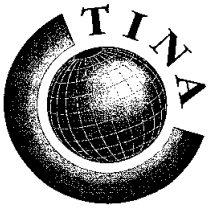


Telecommunications
Information
Networking
Architecture
Consortium

TINA-C Request for Refinements and Solutions

Issue Status: Public Document

Proposal for Unified Response to the RFR/S on Ret

Ret Reference Point Specifications

Version: 1.1

Date of Issue: 30 April 1999

Abstract: This document constitutes the final output of the Evaluation Group of the Request for Refinements and Solutions on the Ret Reference Point, in TINA-C (version 1.0), revised and approved by the SARP-WG (version 1.1) and submitted to the TAB for approval.
It contains the specifications of the Retailer Reference Point (Ret-RP).
It consists of an introduction and specifications of the Access and Usage parts of Ret-RP. Specifications are given in the form of a textual description and of semi-formal specifications of computational interfaces in OMG-IDL language. A complete specification of the IDL interfaces is now delivered separately, in a set of separate IDL files immediately compilable.

Main Authors: **P. Farley (Core Team/BT), S. Hogg (Core Team/Telstra), L. Kristiansen (Core Team/Telenor), C. A. Licciardi (Core Team/CSELT), M. Mampaey (Alcatel), R. Minetti (CSELT), S. Pensivy (Core Team/CNET), C. Smith (BT), R.S. Westerga (Core Team/KPN), M. Yates (Core Team/BT)**

Editors: **P. Farley, R. Minetti, revision editor (v1.1) M. Mampaey**

Group: **Evaluation Group on the Ret-RP, revised by the SARP-WG**

WWW Location: **HTTP://www.tinac.com**

1. Introduction

This document outlines the specifications of the TINA Retailer Reference Point (Ret-RP). It consists of non-formal specifications, in terms of plain text and diagrams, and of semi-formal specifications, using the Object Management Group's Interface Definition Language (OMG-IDL[17]), compliant with TINA-ODL [4].

The purpose is to provide specifications ready to be used for interoperable multi-vendor implementation of computational interfaces at the Ret-RP. As such, the document is a candidate to be the basis for conformance evaluation of TINA-oriented products for the Ret reference point.

The document is the final output of the Evaluation Group for the Request for Refinements and Solutions (RFR/S) on the Ret Reference Point. Specifications contained here are the proposal presented by the Evaluation Group to the Review Panel, within the RFR/S process in TINA-C.

Note: throughout the whole document whenever not otherwise specified, the terminology used follows the definitions given in the TINA Glossary document [2].

1.1 Audience

The primary audience for this document is the Review Panel of the TINA-C RFR/S on Ret-RP [8], and the TINA-C organizations that are responsible for specifications approval.

However, this document is targeted also directly to designers and implementers of TINA products, since it defines the TINA conformance framework, as far as the Ret-RP is concerned.

1.2 How to read this document

The document contains both a description of the reference point and specifications in OMG-IDL language. Since conformance to TINA reference points is stated independently for the access and usage parts, the document treats these parts separately.

Chapter 1 (this Chapter) provides a general introduction to the document.

Chapter 2 refers to the Request for Refinements and Solutions on the Ret-RP [8] giving information concerning specific issues related to the request.

Chapter 3 explains the structure of the Ret-RP, describes the use of the session roles and deals with conformance issues.

Chapter 4 defines the Access Part of Ret-RP. The access part describes how a consumer accesses a retailer to make use of services they provide. The access part addresses the establishment, and use of a secure association between the domains, termed an Access Session [5]. Within the access session, it addresses the control of services, and service sessions. The access part consists of a set of operational interfaces, offered by the consumer and by the retailer business roles. Access interfaces are first defined informally using plain text and diagrams, then by means of semi-formal OMG-IDL specifications; behaviour is described in plain text. Complete specifications of IDL interfaces are given in Annex C and Annex D.

