributes of the service types are defined by properties, which can be different with respect to each of the different service types. The properties are defined

as a list of property name-value pairs. Each attribute has a mode which specifies whether the attribute is mandatory, read-only or normal as defined by Cos trading service.

The *service template* contains information about the service attributes, the environment settings, for example configuration information and references to access the service, and application information such as graphical user interface capabilities, language support.

The end-user service profile is related to service specifics which offer the end-user the ability to change individual attributes of the service (for example, to set personal preferences). The end-user service profile is opaque for the retailer and only passed by to the service provider, which interprets it.

The service template conceives the basis for any contractual relationship between a retailer and a subscriber/end-user and between a retailer and a service provider. The service contract itself restricts the range for service settings defined in the service template. The service settings again can be restricted to a subscription assignment group of one subscriber who defines the individual rights for each user or for the subscription assignment groups by setting attributes in the service profile. In addition, a user can specify individual settings within the range of contractual settings predefined by the subscriber in customizable user profiles.

Figure 4-2 illustrates all relevant information objects using a UML class diagram. The detailed description of these objects and attributes can be found in the text that follows.

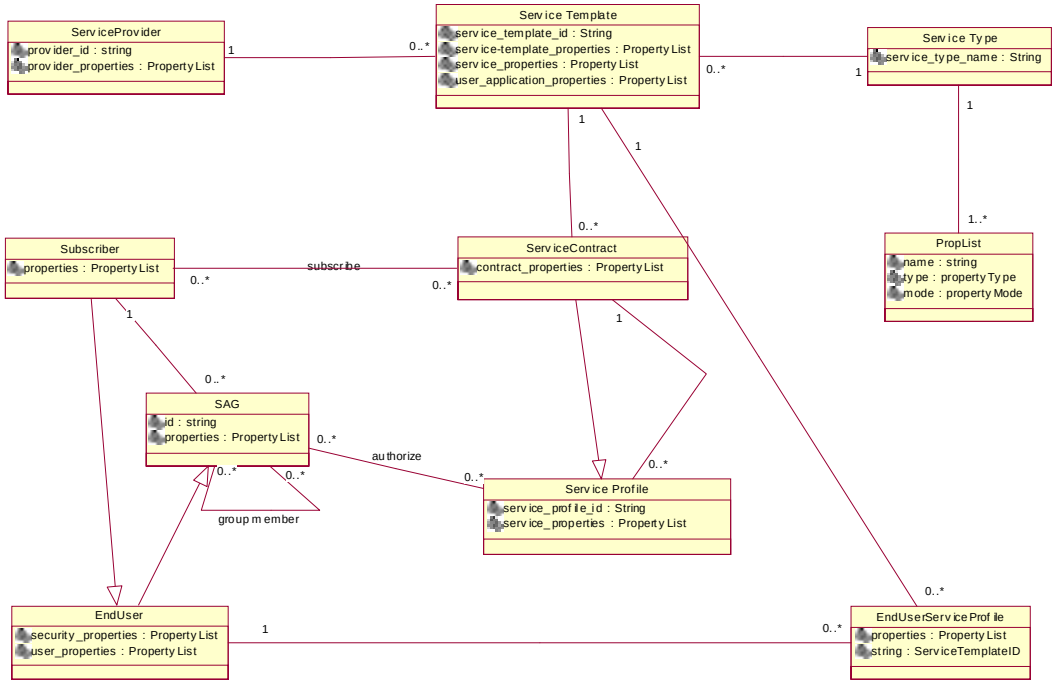

Figure 4-2 Subscription Information Model

4.2.1 Service Provider

```
struct ServiceProvider{
   ProviderId provider_id;
   PropertyList provider_properties;
};
```

The service provider can deploy new service templates of retailer available *service types.* The service provider object contains a service **provider_id** and service *provider_properties* which may contain the address, bank account and other details of the service provider.

4.2.2 Subscriber

```
struct Subscriber {
   SubscriberId subscriber_id;
   PropertyList subscriber_properties;
};
```

A *subscriber* can subscribe to a number of services by signing a *service contract*. The subscriber object contains a subscriber_*id* and *subscriber_properties*.

The following table is a non exhaustive example of valuable subscriber properties. The example uses property types as defined in the COS Trading Service. The property mode specifies whether the property is mandatory, read-only, or normal.

Table 4-1 Subscriber Properties

property type	**name**	**type**	**mode**
	first name	string	normal
	last name	string	normal
	orgname	string	normal
	city	string	normal
	street	string	normal
	postal code	string	normal
	email	string	normal
	payment	string	normal

4.2.3 Service Contract

```
struct ServiceContract{
    ServiceContractId service_contract_id;
    ServiceProfile service_profile;
    PropertyList contract_properties;
};
```

For each subscription a *service contract* exists. The service contract defines the service characteristics for a subscriber and the condition for accessing a service. The service contract properties shall be defined by the retailer. As defined in the access part of the core segment (see Chapter 2) the operation **sign_service_agreement** requires an end-users signature for starting a service. An attribute of the **contract_properties** can be used for the agreement text.

The service contract is a specialization of the *service profile*. Thus it inherits all properties of the service profile which specifies the service characteristics. The **service_contract_id** represents a mapping of the **service_profile_id** (because inheritance of structures is not possible in IDL).

The service contract, which is associated with a *service template*, restricts the usage of a service at subscription time by setting the service properties in the contract. The agreement text indicates that the signature for a service is needed before the service can be used.

4.2.4 Service template

The *service template* defines three kinds of properties:

1. service template properties
2. service properties
3. end-user application properties

```
struct ServiceTemplate{ServiceTemplateId service_template_id;
    ServiceTypeName service_type;
    ProviderId provider_id;
    PropertyList service_template_properties;
    PropertyList service_properties;
    PropertyList user_application_properties;
};
```

The *service_type* defines the category of services a retailer offers.

The *provider_id* identifies the service provider.

Table 4-2 illustrates a non exhaustive example of valuable service template properties. The example properties **remote_provider_id**, **remote_initial_agent_naming_context** and **remote_url** are attributes that can be used to provide a reference to access the service provider domain in order to establish an access as defined in Chapter 2.

The example uses property types as defined in the COS Trading Service. The property mode specifies whether the property is mandatory, read-only or normal.

Table 4-2 Service Template Properties

property structure	**name**	**value**	**mode**
	no_start	bool	normal
	depends_on	string	normal
	config_requirements	string	normal
	autostart	bool	normal
	remote_provider_ID	string	normal
	remote_inital_agent_name_context	string	normal
	remote_service_id	ulong	normal
	remote_userID	string	normal
	remote_password	string	normal
	remote_URL	string	normal

The **user_application_properties** specify the capabilities of the end-user application. A non exhaustive example of user application properties is given in Table 4-3.

Table 4-3 User Application Properties

property structure	**name**	**value**	**mode**
	default_session_context	string	mandatory
	browser	bool	normal
	ORB	string	normal
	java_lib	string	normal
	URL	string	normal
	os	string	normal

4.2.5 Subscription Assignment Group

A subscriber may not want to grant all of its end-users the same service characteristics and usage permissions. In this case he can group them into a *Subscription Assignment Group (SAG)* and than assign *service profiles* to each group. The subscriber can also assign more than one service profile for an end-user, for example an internet travel booking service, where each entry page for flight booking, hotel booking or car reservation can be expressed by a separate service profile. *Subscription Assignment Groups (SAG)* are associated with the subscriber.

```
struct Sag{
    SagId sag_id;
    PropertyList properties;
};
```

The **sag_id** is a string defined by the subscriber to identify its SAGs. The subscriber can describe the SAG using the properties. The **sag_id** together with the **subscriber_id** is unique in the retailer system.

4.2.6 Service Profile

The *service profile* specifies the service settings for the usage of that service. It restricts the service contract settings for a specific end user or SAG. It is associated with the service template, which contains all possible capabilities for service usage defined by the service provider.

```
struct ServiceProfile{
    ServiceProfileId service_profile_id;
    ServiceTypeName service_type;
    ServiceTemplateId service_template_id;
    PropertyList service_properties;
};
```

The *serviceType* is the corresponding classification given by the retailer. The service **template_id** is used as a reference to the service template for the access of a service.

The **service_properties** specify the capabilities of the service.

4.2.7 End-user

An *end-user* will be authorized by a *subscriber* for the access of a service. The *end-user* entry contains an *ID*, *security_properties* and *user_properties* describing the end-user relevant information for subscription. Each *end-user* is part of at least one SAG, which can contain a single user or a group of users.

```
struct EndUser{
   UserID string;
   PropertyList security_properties;
   PropertyList user_properties;
 };
```

The security properties define the kind of authentication a user has, for example password, credential or biometric information. Each end-user can define in the **user_properties** the specific user data such as address, phone number, email..

4.2.8 End-user service profile

The *end-user service profile* defines the a range of end-user specific possible settings for a customized service usage. The end-user service profile is related to the service template, which defines the possible properties for end-user specific settings as part of the service properties.

```
struct EndUserServiceProfile {
   ServiceTemplateId service_template_id;
   PropertyList end_user_service_properties;
 };
```

The **end_user_service_properties** are service dependent and have to be provided by the service provider. The end-user can set its preferences as predefined by the service template which contains the possible range of end-user preferences for a service. The **service_template_id** identifies the **service_template**.

4.2.9 Service type

The *service type* describes the service category (for example, a communication service). For each service type one service template exists, but there might be multiple service templates for one service type. The service type is described using properties similar to those defined in COS Trading service. The properties exist for the service specific characteristics as well as for the preferences which can be set by an end-user.

4.3 Subscription Segments

4.3.1 Overview

The subscription interfaces are only available after successful access to the retailer from either the consumer side to manage subscribers and end-users or from the provider side to manage service templates. If the subscription segments are supported by the retailer, the access of certain services will be controlled by using subscription.

The segments available for the subscription process are:

Subscriber Administration Segment

It allows subscribers to manage their subscription. Four interfaces are provided:

1. **SubscriberMgmt** - this interface is used by the subscriber to create, modify or delete subscriber entries.
2. **SubscriberInfoQuery** - this interface is used to retrieve subscriber related information.
3. **ServiceContractManagement** - this interface allows subscribers to create modify and delete service contracts.
4. **ServiceContractInfoQuery**- this interface allows subscribers to query information about its contracts and to list its subscribed services.

Service Provider Administration Segment

It allows service providers to provide new services in the retailer domain. Two interfaces are provided:

1. **ServiceTemplateMgmt** allows a service provider to deploy, modify, or withdraw a service in the retailer domain.
2. **ServiceTemplateInfoQuery** allows service providers to list all its deployed service templates or get a specific deployed service template.

End-user Administration

- This allows a subscriber to manage its end-users and groups. Four interfaces are provided:

1. **SagMgmt** - this interface is used for the management of end-user groups.
2. **SagInfoQuery** - this interface allows subscribers to get information about existing subscription assignment groups and users.
3. **ServiceProfileMgmt** - this interface allows subscribers to manage service profiles and assign these to subscription assignment groups.
4. **ServiceProfileInfoQuery** - this interface allows subscribers to retrieve information about service profiles and assignment of profiles to subscription assignment groups.

End-user Customization Segment

This allows end-users to customize the service within the range of predefined settings. Two interfaces are defined:

1. **UserProfileMgmt** - this allows an end-user to modify the user profiles and the user service profile settings.
2. **UserProfileInfoQuery** - this allows an end-user to request information about its user profile.

4.4 Scenario Description

To demonstrate the usage of the interfaces the UML sequence diagram in Figure 4-3 provides an example set of interfaces related to roles for which the UML actors are used.

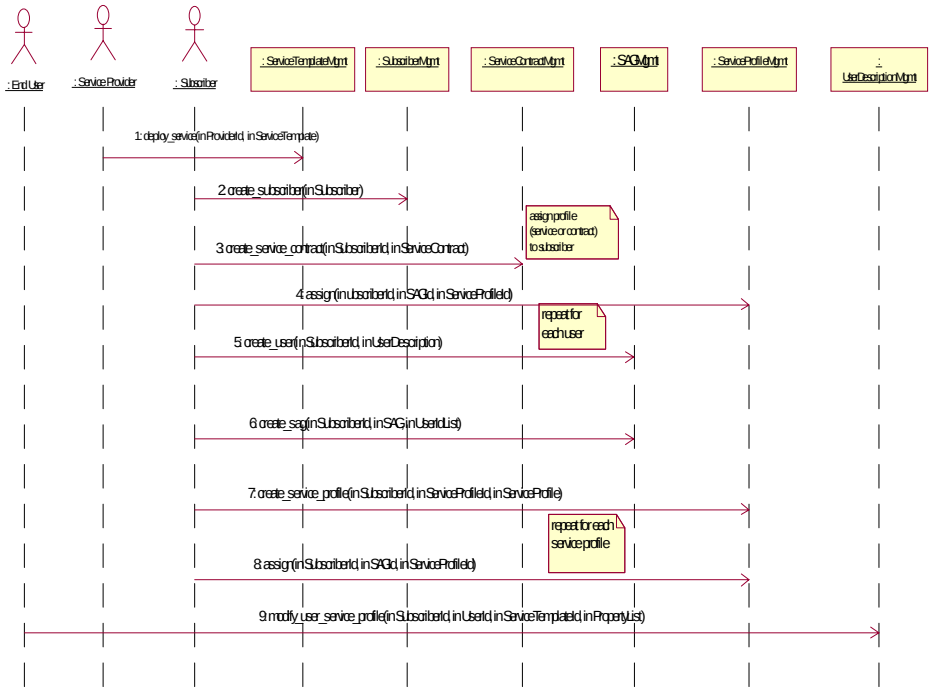

Figure 4-3 Subscription Scenario

1. Prior to any service usage the retailer needs services that can be used. The service provider registers a new instance of a service template to the retailer by using the deploy operation. The service provider provides together with its provider Id the

service template Id which must be unique in the retailer domain. The service provider sets the service properties, which are provided by the retailer to the end-user.

2. A subscriber wants to subscribe to a retailer by using the subscriber administration segment. A new entry will be created for the subscriber. The subscriber can modify or delete its entries or query information about available entries by using the particular operations.

3. After having an account in the system the subscriber sets up a new contract with the retailer and the required settings for the service. Most of the contract information is defined by a property list, which is defined by retailers.

4. If the subscriber is an end-user it assigns a service profile to define the settings to use a service. The default service profile is defined by the service contract. After that the service can be used.

1 to 4 are the necessary steps for simple subscription, where a subscriber is also an end-user. The next steps describe the management for end-users.

5. The subscriber creates for each end-user an entry using its subscriber id and a user id. The subscriber has to create a new entry for each of its users.

6. The subscriber builds a subscription assignment group and can set the properties of that group. The subscriber adds its users (list of user Ids) to the SAG which he has previously created.

7. The subscriber defines the restrictions for its end-users to access and use a service by setting chosen service properties in the service profile.

8. The subscriber assigns the service profiles to SAGs. After that the end-users can use the service.

9. The end-user can now edit its user profile and set its preferences.

4.5 Subscriber Administration

Subscriber administration consists of the management of subscriber entries and the management of service contracts. Subscriber administration is done by the subscriber.

4.5.1 Subscriber Management

The interface **SubscriberMgmt** is used to define new subscriber entries, to modify and to delete them.

4.5.1.1 interface SubscriberMgmt

```
void
create_subscriber(
    in Subscriber  subscriber)
raises (
```

```
    SubscriptionError
);
```

The operation **create_subscriber** allows a subscriber to create a subscriber entry. The subscriber structure contains **subscriber_id** and **subscriber_properties**. The id is given by the subscriber. If the **subscriber_id** already exists, the operation returns an AlreadyExists exception and the subscriber has to try again.

```
void
modify_subscriber(
    in Subscriber  subscriber)
raises (
    SubscriptionError
);
```

The operation **modify_subscriber** modifies subscriber entries, for example a new bank account or a new contact person for billing. The **subscriber_id** is used for identification in the retailer domain. In the case of an invalid **subscriber_id**, the operation returns an InvalidSubscriber exception.

```
void
delete_subscriber(
    in SubscriberId subscriberid)
raises (
    SubscriptionError
);
```

The **delete_subscriber** operation removes a subscriber from the system. The **subscriber_id** is used for identification in the retailer domain. In the case of an invalid subscriber id, the operation returns an InvalidSubscriber exception.

The **SubscriberInfoQuery** interface allows information about existing subscriber to be queried.

4.5.1.2 interface SubscriberInfoQuery

```
void get_subscriber (
    in SubscriberId subscriber_id,
   out Subscriber subscriber)
raises (
    SubscriptionError
);
```

The **get_subscriber** operation returns the information about a subscriber. The **subscriber_id** is used for the identification in the retailer domain. In case of an invalid subscriber id, the operation returns an InvalidSubscriber exception.

4.5.2 Service Contract Management

The ability to create new service contracts, modify these contracts and delete them is given by the **ServiceContractMgmt** interface.

4.5.2.1 interface ServiceContractMgmt

```
void
create_service_contract(
    in SubscriberId subscriber_id,
    in ServiceContract  service_contract)
raises (
    SubscriptionError
);
```

The operation **create_service_contract** is used by the subscriber to provide the contract relevant information. The **subscriber_id** is unique in the retailer domain. The service contract is a structure specifying the **contract_properties** and a default service profile. Exceptions are InvalidSubscriber, InvalidContract.

```
void
modify_service_contract(
    in SubscriberId subscriber_id,
    in ServiceContract  service_contract)
raises (
    SubscriptionError
);
```

The operation **modify_service_contract** is used to modify an existing service contract. The **subscriber_id** identifies the subscriber in the retailer domain and the modifications to the contract will be provided by the subscriber. Exceptions are InvalidSubscriber, InvalidContract.

```
void
delete_service_contract(
    in SubscriberId subscriber_id,
    in ServiceContractId service_contract_id)
raises (
    Subscription Error
);
```

The operation **delete_service_contract** removes an existing service contract. The **subscriber_id** identifies the subscriber in the retailer domain, **service_contract_id** identifies the contract. Exceptions are InvalidSubscriber, InvalidContract.

Information on existing service contracts is provided at the **ServiceContractInfoQuery** interface.

4.5.2.2 *interface ServiceContractInfoQuery*

```
void get_service_contract(
     in SubscriberId subscriber_id,
     in ServiceContractId service_contract_id,
    out ServiceContract service_contract)
raises (
     SubscriptionError
);
```

The Operation **get_service_contract** allows information about a single service contract to be queried and the contract itself to be returned. The **subscriber_id** together with the **contract_id** is unique in the retailer domain. Exceptions are InvalidSubscriber, InvalidContract.

```
void list_subscribed_services(
     in SubscriberId subscriber_id,
    out ServiceProfileIdList service_profile_id_list)
raises (
     SubscriptionError
);
```

The **operation list_subscribed_services** provides a list of all services to which the subscriber has subscribed by a contract. The **subscriber_id** identifies the subscriber in the retailer domain. The operation returns a list of subscribed **service_profile_ids**. Exception is InvalidSubscriber.

4.6 *Service ProviderAdministration*

The service provider administration supports interfaces to service providers to manage the service templates the service provider is going to offer through the retailer domain.

The service template management can be used in order to introduce new services, modify the properties of existing services or delete offered services. The services defined here are actual service offers; for example a video conferencing service of a non-monopolistic telecom operator. The interfaces are used by the service provider.

The service providers can only register those services which are supported by the retailer domain. The retailer itself, in the role of the retailer administrator, decides which kind of services types it supports. Before a service template of a service provider can be registered in the retailer domain, the corresponding service type must be supported. How this is done by the retailer administrator is out of scope of this specification.

At the **ServiceTemplateMgmt** interface the registration, modification and deletion of service templates can be done.

4.6.1 *interface ServiceTemplateMgmt*

```
void deploy_service(
```

```
    in ProviderId provider_id,
    in ServiceTemplate service_template,
   out ServiceTemplateId service_template_id)
raises (
    SubscriptionError
);
```

The operation **deploy_service** allows the service provider to register a new instance of a service template. The operation returns the **service_template_id**, which is used in the retailer domain to identify the service template. The service **provider_id** is given by the service provider and used to identify the provider in the retailer domain. The service template is given by the service provider. The service provider completes the provided **service_properties** and **user_service_properties**, the **service_template_properties** that are needed for the retailer to access the provider domain to start a service and the possible **end-user_application_properties**, which define the conditions for the **user_application** in the consumer domain. Exceptions are InvalidProvider, NotSupportedServiceType and InvalidPropertyList.