Chapter 5 defines the Usage part of Ret-RP. The usage part of Ret-RP describes the interactions between the consumer and retailer domains during the use of a service session. It defines a set of generic session control operations that allow session components in each domain to interact in a generic manner. The usage part of Ret-RP defines a set of profiles in terms of Session Models. TINA

defines 2 session models, one related to service session control; the other to communication session control. A session model defines feature sets to encompass these generic operations. Each feature set provides some facet of session control, and defines interfaces to make this accessible across an interoperable reference point. They can be combined to provide the specific functionality required by the service. Similar to the access part, usage interfaces are first defined informally using plain text and diagrams, then by means of semi-formal OMG-IDL specifications; behaviour is described in plain text. Complete specifications of IDL interfaces are given in Annex E.

Annex A contains the naming conventions for the IDL modules, and their interfaces and dependencies, contained in the succeeding annexes. It also describes the recorded problems in the process of writing, defining and compilation of the IDLs.

Annex B contains IDL specifications for types which are common to both the access and usage parts of Ret-RP.

Annex C and Annex D contain IDL specifications for types and interfaces for the access part of Ret-RP.

Annex E contains IDL specifications for types and interfaces for the usage part of Ret-RP.

Annex F contains IDL specifications for types and interfaces for Stream Binding and Communication Session, which are part of the usage part of Ret-RP

1.3 Relationship to other TINA-C documents

This document is the solution to the Request for Refinements and Solutions on the Ret Reference Point, Snapshot 1. Therefore, it assumes knowledge of the document defining the Request [8].

This document makes uses of concepts and languages described in the following TINA-C Baseline documents:

- **TINA Reference Points** [1]: this document provides a general framework for the TINA reference points, but also the TINA business model; it defines important modelling concepts that are related to the ODP enterprise viewpoint. Familiarity with the TINA business model and reference points, though not essential, is of great help in understanding the Ret-RP specifications.
- **Computational Modelling Concepts** [4]: this document defines the modelling concepts and conceptual tools for computational modelling, i.e. modelling in the ODP computational viewpoint, in TINA. Since the computational viewpoint is where prescriptive specifications are provided, the understanding and knowledge of the computational modelling concepts is useful for the understanding of the Ret-RP specifications.
- **Information Modelling Concepts** [3]: this document defines the modelling concepts and conceptual tools for information modelling, i.e. modelling in the ODP information viewpoint, in TINA. A certain acquaintance with the scope of information modelling and with OMT notation is necessary to understand the information models presented in this document.
- **TINA Glossary of Terms** [2]: this document lists definitions for all TINA terms.

The TINA architecture gives an essential framework to the reference points specifications. Knowledge of the TINA architecture is essential to use the Ret-RP specifications. The TINA architecture can be partitioned into a service architecture and a network resource architecture. The Ret-RP concerns both partitions, which are defined in the following documents:

- **Service Architecture** [5] & [6]: this document defines the TINA architecture for services. The Ret-RP mainly concerns the service architecture, therefore knowledge of this document is useful in order to understand the framework where the Ret-RP is likely to be deployed. More precisely, the Ret-RP access part, and the service session related portion of the usage part, are related to the Service Architecture.
- **Network Resource Architecture** [9]: this document defines the TINA architecture for network resources. Communication Session aspects of the Ret-RP relate to this part of the TINA architecture.

The components offering the interfaces at the Ret-RP are specified in the following documents:

- **Service Component Specifications** [19]: this document provides the detailed specifications of the components in the Service Architecture, some of which offer interfaces specified for the Ret-RP. TINA ODL is used in the service components specifications document.
- **Network Resource Component Specifications** [20]: this document provides the detailed specifications of the components in the Network Resource Architecture, some of which offer interfaces specified for the Ret-RP. TINA ODL is used in the network resource components specifications document.