How the retailer and service provider exchange the type definitions used in the service template is out of scope of this specification. However, how the retailer defines its own service template that is offered as a service to the end-user is internal to the retailer.

```
void
modify_service(
    in ProviderId provider_id,
    in ServiceTemplate service_template)
raises (
    SubscriptionError
);
```

The operation **modify_service** allows a service provider to modify existing service templates (service offers). The service **provider_id** and the **service_template_id** of the **service_template** struct are used to identify in the retailer domain which service should be modified. The service capabilities are defined by the service provider in the **service _properties**. Exceptions are InvalidProvider and InvalidPropertyList.

```
void
withdraw_service(
    in ProviderId provider_id,
    in ServiceTemplateId service_template_id)
raises (
    SubscriptionError
);
```

The operation withdraw allows a service provider to delete an existing service template. The service **provider_id** and the **service_template_id** are used to identify in the retailer domain which service should be removed. Exceptions are InvalidProvider and InvalidServiceTemplateId.

Information about service templates can be obtained on the **ServiceTemplateInfoQuery** interface.

4.6.1.1 *interface ServiceTemplateInfoQuery*

```
void list_service_templates(
     in ProviderId provider_id,
    out ServiceTemplateIdList service_template_id_llst)
raises (
     SubscriptionError
);
```

The operation list returns a list of all service templates for a service provider. The service **provider_id** is used to identify the service provider in the retailer domain. The operation returns a list of service template ids. Exception is raised for InvalidProvider.

```
void get_service_template(
     in ProviderId provider_id,
     in ServiceTemplateId service_template-id,
    out ServiceTemplate service_template)
raises (
     SubscriptionError
);
```

The operation returns the structure of a single service template. The service **provider_id** and the **service_template_id** are used to identify which service template should be returned. Exceptions are InvalidServiceProvider and InvalidServiceTemplateId.

4.7 *End-user Administration*

The end-user administration interface is intended for situations where an organization wants to allow several end-users to be registered with a retailer. The interfaces are used by the subscriber, who manages the end-users.

When a registered subscriber wants to provide access to a subscribed service for several end-users in the name of its organization, then it has to register them in the retailer domain.

The main task of the end-user administration is to:

- register, modify and delete user entries (*User Management*),
- management of user groups (*SAG Management*),
- management of service profiles which are defined for all subscribed services (*Service profile Management*) and to assign / de-assign the service profiles.

4.7.1 *User and SAG Management*

The **SAGMgmt** interface provides operations to administrate the SAGs of the subscriber.

4.7.1.1 *interface SagMgmt*

```
void
create_sag (
    in SubscriberId subscriber_id,
    in Sag  sag,
    in UserIdList user_ids)
raises (
    SubscriptionError
);
```

For the administration of SAGs the subscriber can use this operation to create a new SAG and to add end-users (which have been created by **create_user**). The **subscriber_id** is used for the identification of the subscriber in the retailer domain. The list of **user_ids** is given by the subscriber. Exceptions are invalidSubscriber, InvalidSag, or InvalidUser.

```
void
modify_sag (
    in SubscriberId subscriber_id,
    in Sag  sag)
raises (
    InvalidSubscriber,
    InvalidSag
);
```

The operation **modify_sag** allows a subscriber to modify an existing SAG. The **subscriber_id** is used in the retailer domain to identify which SAG should be modified. Exceptions are InvalidSubscriber and InvalidSag.

```
void
delete_sag (
    in SubscriberId subscriber_id,
    in SagId sag_id)
raises (
    SubscriptionError
);
```

The operation **delete_sag** allows a subscriber to delete an existing SAG. The **subscriber_id** and **sag_id** are used in the retailer domain to identify which SAG should be removed. Exceptions are InvalidSubscriber and InvalidSag.

```
void
create_user(
    in SubscriberId subscriber_id,
    in  EndUser  end_user)
raises (
    SubscriptionError
);
```

The operation **create_user** creates a new user. The operation is used by the subscriber. The **subscriber _id** together with the **user _id** from the **end_user** structure are unique in the retailer domain. The first entry for an end-user in **end_user** is given by the subscriber, the security properties are as well set by the subscriber, whereby the user properties can be defined by the end-user itself by using the end-user customization segment. Exceptions are InvalidSubscriber and InvalidUser.

```
void
modify_user(
    in SubscriberId subscriber_id,
    in  EndUser  end_user)
raises (
    SubscriptionError
);
```

The operation **modify_user** modifies information for an existing end-user. The operation is used by the subscriber to modify an user entry. The **subscriber_id** and the **user_id** identify for which end-user the modification should be performed. Exceptions are InvalidSubscriber and InvalidUser.

```
void
delete_user(
    in SubscriberId subscriber_id,
    in EndUser end_user)
raises (
    SubscriptionError
);
```

The operations **delete_user** deletes an existing user. The **subscriber_id** and the **user_id** are used to in the retailer domain to identify the user that should be removed. Exceptions are InvalidSubscriber and InvalidUser.

```
void
add_sag_users(
    in SubscriberId subscriber_id,
    in SagId sag_id,
    in UserIdList user_ids)
raises (
    SubscriptionError
);
```

A subscriber can add users to specific SAGs by using the operation **add_sag_users**. Before the subscriber can do that, it must have already created the users with the operation create user. The subscriber can use a list with **user_ids** to add them to the subscription assignment group. Exceptions are InvalidSubscriber, InvalidSag and InvalidUser.

```
void
remove_sag_users(
    in SubscriberId subscriber_id,
    in SagId sag_id,
```

```
    in UserIdList user_ids)
raises (
    SubscriptionError
);
```

The operation **remove_sag_users** removes a single user or a list of users from a SAG of a subscriber. **Subscriber_id** and **user_id** are used to identify in the retailer domain which users should be removed. Exceptions are InvalidSubscriber, InvalidSag, and InvalidUser.

SAGInfoQuery interface is provided to retrieve information about existing SAGs, users and their assignment to SAGs.

4.7.1.2 interface SAGInfoQuery

```
void list_sags(
    in SubscriberId subscriber_id,
   out SagIdList sag_id_list)
raises (
    SubscriptionError
);
```

The operation **list_sags** allows a subscriber to get a list of already created **sag_ids**. The **subscriber_id** is used in the retailer domain to identify the subscriber. The exception InalidSubscriber can be raised.

```
 void get_sag(
    in SubscriberId subscriber_id,
    in SagId sag_id,
   out Sag sag)
raises (
    SubscriptionError
);
```

The operation **get_sag** allows a subscriber to query information about a single SAG. It returns the sag structure containing **sag_id** and properties. **Subscribe_id** and **sag_id** are used to identify which SAG should be provided to the subscriber. Exceptions are InvalidSubscribe and InvalidSag.

```
void get_user(
    in SubscriberId subscriber_id,
    in UserId user_id,
   out EndUser end_user)
raises (
    SubscriptionError
);
```

The operation **get_user** allows a subscriber to query information about a single user. The information about a user contained in the struct **EndUser** is returned. The **subscriber_id** and **user_id** must be unique in the retailer domain. Exceptions are InvalidSubscriber and InvalidUser.

```
void list_sag_users(
     in SubscriberId subscriber_id,
     in SagId sag_id,
    out UserIdList user_id_list)
raises (
     SubscriptionError
);
```

The operation **list_sag_users** allows a subscriber to get a list of all **users_ids** for a single SAG. The **Subscriber_id** and **sag_id** are used in the retailer domain for identification of the SAG. Exceptions are InvalidSubsciber and InvalidUser.

```
void list_users(
     in SubscriberId subscriber_id,
    out UserIdList user_id_list)
raises (
     InvalidSubscriber
);
```

The operation **list_users** returns a list of all **users_ids** of one subscriber. The **subscriber_id** is used to identify the subscriber in the retailer domain.

4.7.2 Service Profile Management

The management of service profiles that are defined for all subscribed services, and the permission for users to use a subscribed service by assigning service profiles to SAGs can be done at the **ServiceProfileMgmt** interface.

4.7.2.1 interface ServiceProfileMgmt

```
void
create_service_profile(
     in SubscriberId subscriber_id,
     in ServiceProfileId service_profile_id,
     in ServiceProfile service_profile)
raises (
     SubscriptionError
);
```

The operation **create_service_profile** allows a subscriber to create a new service profile. The **profile_id** is given by the subscriber. The service profile contains service parameters which may restrict the service usage. The service profile settings depend on the possibilities the service provider allows and are provided as a list of properties. The subscriber can define different service profiles for one service. Exceptions are InvalidSubscriber and InvalidService ProfileId.

```
void
modify_service_profile(
     in SubscriberId subscriber_id,
```

```
    in ServiceProfile service_profile)
raises (
    SubscriptionError
);
```

The operation **modify_service_profiles** allows a subscriber to modify the service properties of an existing service profile. Exceptions are InvalidSubsciber and InvalidServiceProfileId.

```
void
delete_service_profile(
    in SubscriberId subscriber_id,
    in ServiceProfileId service_profile_id)
raises (
    SubscriptionError
);
```

The operation **delete_service_profiles** allows a subscriber to delete an existing service profile. Exceptions are InvalidSubsciber and InvalidServiceProfileId.

```
void
assign(
    in SubscriberId subscriber_id,
    in SagId sagId,
    in ServiceProfileId service_profile_id)
raises (
    SubscriptionError
);
```

The operation **assign** allows a subscriber to assign a service profile to a SAG. The previously created service profile will be assigned to a SAG. Exceptions are InvalidSubsciber, InvalidSag and InvalidServiceProfileId.

```
void
deassign(
    in SubscriberId subscriber_id,
    in SagId sag_id,
    in ServiceProfileId service_profile_id)
raises (
    SubscriptionError
);
};
```

The operation **deassign** allows a subscriber to remove a service profile from a SAG. Exceptions are InvalidSubsciber, InvalidSag, and InvalidProfileId.