1.4 Main inputs to this document

The Ret-RP specifications document relies on the responses to the Request for Refinements and Solutions for the Ret-RP, but adds considerable value to the individual responses presented by TINA-C member companies.

Responses to the RFR/S were presented by the TINA-C Core Team [10] and by Alcatel [11], BT [12], Ericsson [13], France Télécom [14] and Telia [15]; the response presented by Alcatel was produced as part of the TINA-C Auxiliary Project “VITAL” (an ACTS project sponsored by the European Union); the responses presented BT and Ericsson both make use of experiences gained as part of the TINA-C Auxiliary Project “TIMMAP”, and well as other company internal activities. The other responses are based on company-internal activities.

Another important input to the work on Ret-RP is the TINA Service Architecture document [5], defining the architectural framework behind the reference point.

1.5 Overall functionality and scope of the reference point

The Ret-RP defines the interactions between stakeholders in the consumer business role and stakeholders in the retailer business role. It is used to support the consumer’s needs for access to the retailer’s services. In the Consumer/Retailer relationship, the consumer plays the User role and the retailer plays the Provider role. The consumer models two stakeholders: the Subscriber and the End-user; the Subscriber is the entity that has a business relationship with the Provider, whereas the End-user is the person that actually makes use of the capabilities provided by the Retailer.

The Ret-RP is separated in two parts: the Access part and the Usage part.

The Access part of Ret-RP supports a consumer accessing a retailer to make use of services they provide. The access part addresses the establishment, and use of a secure association between the domains, termed an Access Session. Within the access session, it addresses the control of services, and service sessions supports the consumer’s access to a retailer and their services. It corresponds to the functionality, interfaces and objects related to the access session. It defines interfaces to support use of the following functionality:

- initiation of dialogue between the consumer and retailer domains,
- identification of the domains to each other (either domain can remain anonymous dependent on the interaction requested),
- establishment of a secure association between the domains, an access session,
- set up of the default context for the control and management of usage functionality,
- discovery of service[1] offerings,
- listings of access sessions, service sessions and subscribed services
- initiation of usage between the domains,
- control and management of sessions (e.g., stop, suspend, resume, join, notify changes, etc.).

The Usage part of the Ret-RP supports the consumer's use of a retailer's services. It can be split in turn into two parts: the Service part and the Communication part. The Service part corresponds to the functionality, interfaces and objects related to the Service Session; the Communication part corresponds to the functionality, interfaces and objects related to the Communication Session. For more details about the concepts of Access Session, Service Session and Communication Session see [5]. The usage part defines interfaces to support use of the following functionality:

- control and management of sessions (e.g., announce, stop, suspend, invite, notify changes, negotiate transfer of control rights, etc.),
- control and management of stream flow binding,
- domain management (e.g., subscriber management, service management) through management services, that are supported consistently with user services.

The following principles described in the service architecture [5] are used when implementing the Ret-RP reference point requirements:

- Session concept and session graph, providing the definition of the session model and the information structure shared between the parties involved in the service.
- Personal and session mobility, provides the description of how to transfer and manage personal environments between user access points inside a session.
- Management, providing the mechanisms to manage both administrative information (e.g. subscribers) and FCAPS (e.g., fault management for a service).

Due to the complex relationships that can occur amongst consumers, retailers and third party providers, the Ret-RP has the following peculiarity: the usage part (i.e. Service Session and Communication Session) can be used as a reference point between consumer and another stakeholder (retailer, or third party provider) than the retailer to which the consumer is bound for the Ret-RP access part. Figure 1-1 shows the two possible situations: In Case 1, the more straightforward situation, the consumer **A** interacts directly only with retailer **B**, and this interaction takes place on the Ret-RP; on the other hand, interactions between retailer **A** and a third party **C**, which can be a retailer or a Third Party Provider, take place on the RtR or 3Pty reference points, respectively. In Case 2, the consumer performs all access-related interactions across the access part of the Ret-RP with retailer **B**; the actual usage of the service, however, requires a direct interaction between the consumer **A** and the retailer or third party provider **C**, which occurs via the usage part of the Ret-RP between **A** and **C**. Thus, the usage part of Ret-RP must be able to be used indifferently in both cases. Also, no

1. These services can be primary (e.g. Video on Demand (VoD)), ancillary to the primary (e.g. configuration management for VoD) or administrative (e.g., subscriber management for VoD). See [5] for definitions.