The interface **ServiceProfileInfoQuery** is provided to query information about existing service profiles, their states, and their assignments to SAGs and users.

4.7.2.2 *interface ServiceProfileInfoQuery*

```
void list_service_profiles(
     in SubscriberId subscriber_id,
    out ServiceProfileIdList  service_profile_id_list)
raises (
     SubscriptionError
);
```

The operation **list_service_profiles** returns a list of all service profiles ids of a subscriber. Exception is InvalidSubscriber.

```
void list_assigned_service_profiles(
     in SubscriberId subscriber_id,
     in SagId sag_id,
    out ServiceProfileIdList service_profile_id_list)
raises (
     SubscriptionError
);
```

The operation **list_assigned_service_profiles** returns all service profiles assigned to a single SAG. Exceptions are InvalidSubscriber and InvalidSag.

```
void get_service_profile(
     in SubscriberId subscriber_id,
     in ServiceProfileId service_profile_id,
    out ServiceProfile service_profile)
raises (
     SubscriptionError
);
```

The operation **get_service_profile** returns a single service profile. The struct ServiceProfile contains the **profile_id** and the service properties. Exceptions are InvalidSubscriber and InvalidServiceProfileId.

```
void list_assigned_sags(
     in SubscriberId subscriber_id,
     in ServiceProfileId service_profile_id,
    out SagIdList sag_id_list)
raises (
     SubscriptionError
);
```

The operation **list_assigned_sags** returns a list of SAG Ids assigned to single service profile. Exceptions are raised for InvalidSubscriber and invalidServiceProfileID.

```
void list_assigned_users(
     in SubscriberId subscriber_id,
     in ServiceProfileId service_profile_id,
    out UserIdList user_id_list)
```

```
raises (
SubscriptionError
);
```

The operation **list_assigned_users** returns a list of **user_ids** that are assigned to single service profile. Exceptions are InvalidSubscriber and invalidServiceProfileID.

4.8 End-user Customization

The End-user customization segment allows end-users to customize the service in the range of predefined settings.

The interface **UserProfileMgmt** allows an end-user to modify the user profile settings for and the user service profile settings.

The interface **UserProfileInfoQuery** allows an end-user to request information about its user profiles.

4.8.1 interface UserProfileMgmt {

```
void modify_security_properties(
    in SubscriberId subscriber_id,
    in UserId user_id,
    in PropertyList  security_properties)
raises (SubscriptionError);
```

The operation **modify_security_properties** provides the possibility to change for example the user password. The user password can be one attribute of the user properties. The **subscriber_id** and the **user_id** are used to identify the user in the retailer domain. Exceptions are raised for InvalidSubscriber, InvalidProperty and InvalidUserId.

```
void modify_user_profile(
    in SubscriberId subscriber_id,
    in UserId user_id,
    in PropertyList user_properties )
raises (SubscriptionError);
```

The operation **modify_user_profile** allows an end-user to detail its personal entries in the **user_properties**. Subscriber and **user_id** are used to identify in the retailer domain the end-user.

```
void modify_user_service_profile(
    in SubscriberId subscriber_id,
    in UserId user_id,
    in EndUserServiceProfile  end_user_service_profile )
raises (SubscriptionError);
```

The operation **modify _user_service_profile** provides an end-user with the ability to change the personal preferences for the usage of a service predefined by the service provider. The **Subscriber_id** and the **user_id** are used to identify the end-user in the retailer. The **service_template_id** is part of the **end_user_service_profile** structure and is used as the reference for the subscribed service.

```
void delete_user_service_profile(
      in SubscriberId subscriber_id,
      in UserId userId,
      in ServiceTemplateId service_template_id)
raises (SubscriptionError);
```

The operation **delete_user_service_profile** removes the user service profile in the retailer domain. The **subscriber_id** and the **user_id** are used to identify the user in the retailer domain, the **service_template_id** to identify for which service the profile should be deleted. Exceptions are raised for InvalidSubscriberId, InvaliduserID, InvalidServiceTemplateId.

4.8.2 interface UserProfileInfoQuery {

The query interface allows an end-user to question its user descriptions and user service profiles

```
void get_user_description(
      in SubscriberId subscriber_id,
      in UserId user_id
     out EndUser end_user )
raises (SubscriptionError);
```

The operation **get_user_description** provides the end-user with information about its **user_id** and **user_properties**. The **subscriber_id** and the **user_id** are used to identify the end-user in the retailer domain. Exceptions are raised for InvalidSubscriber and InvalidUserId.

```
void list_user_service_profile_ids (
      in SubscriberId subscriber_id,
      in UserId user_id,
     out ServiceTemplateIdList service_template_id_list )
raises (SubscriptionError);
```

The operation **list_service_profiles_ids** returns a list of subscribed end-user service profiles which are associated with the retailer's service **template_ids**. Subscriber and **user_id** are used to identify the user in the retailer domain. Exceptions are raised for InvalidSubscriber and InvalidUserId.

```
void get_user_service_profile(
      in SubscriberId subscriberId,
      in UserId userId,
      in ServiceTemplateId service_template_id,
     out EndUserServiceProfile end_user_service_profile )
```

```
raises (SubscriptionError);
};
```

The operation **get_user_service_profile** returns the end-user service profile which is associated with the retailer's **service_template_ids**. Subscriber and **user_id** are used to identify the user in the retailer domain. The **service_template_id** can be obtain from the operation **list_user_service_profile_ids**. Exceptions are raised for InvalidSubscriber, InvalidServiceTemplateId and InvalidUserId.

Common Types 5

Contents

This chapter contains the following sections.

5.1 Common Information View

This section describes common types of information which have a high potential for re-use (in several segments, or between other domains than the ones described in the TSAS document).

5.1.1 Properties and Property Lists

Properties are attributes or qualities of something. In TSAS, properties are used to assign a quality to something, or search for items or entities that have that particular quality. The entities that can be qualified by such a property for TSAS can be users, providers, services, sessions, interfaces. Each of these will have different properties, and each property may have a range of different values and structures. While some properties will be defined in this document, some supplementary properties can be defined later and eventually be provider specific.

With this in mind, the type **Property** has been chosen to represent a property. Its IDL definition is taken from the CORBA Property Service.

```
typedef string PropertyName;
typedef sequence <PropertyName> PropertyNameList;
typedef any PropertyValue;
struct Property {
        PropertyName  name;
        PropertyValue  value;
};
typedef sequence <Property> PropertyList;
```

As can be seen above, the **Property** is a structure consisting of a name and a value. The name is a string, and the value is an any. This format allows the recipient of the property to read the string and match it against the properties they know about. If it is a property they know, then they will also know the format of the value. If they do not know the property, then they should not read the value. The any value contains a typecode that can be looked up in the interface repository to find the type of the value. The **Property**, and **PropertyList** are used to attribute qualities to entities. Some of these qualities may also be provider-specific, and so they can also use these types to extend the TSAS specifications.

TSAS defines property names and values where it is possible to do so. For some property lists (for example, **InterfaceProperties**) it is up to the user (consumer-/retailer-/service provider domain) to determine properties that can be associated with it.

```
enum WhichProperties {
        NoProperties,
        SomeProperties,
        SomePropertiesNamesOnly,
        AllProperties,
        AllPropertiesNamesOnly
};

struct MatchProperties {
        WhichProperties  which_properties;
        PropertyList  properties;
};
```

MatchProperties is used to scope the return values of some operations. These operations return lists of items. **MatchProperties** is used to identify which items to return, based on the item's properties. For the **operation list_user_services**, the items are a user's subscribed services. The **MatchProperties** parameter defines the properties of the subscribed services that are to be returned in the list.

MatchProperties contains a **PropertyList** and an enumerated type **WhichProperties**. The **PropertyList** contains the properties that need to be matched. The **WhichProperties** identifies whether some, all or none of the properties must be matched, and whether the property name and value, or just the property name must be matched.

For example, in the operation **list_subscribed_services**:

If WhichProperties is...	Then the subscribed services...
NoProperties	don't have to match any property, and consequently all subscribed services are returned.
SomeProperties	must match at least one property in the **PropertyList**, (both the property name and value must match), to be included in the returned list.
SomePropertiesNamesOnly	must match at least one property name in the **PropertyList** to be returned. The values of the properties in the **PropertyList** may not be meaningful and should not be used.
AllProperties	must match all the properties in the **PropertyList**, (both the property name and value must match), to be included in the returned list.
AllPropertiesNamesOnly	must match all the property names in the **PropertyList** to be returned. The values of the properties in the **PropertyList** may not be meaningful, and should not be used.

5.2 *User Information*

```
typedef string UserId;
typedef string UserName;
typedef PropertyList UserPropertyList;
```

The **user_id** (of type **UserId**) identifies the user to the provider. It is unique to this user within the scope of this provider. The **UserId** does not contain the name of the provider, and so cannot be used to contact the provider. It may be sent to a broker/naming service when attempting to contact a provider along with the provider name.

UserPropertyList is a sequence of **UserProperty**. It contains information about the user that needs to be passed to the provider. The following property names are defined for **UserProperty**. Other property names are allowed, but are provider specific.

```
// Property Names defined for UserPropertyList:
// name:    "PASSWORD"
// value:   string
```

```
// use:      user password, as a string.

// name:     "SecurityContext"
// value:    opaque
// use:      to carry a provider specific security context
// e.g.:     could be used for an encoded user password.
```

5.2.1 Usage Related Types

5.2.1.1 SessionId

```
typedef unsigned long SessionId;
```

All the service sessions running in the service provider domain are identified by a **session_id** (of the type **SessionId**). The retailer must translate these **session_ids**, to provide the consumer with a list of **session_ids** that are unique in the consumer domain. The **SessionId** is a long (32 bits). This **session_id** is the same as the **session_id** provided when a service session is started.