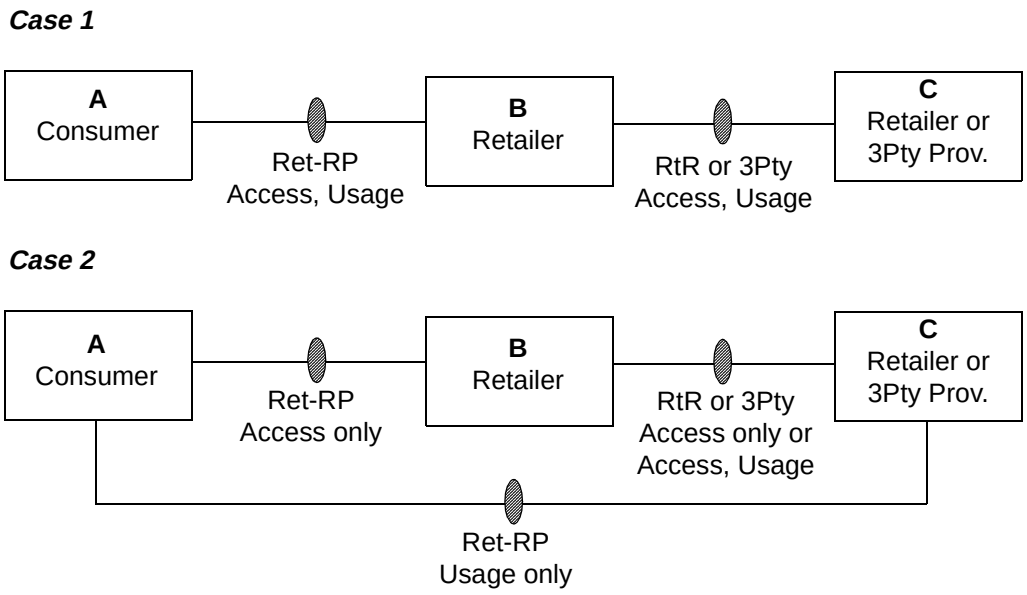


Figure 1-1. Use of the Ret-RP with respect to the TINA business roles.

assumption is made on the RtR and 3Pty reference points. An example of this situation in the case of the TINA object model, with a consumer, a retailer and a third party provider is described in [1] (Section 3.3, Example 2).

1.5.1 Business role life-cycle

The Ret-RP supports the whole lifecycle of the relationship between consumer and retailer, which is described as Subscriber and End-user life-cycle. The Service Architecture [5] describes these life-cycles in detail.

The Subscriber lifecycle describes the processes by which a Subscriber establishes a relationship with a retailer, and modifies or terminates the relationship. The relationship includes subscription, customization, and the association between Subscriber and End-user.

The End-user lifecycle describes the process by which end-users can access and use services.This includes end-user system setup, retailer contact, and service customization.

2. Context of the Answer to the RFR/S on Ret-RP

2.1 Areas of non-compliance

This document complies with the TINA-C Request for Refinements and Solutions, The Ret reference point, Snapshot 1 (Issued in August 1996) [8] and with the TINA Service Architecture Version 5.0 [5].

2.2 Interrelationships with other reference points

The only direct dependency is on the TCon and ConS reference points. These are required to connect streams that are requested in the service layer with stream bindings setup between the TCSM and CSM, in cases where: the Participant-oriented Stream Binding Feature Set is supported in the TINA Service Session Model; or the TINA Communications Session Models is supported, (both defined in the usage part of the Ret-RP).

However, other reference points require similar functionality to that provided across Ret-RP, and will want to make use of similar interfaces. The Ret-RP supports interactions between consumer and retailer domains where these domains act in user and provider roles respectively. These roles are defined where one domain wishes to make use of services offered by the other domain. Many other reference points address a similar situation, such as Retailer-to-Third Party Service Provider (3Pty), Retailer-to-Connectivity Provider (ConS), Consumer-to-Broker (Bkr) and in some cases Retailer-to-Retailer (RtR). It is envisaged that use will be made of the Ret-RP specifications in defining these other reference points. In fact other reference point re-use of Ret-RP specifications can be split according to the access and usage parts of a reference point.

2.3 Main assumptions

Assumptions in this document are the same as in the TINA Service Architecture Version 5.0 [5]. In summary, these are the two main assumptions:

- a pervasive, interoperable, OMG-CORBA based DPE is assumed, providing security services;
- the existence of a naming addressing and resolution framework are assumed.

2.4 Project/prototyping experience

This document is the product of the feedback process from both the TINA-C Core Team and the TINA-C Auxiliary Projects on the prototype implementations concerning various versions of the TINA Service Architecture. In particular, the following prototyping experiences contributed to the specifications in this document:

- The Core Team internal validation work: prototypes in Java and C++ of the consumer and retailer components for the access part of Ret-RP; and SmallTalk language of the CSM/ TCSM interactions;
- The TIMMAP auxiliary project (participants: BT, Ericsson, Iona and TeleDanmark) implementation activity, especially for access interfaces;
- The ACTS-VITAL auxiliary project (participants: Alcatel, CSELT, Iona, Portugal Telecom, Telefónica and other non-TINA-C member companies and institutions) implementation activity, especially for usage interfaces.
- The Global One TINA Technical Trial (participants: Deutsche Telecom AG, France Télécom, and Sprint.)

3. Definition of Ret Reference Point

The reference point definition is a semi-formal specification of the business relationship between the consumer and retailer business roles. Conforming to the TINA Business Model and Reference Points document [7], the Ret-RP is separated into an *Access* part and a *Usage* part. For the Ret-RP, the access part describes how a consumer business role accesses a retailer business role to make use of services it provides; the usage part describes the interactions between the involved business roles during the use of a service. Each part is handled independently.

3.1 Business Roles and Session Roles

As stated in [5], a business role can take different session roles. Three basic session roles are defined: *User*, *Provider* and *Peer*. In the Ret-RP specification only the first two are taken into consideration. A differentiation is made when dealing with access or usage related interactions. So, the roles become access/usage user and access/usage provider. The term *usage party* is used in this document as a synonymous of usage user. The session roles and business domains naming conventions are reflected in the naming of the module structure for the IDL specifications (Annex A).

Although the Ret-RP specifications refer to business roles (in conformance to [7]), the applicability of the specifications themselves can be extended to relationships where the same session roles are involved, irrespective of the business roles involved. For example, whenever an access user and an access provider can be defined, the access part of the Ret-RP specifications can be applied. Similarly, whenever a usage party and usage provider can be defined (with the same semantics as in Ret), the usage part of the Ret-RP specifications can be applied. However, the means to extend Ret-RP specifications to contexts other than the consumer/retailer relationship are outside of the scope of this document.

3.2 Conformance to the Ret Reference Point Specifications

As stated in [7], conformance to a TINA reference points is separate for access and usage part. This means that **a TINA system can be claimed conformant to the access part of the Ret-RP only, to the usage part of the Ret-RP only, to both parts of the Ret-RP, or non conformant to the Ret-RP**.

What criteria are adopted by the owner of the TINA system considered in choosing any of these possibilities depends on the policies of the owner itself, and is outside of the scope of this document. This document provides the necessary guidelines to identify what conformance to Ret-RP means.