5.2.2 Invitations and Announcements

Invitations allow a session to ask a specific consumer to join a 'running' service session. Invitations are delivered by the service provider running the service session to the appropriate retailer, and from that retailer to the consumer domain for the end-user, if an access session exists. If no access session exists with the user domain, the invitation may be delivered using other methods. For example, the invitation may be delivered to a 'pre-registered' interface, or stored by the retailer until an access session is established. Such a 'pre-registered' interface is not defined in this document, but can be defined in a provider specific segment. The invitations contain sufficient information for the invited user to be able to identify the user that issued the invitation, find and join the session, or refuse the invitation.

```
typedef unsigned long InvitationId;
typedef string InvitationReason;

struct InvitationOrigin {
        UserId  user_id;
        SessionId  session_id;
};

struct SessionInvitation {
        InvitationId  id;
        UserId  invitee_id;
        SessionPurpose  purpose;
        ServiceInfo  service_info;
        InvitationReason  reason;
        InvitationOrigin  origin;
        PropertyList  inv_properties;
};
```

```
typedef sequence <SessionInvitation> InvitationList;

enum InvitationReplyCodes {
        SUCCESS, UNSUCCESSFUL, DECLINE, UNKNOWN, ERROR,
        FORBIDDEN, RINGING, TRYING, STORED, REDIRECT, NEGOTIATE,
        BUSY, TIMEOUT
};

typedef PropertyList InvitationReplyPropertyList;

struct InvitationReply {
    InvitationReplyCodes  reply;
    InvitationReplyPropertyListannouncementreplypropertylist  properties;
};
```

SessionInvitation describes the service session to which the end-user (in the consumer domain) has been invited, and provides an **InvitationId** to match this invitation when joining later on. This structure does not give interface references to the session, nor any information that would allow the end-user to join the service session without first having an access session running with this retailer. The structure also provides a **UserId** with the **user_id** of the invited end-user. The consumer domain can check that the invitation is meant for an end-user known to this domain.

SessionPurpose is a string describing the purpose of the service session. A service session purpose may be defined when that service session is started or during the service session.

ServiceInfo is the subscribed service that the end-user can use to join the session.

InvitationReason is a string describing the reason that this invitation was sent to the invited end-user. It can be defined by the end-user that issued the invitation, or by the service session itself.

InvitationOrigin is a structure defining where the invitation has been issued. It contains, for example, the **user_id** of the end-user that started the session.

An **InvitationReply** is returned that allows the end-user to inform the service provider (via the retailer) of the action it will eventually take regarding the invitation. This information is not binding; that is, the end-user can reply that it will join the service session, then take no action, or join later, or not join at all. The following reply codes are defined:

- SUCCESS - the end-user in the consumer domain agrees to join the service session. The use of this reply code should be followed by the end-user taking action to join the service session (see the **join_session_with_invitation()** operation on the **ProviderInvite** interface described in Section 3.3.2, “ProviderInvite Interface,” on page 3-7). However, this reply code can be followed by another reply code (sent with the **reply_to_invitation()** operation that is described in Section 3.3.2, “ProviderInvite Interface,” on page 3-7), in case the end-user changes his mind, but the RINGING reply code should then have been used instead.

- UNSUCCESSFUL - the end-user couldn't be contacted through this operation. However, if the same invitation was sent to multiple interfaces, a reply from another interface may indicate that the end-user will join the session.
- DECLINE - the end-user declines to join the session.
- UNKNOWN - the end-user that is invited is not known by this interface. As was said before, the **SessionInvitation** contains a **UserId** that allows the consumer domain to check if the invitation is meant for an end-user known to this domain.
- FAILED - the end-user is unable to join the service session. No reason is given. The invitation may be badly formatted, or the end-user may be unable to join service sessions.
- FORBIDDEN - the consumer domain is not authorized to accept the request.
- RINGING - the end-user is known by this consumer domain and is being contacted. The service provider should not assume that the end-user will join the session. If the end-user wishes to join the service session, then it can do so as described in SUCCESS above. If it wishes to keep the service provider informed about its status regarding this invitation, it can use the **reply_to_invitation()** operation as described in SUCCESS above.
- TRYING - the end-user is known by this consumer domain, but cannot be contacted directly. The consumer domain is performing some action to attempt to contact the end-user. The service provider can treat this as RINGING.
- STORED - the end-user is known by this domain, but is not being contacted at present. The invitation has been stored though for retrieval by the user later on. The service provider can treat this as RINGING, although it may be a while before the user responds.
- REDIRECT - the end-user is known by this consumer domain, but it is not available through this interface. The service provider should use the address given back in **InvitationReplyProperies** to contact the end-user.
- NEGOTIATE - the end-user is known by this consumer domain, but it is not being contacted at present. The **InvitationReplyProperies** contains a set of alternatives that the service provider could try in order to contact the end-user. These alternatives are not defined by TSAS. They are service provider specific at present.
- BUSY - the end-user cannot be contacted because it is 'busy.' This code should be treated similar to UNSUCCESSFUL.
- TIMEOUT - the end-user cannot be contacted, as the consumer domain has timed out while trying to contact it, (that is, the consumer domain has a time out value for contacting the end-users), for example, pop-up window, ringing phone, and this time has expired. This code should be treated similar to UNSUCCESSFUL.

These invitation reply codes have been taken from the Internet Engineering Task Force working group MMUSIC, (Multimedia Multiparty Session Control) draft standard 'Session Initiation Protocol' (SIP).

Announcements allow a session to publish itself to a 'group' of end-users. The announcements are not directed to a specific end-user, nor are they 'delivered' to any consumer domain. Announcements issued by the service providers are stored by the retailer until the consumer domain requests a list of announcements.

Announcements are returned to the end-user that requested it in the consumer domain, depending upon the ‘groups’ to which the end-user belongs. These are defined by user properties, but no specific mechanism for defining announcement groups is specified by TSAS. Announcements contain sufficient information for the consumer to join the service session.

```
typedef PropertyList AnnouncementProperties;

struct SessionAnnouncement {
        AnnouncementId  announcement_id;
        SessionPurpose  session_purpose;
        ServiceInfo  service_info;
        AnnouncementProperties  properties;
};

typedef sequence <SessionAnnouncement> AnnouncementList;

typedef unsigned long AnnouncementId;
```

SessionAnnouncement describes the session that is being announced, and the ‘group’ of users that the announcement is broadcast to. It is a structure containing the **announcement_id**, the **session_purpose**, the **service_info**, and a list of announcement properties. No property names or values are defined by TSAS for announcements. The announcement properties allow the service providers to define their own types for announcements, which can be passed using the announcement operations defined by TSAS.

AnnouncementId identifies an announcement to the consumer domain. The consumer domain can request a list of announcements that are associated with a specific end-user. The **AnnouncementId** is used by the consumer domain to discriminate between the announcements it receives. The ids for each announcement can only be used by the end-user they are meant for. They do not uniquely identify the announcement across consumer domains.

5.3 Access Session Information

```
typedef unsigned long AccessSessionId;
```

The **accessSessionId** of type **AccessSessionId** is used to identify an access session. The **accessSessionId** corresponding to the end-user’s current access session is returned at the end of the access session set-up phase. The **accessSessionId** for other access sessions can be found **using list_access_sessions()** in the access control segment, on the **AccessControl** interface. The **AccessSessionId** is scoped by the end-user, (that is, for a single end-user (**UserId**) all **AccessSessionIds** are unique).

5.4 User Information

This section describes user related information types more dedicated to the access session, and that have not already been described.

```
struct UserInfo {
        UserId user_id;
        UserName name;
        UserPropertyList user_properties;
};
```

UserInfo describes the end-user. It is a struct of **UserId**, the user's name (that is, readable by a human), and **UserPropertyList**. It is returned by **get_user_info()** on the **ProviderContext** interface.

5.5 User Context Information

The user context information described in this section concerns the user in the generic sense, for example:

- the end-user as a user of the retailer,
- the retailer as a user of the service provider,
- the service provider when it contacts the retailer as a user (for example, for deploying services).

Consequently the user context information can be used in most of the user-provider contexts.

```
typedef string UserCtxtName;

typedef PropertyList UserCtxtPropertyList;

struct UserCtxt {
        UserCtxtName  ctxt_name;
        AccessSessionId  as_id;
        UserCtxtPropertyList  properties;
};

typedef sequence <UserCtxt> UserCtxtList;
```

UserCtxt informs the provider about the user and the user domain, including the name of the context. The user context properties can contain a list of references to supported interfaces.

UserCtxtName is a name given to this user context. It is generated by the user domain. It is used to distinguish between access sessions to different user domains. When listing the access sessions, the **UserCtxtName** is returned, along with the **AccessSessionId**, as the former should be a more human readable name (when end-users are involved).

Properties is a list of user context related properties that might contain; for example, a list of references to supported interfaces.

5.6 Service and Session Information

The service and session information described in this section concerns the user and provider in the generic sense, for example:

- The end-user as a user of the retailer, in its turn provider to the end-user.
- The retailer as a user of the service provider, in its turn provider to the retailer.

```
struct ServiceInfo {
        ServiceId  id;
        UserServiceName  name;
        ServicePropertyList  properties;
};
```

ServiceInfo is a structure that describes a subscribed service of the user.

ServiceId is the identifier for the service. **ServiceId** is unique among all the user's subscribed services. Other users may be subscribed to the same service, but will have a different **ServiceId**. The **ServiceId** value persists for the lifetime of a subscription.

UserServiceName is the name of the service as a string. The name is chosen by the subscriber when it subscribes to the service. It is the name of the service that would ultimately be displayed on the end-user's screen.

ServicePropertyList is a property list, which defines the characteristics of this service. They can be used to search for types of service with the same characteristics, (for example, using **discover_services()** on the **ServiceDiscovery** interface of the service discovery segment).

TSAS has defined no properties for **ServicePropertyList**, and so its use is provider specific.

```
struct SessionInfo {
        SessionId  id;
        SessionPurpose  purpose;
        UserSessionState  state;
        InterfaceList  itfs;
        SessionProperties  properties;
};
```

SessionInfo is a structure that contains information that allows the end-user to refer to a particular service session when using interfaces within an access session. It can also contain information for the usage part of the service session, including the interface references to interact with the service session. The description of these service session interfaces (and their types) is provider specific (outside the scope of TSAS).

Id is the identifier for this service session. It is unique to this service session, among all service sessions that this end-user interacts with through this retailer. If the end-user interacts with multiple retailers concurrently, then they may return **SessionIds** that are identical.