Both the access and the usage part of Ret-RP are “profiled”; conformance to each part does not mean support for all features, but means support for all mandatory features[2] in conformance to the TINA specifications, and conformance the TINA specifications for optional features if supported.

The access part is profiled in terms of mandatory and optional interfaces and operations.

The usage part is profiled in terms of Session Models. Conformance to the usage part of Ret-RP is accomplished by conforming to one or more Session Models. TINA defines 2 session models: TINA Service Session Model, and TINA Communication Session Model.

The TINA Service Session Model is related to service session control, i.e. ending a session, inviting participants, managing session stream bindings, etc.

2. Here, feature means interface or operation.

The TINA Communication Session Model is related to communication session control, i.e. set up and controlling stream flow connections that support stream bindings.

Each session may support a number of session models, or may only support a single model. A session may support either the TINA Service Session Model, or the TINA Communication Session Model, or may support both session models. It may also support other session models (not defined by TINA, or Ret-RP) in addition to, or in place of the TINA defined session models. If a session supports either of the TINA session models, then it conforms to the usage part of Ret-RP, even if it also supports other non-TINA session models.

Sessions which do not support either of the TINA session models do not conform to the usage part of Ret-RP.

The TINA session models are profiled into feature sets. A feature set is a 'self-contained' set of interfaces, which provide a set of generic session control operations. Each feature set defines a number of interfaces, which are offered by party and provider domains.

Each TINA session model specify a *mandatory* feature set (which has to be supported by all systems conforming to the session model). Additional feature sets are specified; they do not need to be supported, but systems that conform to the Ret-RP usage part and support feature sets other than the minimum one must comply to the following:

- Support each feature set in an all-or-nothing fashion, unless some feature[2] is explicitly stated as optional.
- Comply to the dependencies among feature sets (e.g. if feature set A depends on feature set B, support either A and B or just B).

Conformance to the access part and usage part of Ret-RP is claimed separately for Consumer side and Retailer side (that is, a TINA system can claim compliance separately for: Access Part - Consumer Side, Access Part - Retailer Side, Usage Part - Consumer Side, Access Part - Retailer Side).

Therefore, for a TINA system, the minimum level of compliance to the access part of the Ret-RP means support at least for the mandatory interfaces and operations of one of the two sides (consumer or retailer). The minimum level of compliance for a TINA system to the usage part of the Ret-RP means support at least for the basic feature set of one of the two sides (consumer or retailer).

In order for two TINA systems to interact via the Ret-RP, it is required that one system conforms to the consumer side and the other to the retailer side, for the access and/or the usage part respectively.

3.3 Common Information View

This section describes the types of information which are common to both the access and usage parts of Ret-RP.

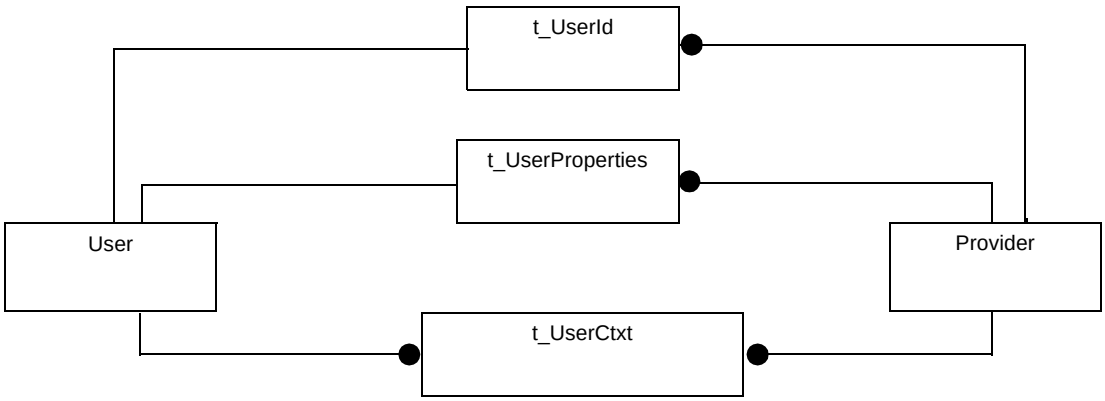


Figure 3-1. Relationship and Cardinality of common types between User and Provider

3.3.1 Properties and Property Lists

Properties are attributes or qualities of something. In Ret-RP, properties are used to assign a quality to something, or search for those somethings that have that particular quality.

The somethings for Ret-RP can be users, services, sessions, interfaces, etc. Each of these will have different properties, and each property may have a range of different values and structures. (Also for some it is now clear what properties will be defined for them, and some properties will be retailer-specific.)

With this in mind, the type `t_Property` has been chosen to represent a property. Its IDL definition is taken from the CORBA Object Service for Trading, and copied into the TINACommonTypes module.

```
// module TINACommonTypes
typedef string t_PropertyName;
typedef sequence<t_PropertyName> t_PropertyNameList;
typedef any t_PropertyValue;
struct t_Property {
      t_PropertyName name;
      t_PropertyValue value;
};
typedef sequence<t_Property> t_PropertyList;

enum t_HowManyProps {none, some, all};
union t_SpecifiedProps switch (t_HowManyProps) {
      case some: t_PropertyNameList prop_names;
      case none:
      case all: octet dummy;
};
```

```
typedef string Istring;
```

As can be seen above, the `t_Property` is a structure consisting of a name, and a value. The name is a international 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, but this should be un-necessary most of the time.)

The `t_Property`, and `t_PropertyList` are used to attribute qualities to entities, when we do not wish to define what all of those qualities are at present. (Some of these qualities may also be retailer-specific, and so they can also use these types to extend the Ret-RP.)

For example, some characteristics about the terminal are sent to the retailer after an access session is established. The type `t_TerminalProperties` is defined as a property list to allow these characteristics to be sent to the retailer. It is not clear precisely what characteristics need to be sent to the retailer for all types of terminal, and some may need to send different information than others.

Ret-RP defines a particular property for `t_TerminalProperties`, named: `"TERMINAL INFO"` which has a value of type `t_TerminalInfo`. `t_TerminalInfo` is a structure that holds some information on terminal characteristics. When the retailer reads the `t_TerminalProperties`, and finds a `t_Property` with the name `"TERMINAL INFO"`, then it can look at the value to find the terminal characteristics. The value will still be of type `any`, but is formatted with the information in the `t_TerminalInfo` structure.

However, `t_TerminalInfo` may not be complete, or relevant for all types of terminal. If it does not contain sufficient information, then a future release of Ret-RP, or a retailer, can define another property, e.g. named `"ADDITIONAL TERMINAL INFO"`, with an appropriate value format, to contain the extra characteristics. The retailer will then receive both properties in the `t_TerminalProperties` list.

If `t_TerminalInfo` contains irrelevant information, a retailer can define an entirely different property, and the consumer should send that instead of the `"TERMINAL INFO"` property.

Ret-RP defines property names and values where it is possible to do so. For some property lists, e.g. `t_InterfaceProperties`, it is up to the consumer/retailer to determine properties that can be associated with it.

```
// module TINACommonTypes
enum t_WhichProperties {
      NoProperties,
      SomeProperties,
      SomePropertiesNamesOnly,
      AllProperties,
      AllPropertiesNamesOnly
};

struct t_MatchProperties {
      t_WhichProperties whichProperties;
      t_PropertyList properties;
};
```

t_MatchProperties is used to scope the return values of some operations. These operations return lists of something. t_MatchProperties is used to identify which somethings to return, based on the something's properties. (e.g. for the operation listSubscribedServices, the something's are a consumer's subscribed services. The t_MatchProperties parameter defines the properties of the subscribed services which are to be returned in the list.)