Purpose is a string containing the purpose of the service session. This may have been defined when the service session was created, or subsequently by service specific interactions that are service provider specific.

State is the service session state as perceived by this end-user. It can be: **UserUnknownSessionState**, **UserActiveSession**, **UserSuspendedSession**, **UserSuspendedParticipation**, **UserInvited**, or **UserNotParticipating**.

ltfs is a list of interface types and references supported by the service session. It may include service specific interfaces for the user to interact with the service session. Further details are service provider specific.

Properties is a list of properties of the service session. Its use is service provider specific.

5.6.1 Base Interface

```
interface SegmentBase
{
    void release_segment ( ) ;
};
```

This is the definition of the base interface from which the segment interfaces can inherit in order for all of them to support the **release_segment()** operation.

OMG IDL A

```
#ifndef _DFTSAS_IDL_
#define _DFTSAS_IDL_

#include "CORBA.idl"

module IOP {
    const ServiceId ACCESS_SESSION_ID = OMG_assigned;
    const ServiceId SERVICE_SESSION_ID = OMG_assigned;
};

module DfTsas {
    typedef string SegmentId;
    typedef sequence<SegmentId> SegmentIdList;

    const SegmentId INVITATION_SEGMENT = "Invitation";
    const SegmentId CONTEXT_SEGMENT = "Context";
    const SegmentId ACCESS_CONTROL_SEGMENT = "Access control";
    const SegmentId SERVICE_DISCOVERY_SEGMENT = "Service discovery";
    const SegmentId SESSION_CONTROL_SEGMENT = "Session control";
    const SegmentId SUBSCRIBER_ADMINISTRATION_SEGMENT = "Subscriber administration";
    const SegmentId SERVICE_PROVIDER_ADMINISTRATION_SEGMENT = "Service provider administration";
    const SegmentId END_USER_ADMINISTRATION_SEGMENT = "End user administration";
    const SegmentId END_USER_CUSTOMIZATION_SEGMENT = "End user customization";

    typedef string PropertyName;
    typedef sequence<PropertyName> PropertyNameList;
    typedef any PropertyValue;

    struct Property {
        PropertyName name;
        PropertyValue value;
    };

    typedef sequence<Property> PropertyList;

    enum HowManyProps { none, some, all };
```

```
union SpecifiedProps switch ( HowManyProps) {
      case some: PropertyNameList prop_names;
};

typedef string InterfaceName;
typedef sequence<InterfaceName> InterfaceNameList;

typedef PropertyList InterfacePropertyList;

struct InterfaceStruct {
      InterfaceName name;
      Object ref;
      InterfacePropertyList properties;
};

typedef sequence<InterfaceStruct> InterfaceList;

typedef string  ServiceId;

typedef PropertyList  ServicePropertyList;

typedef string  ServiceToken;

typedef unsigned long  SessionId;
typedef sequence<SessionId> SessionIdList;

typedef string UserSessionState;    // defined values for this type, see doc.

typedef PropertyList SessionPropertyList;

struct SessionInfo {
      SessionId id;
      InterfaceList refs;
      SessionPropertyList properties;
};

typedef PropertyList  EndAccessPropertyList;

enum PropertyErrorCode {
      UnknownPropertyError,
      InvalidProperty,
      UnknownPropertyName,
      InvalidPropertyName,
      InvalidPropertyValue,
      NoPropertyError
};

exception PropertyError {
      PropertyErrorCode error;
      PropertyName name;
      PropertyValue value;
};

enum InterfaceErrorCode {
```

```
        UnknownInterfaceError,
        InvalidInterfaceName,
        InvalidInterfaceRef,
        InvalidInterfaceProperty
};

exception InterfaceError {
        InterfaceErrorCode error;
        InterfaceName name;
        PropertyName property_name;   // if error=InvalidInterfaceProperty, this contains the property in error.
};

enum DomainErrorCode {
        UnknownDomainError,
        InvalidDomainId,
        InvalidDomainRef
};

exception DomainError {
        DomainErrorCode error;
};

enum AuthErrorCode {
        UnknownAuthError,
        InvalidAuthType,
        InvalidAuthCapability,
        NoAcceptableAuthCapability,
        InvalidChallenge
};

exception AuthError {
        AuthErrorCode error;
};

enum AccessErrorCode {
        UnknownAccessError,
        InvalidAccessType,
        InvalidAccessInterface,
        AccessDenied
};

exception AccessError {
        AccessErrorCode error;
};

enum ServiceErrorCode {
        UnknownServiceError,
        InvalidServiceId,
        ServiceAccessDenied,
        InvalidServiceToken
};

exception ServiceError {
        ServiceErrorCode error;
};
```

```
 enum ServiceAgreementErrorCode {
       UnknownServiceAgreementError,
       InvalidServiceAgreementText,
       InvalidSigningAlgorithm
 };

 exception ServiceAgreementError {
       ServiceAgreementErrorCode error;
 };

enum SessionErrorCode {
     UnknownSessionError,
     InvalidSessionId,
     InvalidUserSessionState,
     SessionNotAllowed,
     SessionNotAccepted
};

exception SessionError {
     SessionErrorCode error;
     SessionId session_id;
};

enum SegmentErrorCode {
     InknownSegmentError,
     InvalidSegmentId
};

exception SegmentError {
     SegmentErrorCode error;
     SegmentId segment_id;
};

typedef string UserId;
typedef string UserName;

struct EndUser{
     UserId user_id;
     PropertyList security_properties;
     PropertyList user_properties;
};

enum WhichProperties {
     NoProperties,
     SomeProperties,
     SomePropertiesNamesOnly,
     AllProperties,
     AllPropertiesNamesOnly
};

struct MatchProperties {
     WhichProperties  which_properties;
     PropertyList  properties;
};
```

```
typedef MatchProperties DiscoverServiceProperties;
typedef MatchProperties UserServiceProperties;
typedef MatchProperties SessionSearchProperties;
typedef MatchProperties AnnouncementSearchProperties;

typedef PropertyList UserPropertyList;

typedef unsigned long InvitationId;
typedef string InvitationReason;

struct InvitationOrigin {
    UserId  user_id;
    SessionId  session_id;
};

typedef string SessionPurpose;

typedef string UserServiceName;

struct ServiceInfo {
    ServiceId  id;
    UserServiceName  name;
    ServicePropertyList  properties;
};

typedef sequence<ServiceInfo> ServiceList;

struct SessionInvitation {
    InvitationId  id;
    UserId  invitee_id;
    SessionPurpose  purpose;
    ServiceInfo  service_info;
    InvitationReason  reason;
    InvitationOrigin  origin;
    PropertyList  inv_properties;
};

typedef sequence <SessionInvitation> InvitationList;

enum InvitationReplyCodes {
    SUCCESS, UNSUCCESSFUL, DECLINE, UNKNOWN, ERROR,
    FORBIDDEN, RINGING, TRYING, STORED, REDIRECT, NEGOTIATE,
    BUSY, TIMEOUT
};

typedef PropertyList InvitationReplyPropertyList;

struct InvitationReply {
    InvitationReplyCodes  reply;
    InvitationReplyPropertyList  properties;
};

typedef PropertyList AnnouncementPropertyList;

typedef unsigned long AnnouncementId;
```

```
struct SessionAnnouncement {
      AnnouncementId  announcement_id;
      SessionPurpose  session_purpose;
      ServiceInfo  service_info;
      AnnouncementPropertyList  properties;
};

typedef sequence<SessionAnnouncement> AnnouncementList;

typedef unsigned long AccessSessionId;
typedef sequence<AccessSessionId> AccessSessionIdList;

struct UserInfo {
      UserId user_id;
      UserName name;
      UserPropertyList user_properties;
};

typedef string UserCtxtName;
typedef sequence<UserCtxtName> UserCtxtNameList;

typedef PropertyList UserCtxtPropertyList;

struct UserCtxt {
      UserCtxtName  ctxt_name;
      AccessSessionId  as_id;
      UserCtxtPropertyList  properties;
};

typedef sequence <UserCtxt> UserCtxtList;

typedef PropertyList JoinPropertyList;

typedef sequence<SessionInfo> SessionList;

typedef PropertyList AccessSessionPropertyList;

struct AccessSessionInfo {
      AccessSessionId id;
      UserCtxtName ctxt_name;
      AccessSessionPropertyList properties;
};

typedef sequence<AccessSessionInfo> AccessSessionList;

struct ApplicationInfo {
      string name;
      string version;
      string serial_num;
      string licence_num;
      PropertyList properties;
};

enum WhichAccessSession {
```

```
        CurrentAccessSession,
        SpecifiedAccessSessions,
        AllAccessSessions
};

union SpecifiedAccessSession switch (WhichAccessSession) {
        case SpecifiedAccessSessions: AccessSessionIdList as_id_list;
        case CurrentAccessSession: octet empty_1;
        case AllAccessSessions: octet empty_2;
};

enum WhichUserCtxt {
        CurrentUserCtxt,
        SpecifiedUserCtxts,
        AllUserCtxts
};

union SpecifiedUserCtxt switch (WhichUserCtxt) {
        case SpecifiedUserCtxts: UserCtxtNameList ctxt_names;
        case CurrentUserCtxt: octet empty_1;
        case AllUserCtxts: octet empty_2;
};

enum ListErrorCode {
        ListUnavailable
};

exception ListError {
        ListErrorCode error;
};

enum InvitationErrorCode {
        InvalidInvitationId
};

exception InvitationError {
        InvitationErrorCode errorCode;
};

struct PropertyErrorStruct {
        PropertyErrorCode error;
        PropertyName name;
        PropertyValue value;
};

enum InvitationReplyErrorCode {
        InvalidInvitationReplyCode,
        InvitationReplyPropertyError
};

exception InvitationReplyError {
        InvitationReplyErrorCode error;
        PropertyErrorStruct property_error;
};
```

```
enum AnnouncementErrorCode {
    InvalidAnnouncementId
};

exception AnnouncementError {
    AnnouncementErrorCode error;
};

enum ApplicationInfoErrorCode {
    UnknownAppInfoError,
    InvalidApplication,
    InvalidAppInfo,
    UnknownAppName,
    InvalidAppName,
    UnknownAppVersion,
    InvalidAppVersion,
    InvalidAppSerialNum,
    InvalidAppLicenceNum,
    AppPropertyError,
    AppSessionInterfacesError,
    AppSessionModelsError,
    AppSIDescError
};

exception ApplicationInfoError {
    ApplicationInfoErrorCode error;
    PropertyErrorStruct property_error;
};

enum UserCtxtErrorCode {
    InvalidUserCtxtName,
    InvalidUserAccessIR,
    InvalidUserTerminalIR,
    InvalidUserInviteIR,
    InvalidTerminalId,
    InvalidTerminalType,
    InvalidNAPId,
    InvalidNAPType,
    InvalidTerminalProperty,
    UserCtxtNotAvailable
};

exception UserCtxtError {
    UserCtxtErrorCode error;
    UserCtxtName ctxt_name;
    PropertyErrorStruct property_error;
};

enum SpecifiedAccessSessionErrorCode {
    UnknownSpecifiedAccessSessionError,
    InvalidWhichAccessSession,
    InvalidAccessSessionId
};

exception SpecifiedAccessSessionError {
```

```
        SpecifiedAccessSessionErrorCode error;
        AccessSessionId id;
};

interface SegmentBase {
        void release_segment ( );
};

module Core {
        typedef string  DomainId;

        typedef string    AuthType;         // defined values for this type, see doc.

        struct AuthDomain {
            DomainId domain_id;
            Object ref;
        };

        typedef string    AccessType;       // defined values for this type, see doc.
        typedef string SigningAlgorithm;

        struct SignatureAndSessionInfo {
             string digital_signature;
             SessionInfo session_info;
        };

        typedef string    AuthCapability;         // defined values for this type, see doc.
        typedef sequence<AuthCapability>  AuthCapabilityList;

        interface Initial {
            void initiate_authentication (
                    in AuthDomain user_domain,
                    in AuthType auth_type,
                    out AuthDomain provider_domain
            ) raises (
                    DomainError,
                    AuthError
            );

            void request_access (
                    in AccessType access_type,
                    in Object user_access,
                    out Object provider_access
            ) raises (
                    AccessError
            );
        };

        interface Authentication {
            void select_auth_method (
                    in AuthCapabilityList auth_caps,
                    out AuthCapability selected_cap
            ) raises (
                    AuthError
            );
```

```
    void authenticate (
            in AuthCapability selected_cap,
            in string challenge,
            out string response
    ) raises (
            AuthError
    );

    void abort_authentication ( );
};

interface Access {
    void end_access (
            in EndAccessPropertyList end_access_properties
    ) raises (
            PropertyError
    );

    void list_available_services (
            in UserServiceProperties desired_properties,
              out ServiceList services)
    raises (
            PropertyError,
            ListError
    );

    void select_service (
            in ServiceId service_id,
            in ServicePropertyList service_properties,
            out ServiceToken service_token
    ) raises (
            ServiceError,
            PropertyError
    );

            void start_session (
            in ServiceToken service_token,
            in ApplicationInfo  app,
            out SessionInfo session_info)

    raises (
              ServiceError
    );

    void sign_service_agreement (
                in ServiceToken service_token,
                in string agreement_text,
                in SigningAlgorithm signing_algorithm,
                out SignatureAndSessionInfo signature_session_info
            ) raises (
                ServiceError,
                ServiceAgreementError
            );
```

```
        void end_session (
            in SessionId session_id
        ) raises (
            SessionError
        );

        void list_segments (
            out SegmentIdList segment_ids);

        void get_segment (
            in SegmentId segment_id,
            in InterfaceList user_refs,
            out InterfaceList provider_refs
        ) raises (
            SegmentError,
            InterfaceError
        );

        void release_segments (
            in SegmentIdList segment_ids
        ) raises (
            SegmentError
        );
    };
};

module Sub {
    enum SubExceptionCode{
        subInvalidService,
        subInvalidUser,
        subInvalidSubscriber,
        subInvalidContract,
        subInvalidProvider,
        subInvalidServiceTemplateId
        subNotSubscribed,
        subInvalidSag,
        subInvalidServiceProfileId,
        subInvalidSubscription,
        subNotSupportedServiceType,
        subAlreadyExists,
        subAlreadyAssigned
    } ;

    exception SubscriptionError{
        SubExceptionCode  reason;
    };

    typedef sequence <UserId> UserIdList;
    typedef string  SubscriberId;
    typedef sequence <SubscriberId> SubscriberIdList;
                typedef string  ProviderId;
    typedef string  ServiceTypeName;
    typedef string  ServiceTemplateId;
    typedef sequence <ServiceTemplateId> ServiceTemplateIdList;
    typedef string  ServiceProfileId;
```

```
typedef sequence <ServiceProfileId> ServiceProfileIdList;

struct ServiceProfile{
    ServiceProfileId service_profile_id;
    ServiceTypeName service_type;
    ServiceTemplateId service_template_id;
    PropertyList service_properties;
};

struct ServiceTemplate{
    ServiceTemplateId service_template_id;
    ServiceTypeName service_type;
                    ProviderId provider_id;
    PropertyList service_template_properties;
    ServiceProperties service_properties;
    PropertyList user_application_properties;
};

typedef string ServiceContractId;
struct ServiceContract {
    ServiceContractId service_contract_id;
    PropertyList contract_properties;
    ServiceProfile service_profile;
};

struct Subscriber {
    SubscriberId subscriber_id;
    PropertyList subscriber_properties;
};

struct Provider {
    ProviderId provider_id;
    PropertyList provider_properties;
};

struct EndUser{
    UserId user_id;
                    PropertyListsecurity_properties;
    PropertyList user_properties;
};

        struct EndUserServiceProfile{
    ServiceTemplateId service_template_id;
    PropertyList end_user_service_properties;
};

typedef string SagId;
typedef sequence <SagId>  SagIdList;

struct Sag {
    SagId sag_id;
    string  sag_description;
};

module ServiceProviderAdmin {
```

```
    interface ServiceTemplateMgmt : SegmentBase {
        void deploy_service (
        in ProviderId provider_id,
                    in ServiceTemplate  service_template,
                out ServiceTemplateId service_template_id
        ) raises (
          SubscriptionError
        );

        void modify_service (
                        in ProviderId provider_id,
          in ServiceTemplate  service_template)
          raises (SubscriptionError);

        void withdraw_service (
          in ProviderId provider_id,
                        in ServiceTemplateId service_template_id
        ) raises (
          SubscriptionError
        );
    };

    interface ServiceTemplateInfoQuery : SegmentBase {
        void list_service_templates (
                        in ProviderID provider_id,
           out ServiceTemplateIdList service_template_id_list
        ) raises (
          SubscriptionError
        );

        void get_service_template(
        in ProviderId provider_id,
                    in ServiceTemplateId service_template_id,
           out ServiceTemplate service_template
        ) raises (
          SubscriptionError
        );
    };
};

module SubscriberAdmin {
    interface SubscriberMgmt : SegmentBase {
        void create_subscriber(
          in Subscriber  subscriber
        ) raises (
          SubscriptionError
        );

        void modify_subscriber(
          in Subscriber  subscriber
        ) raises (
          SubscriptionError
        );

        void delete_subscriber(
```

```
            in SubscriberId subscriber_id
        ) raises (
            SubscriptionError
        );


    void get_subscriber (
            in SubscriberId subscriber_id,
                out Subscriber subscriber)
        raises (
            SubscriptionError
        );
    };

    interface ServiceContractMgmt : SegmentBase {
        void create_service_contract(
            in SubscriberId subscriber_id,
            in ServiceContract  service_contract
        ) raises (
            SubscriptionError
        );

        void modify_service_contract(
            in SubscriberId subscriber_id,
            in ServiceContract  service_contract
        ) raises (
            SubscriptionError
        );

        void delete_service_contract(
            in SubscriberId subscriber_id,
            in ServiceContractId service_contract_id
        ) raises (
            SubscriptionError
        );
    };

    interface ServiceContractInfoQuery : SegmentBase {
        void get_service_contract(
            in SubscriberId subscriber_id,
            in ServiceContractId service_contract_id,
                    out ServiceContract service_contract
        ) raises (
            SubscriptionError
        );

        void list_subscribed_services(
            in SubscriberId subscriber_id,
            out ServiceProfileIdList service_profile_id_list
        ) raises (
            SubscriptionError
        );
    };
};
```

```
module EndUserAdmin {
   interface SagMgmt : SegmentBase {
        void create_Sag(
          in SubscriberId subscriber_id,
          in Sag  sag,
          in UserIdList user_ids
        ) raises (
          SubscriptionError
        );

        void modify_Sag(
          in SubscriberId subscriber_id,
          in Sag  sag
        ) raises (
          SubscriptionError
        );

        void delete_Sag(
          in SubscriberId subscriber_id,
          in SagId sag_id
        ) raises (
          SubscriptionError
        );

        void create_user(
          in SubscriberId subscriber_id,
          in EndUser  end_user
        ) raises (
          SubscriptionError
        );

        void modify_user(
          in SubscriberId subscriber_id,
          in EndUser  end_user
        ) raises (
          SubscriptionError
        );

        void delete_user(
          in SubscriberId subscriber_id,
          in UserId user_id
        ) raises (
          SubscriptionError
        );

        void add_Sag_users(
          in SubscriberId subscriber_id,
          in SagId sag_id,
          in UserIdList user_ids
        ) raises (
          SubscriptionError
        );

        void remove_Sag_users(
          in SubscriberId subscriber_id,
```

```
            in SagId sag_id,
            in UserIdList user_ids
        ) raises (
            SubscriptionError
        );
};

interface SagInfoQuery : SegmentBase {
        void list_Sags(
            in SubscriberId subscriber_id,
                    out SagIdList sag_id_list
        ) raises (
            SubscriptionError
        );

        void get_Sag(
            in SubscriberId subscriber_id,
            in SagId sag_id,
            out Sag sag
        ) raises (
            SubscriptionError
        );

        void get_user(
            in SubscriberId subscriber_id,
            in UserId user_id,
            out EndUser end_user
        ) raises (
            SubscriptionError
        );

        void list_sag_users(
            in SubscriberId subscriber_id,
            in SagId sag_id,
            out UserIdList user_id_list
        ) raises (
            SubscriptionError
        );

        void list_users(
            in SubscriberId subscriber_id,
            out UserIdList user_id_list
        ) raises (
            SubscriptionError
        );
};

interface ServiceProfileMgmt : SegmentBase {
        void create_service_profile(
            in SubscriberId subscriber_id,
            in ServiceProfileId service_profile_id,
            in ServiceProfile service_profile
        ) raises (
            SubscriptionError
        );
```

```
        void modify_service_profile(
          in SubscriberId subscriber_id,
          in ServiceProfile service_profile
        ) raises (
          SubscriptionError
        );

        void delete_service_profile(
          in SubscriberId subscriber_id,
          in ServiceProfileId service_profile_id
        ) raises (
          SubscriptionError
        );

        void assign(
          in SubscriberId subscriber_id,
          in SagId sag_id,
          in ServiceProfileId service_profile_id
        ) raises (
          SubscriptionError
        );

        void deassign(
          in SubscriberId subscriber_id,
          in SagId sag_id,
          in ServiceProfileId service_profile_id
        ) raises (
          SubscriptionError
        );
};

interface ServiceProfileInfoQuery : SegmentBase {
        void list_service_profiles(
          in SubscriberId subscriber_id,
          out ServiceProfileIdList service_profile_id_list
        ) raises (
          SubscriptionError
        );

        void list_assigned_service_profiles(
          in SubscriberId subscriber_id,
          in SagId sag_id,
          out ServiceProfileIdList service_profile_id_list
        ) raises (
          SubscriptionError
        );

        void get_service_profile(
          in SubscriberId subscriber_id,
          in ServiceProfileId service_profile_id,
          out ServiceProfile service_profile
        ) raises ( SubscriptionError
        );
```

```
            void list_assigned_Sags(
               in SubscriberId subscriber_id,
               in ServiceProfileId service_profile_id,
               out SagIdList sag_id_list
            ) raises (SubscriptionError
            );

            void list_assigned_users(
               in SubscriberId subscriber_id,
               in ServiceProfileId service_profile_id,
               out UserIdList user_id_list
            ) raises (SubscriptionError
            );
      };
};

module EndUserCustomization {
      interface UserProfileMgmt : SegmentBase {
            void modify_security_properties(
               in SubscriberId subscriber_id,
               in UserId user_id,
               in PropertyList security_properties,
            ) raises (SubscriptionError
            );

            void modify_user_profile(
               in SubscriberId subscriber_id,
               in UserId user_id,
               in PropertyList user_properties
            ) raises (SubscriptionError);

            void modify_user_service_profile(
               in SubscriberId subscriber_id,
               in UserId user_id,
               in EndUserServiceProfile end_user_service_profile
            ) raises (SubscriptionError
            );

            void delete_user_service_profile(
               in SubscriberId subscriber_id,
               in UserId user_id,
               in ServiceTemplateId service_template_id
            ) raises (
               SubscriptionError
            );
      };

      interface UserProfileInfoQuery : SegmentBase {
            void get_user_description(
               in SubscriberId subscriber_id,
               in UserId user_id,
               out EndUser end_user
            ) raises (SubscriptionError);

            void list_user_service_profile_ids (
```

```
                in SubscriberId subscriber_id,
                in UserId user_id,
                out ServiceTemplateIdList service_template_id_list
            ) raises (
                SubscriptionError
            );
            void get_user_service_profile(
                in SubscriberId subscriber_id,
                in UserId user_id,
                in ServiceTemplateId service_template_id,
                out EndUserServiceProfile end_ user_service_profile
            ) raises (
                SubscriptionError
            );
        };
    };
};

module Invitation
{
    interface UserInvite : SegmentBase
    {
        void invite_user (
            in  SessionInvitation invitation,
            out InvitationReply reply
        ) raises (
            InvitationError
        );

        void cancel_invite_user (
            in UserId invitee_id,
            in InvitationId id
        ) raises (
            InvitationError
        );
    };

    interface ProviderInvite : SegmentBase
    {
        void list_session_invitations (
            out InvitationList invitations
        ) raises (
            ListError
        );

        void list_session_announcements (
            in AnnouncementSearchProperties  desired_properties,
            out AnnouncementList  announcements
        ) raises (
            PropertyError,
            ListError
        );

        void join_session_with_invitation (
            in InvitationId  invitation_id,
```

```
                in ApplicationInfo  app,
                in JoinPropertyList  join_properties,
                out SessionInfo  session_info
            ) raises (
                SessionError,
                InvitationError,
                ApplicationInfoError,
                PropertyError
            );

            void join_session_with_announcement (
                in AnnouncementId announcement_id,
                in ApplicationInfo app,
                in JoinPropertyList  join_properties,
                out SessionInfo session_info
            ) raises (
                SessionError,
                AnnouncementError,
                ApplicationInfoError,
                PropertyError
            );

            void reply_to_invitation (
                in InvitationId invitation_id,
                in InvitationReply reply
            ) raises (
                InvitationError,
                InvitationReplyError
            );
        };
};

module Context
{
        interface UserContext : SegmentBase
        {
            void get_user_ctxt(
                out UserCtxt user_ctxt
            );
        };

        interface ProviderContext : SegmentBase
        {
            void set_user_ctxt (
                in UserCtxt user_ctxt
            ) raises (
                UserCtxtError
            );

            void get_user_ctxts (
                in SpecifiedUserCtxt ctxt,
                out UserCtxtList user_ctxts
            ) raises (
                UserCtxtError,
                ListError
```

```
        );

        void get_user_info(
            out UserInfo user_info
        );
    };
};

module AcsCtrl
{
    interface AccessControl : SegmentBase
    {
        void list_access_sessions (
            out AccessSessionList as_list
        ) raises (
            ListError
        );

        void end_access_sessions(
            in SpecifiedAccessSession as
        ) raises (
            SpecifiedAccessSessionError
        );
    };
};

module ServDisc
{
    interface ServiceDiscovery : SegmentBase
    {

        void discover_services(
            in DiscoverServiceProperties  desired_properties,
            in unsigned long  how_many,
            out ServiceList services
        ) raises (
            PropertyError,
            ListError
        );

        void get_service_info (
            in ServiceId  service_id,
            in UserServiceProperties  desired_properties,
            out ServicePropertyList service_properties
        ) raises (
            ServiceError,
            PropertyError
        );
    };
};

module SessCtrl
{
    interface SessionControl : SegmentBase
    {
```

```
            void list_service_sessions (
                  in SpecifiedAccessSession  as,
                  in SessionSearchProperties  desired_properties,
                  out SessionList sessions
            ) raises (
                  SpecifiedAccessSessionError,
                  PropertyError,
                  ListError
            );

            void end_sessions (
                  in SessionIdList session_id_list
            ) raises (
                  SessionError
            );

            void end_my_participations (
                  in SessionIdList session_id_list
            ) raises (
                  SessionError
            );

            void resume_session (
                  in SessionId session_id,
                  in ApplicationInfo app,
                  out SessionInfo session_info
            ) raises (
                  SessionError,
                  ApplicationInfoError
            );

            void resume_my_participation (
                  in SessionId session_id,
                  in ApplicationInfo app,
                  out SessionInfo session_info
            ) raises (
                  SessionError,
                  ApplicationInfoError
            );
      };
};
module AccessSessionInformation
{
      struct newAccessSessionInfo {
         AccessSessionInfo access_session;
      };

      struct endAccessSessionInfo {
         AccessSessionId as_id;
      };

      struct cancelAccessSessionInfo {
         AccessSessionId as_id;
      };
```

```
        struct  newServicesInfo {
            ServiceList services;
        };
    };

    module ServiceSessionInformation
    {
        struct newSessionInfo {
            SessionInfo session;
        };

        struct endSessionInfo {
            SessionId sessionId;
        };

        struct endMyParticipationInfo {
            SessionId sessionId;
        };

        struct suspendSessionInfo {
            SessionId sessionId;
        };

        struct  suspendMyParticipationInfo {
            SessionId sessionId;
        };

        struct  resumeSessionInfo {
            SessionInfo session;
        };

        struct  ResumeMyParticipationInfo {
            SessionInfo session;
        };

        struct  JoinSessionInfo {
            SessionInfo session;
        };
    };
};
#endif // for #ifndef _DFTSAS_IDL_
```

Compliance Points B

This specification does contain elements that are intended to become part of the CORBA standard and, thus, would have to be supported by all CORBA ORBs.

The specification provides three compliance points for implementations of Telecommunication Service Access and Subscription (TSAS), namely, Core Segment, Service Access Segment, and Subscription Segment.

B.1 Core Segment Compliance Point

All conforming implementations must support all interfaces that are defined in Chapter 2 and in document telecom/00-02-03 which contains the IDL specification, following the specified semantics.

B.2 Service Access Segments Compliance Point

An implementation may support any segment defined in Chapter 3, but there is no need to support any of the segments.

When segments are implemented they need to be conformant to the specification given in Chapter 3 and in document telecom/00-02-03 which contains the IDL specification.

B.3 Subscription Segments Compliance Point

An implementation may support any segment defined in Chapter 4, but it is not required to support any of the segments.

When segments are implemented they need to be conformant to the specification given in Chapter 4 and in document telecom/00-02-03 which contains the IDL specification.

B.4 Changes to CORBA

The identification of multiple access sessions introduced in Chapter 2 requires an extension to a very small part of the CORBA architecture. This section details those proposed changes. They are made against CORBA 2.3.1, document formal/99-10-07.

B.4.1 Changes to CORBA Specification

The following service context identifiers are added to the list of service contexts in Section 13.6.7.

```
const ServiceId ACCESS_SESSION_ID = XX; // Reserved for TSAS
const ServiceId SERVICE_SESSION_ID = XX; // Reserved for TSAS
```

The reason for defining these two ServiceIds, is so that a server on which a CORBA invocation is performed, can always retrieve sufficient context information from CORBA so that the client that has performed this invocation is uniquely identified. This has to do with the implementation choices: it is possible that one CORBA server implements more than one access session, or more than one service session. These sessions can potentially be used by different CORBA clients. When a CORBA invocation is made on the CORBA server, it must be able to identify the context (access session identification or service session identification) in which this invocation takes place.

Index