t_MatchProperties contains a t_PropertyList, and an enumerated type t_WhichProperties. The t_PropertyList contains the properties which need to be matched. The t_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 listSubcribedServices, if t_WhichProperties is:

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

3.3.2 User Information

```
// module TINACommonTypes
typedef Istring t_UserId;
typedef Istring t_UserName;
typedef t_PropertyList t_UserProperties;
```

t_UserId identifies the user to the retailer. It is unique to this user within the scope of this retailer. It is used in requestNamedAccess(), and is returned by getUserInfo(). The t_UserId does NOT contain the name of the retailer, and so cannot be used to contact the retailer. It may be sent to a broker/naming service when attempting to contact a retailer along with the retailer name.

t_UserProperties is a sequence of t_UserProperty. It contains information about the user, that they wish to pass to the retailer. The following property names are defined for t_UserProperty. Other property names are allowed, but are retailer specific.

```
// Property Names defined for t_UserProperties:
// name:       "PASSWORD"
// value:      string
// use:        user password, as a string.

// name:       "SecurityContext"
// value:      opaque
// use:        to carry a retailer specific security context
```

```
//              e.g. could be used for an encoded user password.
```

3.3.2.1 e_UserDetailsError Exception

```
// module TINACommonTypes
enum t_UserDetailsErrorCode {
      InvalidUserName,
      InvalidUserProperty
};

exception e_UserDetailsError {
      t_UserDetailsErrorCode errorCode;
      t_UserName name;
      t_PropertyErrorStruct propertyError;
};
```

The e_UserDetailsError exception is defined for operations which require a t_UserDetails parameter. (e.g. inviteUserReq() on usage part of Ret-RP). The exception is raised if the t_UserName, or the t_UserProperties are invalid.

The following error codes can be used to define the problem encountered:

- InvalidUserName:
  The t_UserName parameter does not contain a valid party identifier. (This can be because the t_UserName is wrongly formatted, or the t_UserName given does not refer to any known user.)

  The t_UserName name variable in the exception contains the value of the t_UserName parameter passed in the operation invocation.
- InvalidUserProperty:
  The t_UserProperties parameter is in error.

  The propertyError element of the exception describes the type of error in the user property.

  If the propertyError contains InvalidPropertyName, then the property name is not legal for this operation. If it contains InvalidPropertyValue, then the value is not a legal value for the property name.

  If the propertyError contains UnknownPropertyName, then the session does not recogise the property name. Some sessions may ignore t_PropertyName's that they do not recognise. They should not process t_PropertyValue associated with the t_PropertyName but may process the other t_Property's in the t_UserProperties parameter. Such sessions do not need to raise the exception with this error code.

3.3.3 User Context Information

```
// module TINACommonTypes
typedef Istring t_UserCtxtName;


// module TINAProviderAccess
struct t_UserCtxt {
```

```
        TINACommonTypes::t_UserCtxtName ctxtName;
        TINAAccessCommonTypes::t_AccessSessionId asId;
        Object accessIR;            // type: i_UserAccess
        Object terminalIR;          // type: i_UserTerminal
        Object inviteIR;            // type: i_UserInvite
        Object sessionInfoIR;       // type: i_UserSessionInfo
        TINAAccessCommonTypes::t_TerminalConfig terminalConfig;
    };
```

t_UserCtxt informs the retailer about the consumer domain, including the name of the context, interfaces available during this access session, and terminal configuration information The t_UserCtxt is only used within the access part of Ret-RP, but is included here to aid the read in understanding the t_UserCtxtName. A full description is given in Section 4.3.3.

t_UserCtxtName is a name given to this consumer context. It is generated by the consumer domain. It is used to distinguish between access sessions to different consumer domains/terminals. When listing the access sessions, the t_UserCtxtName is returned, (along with the t_AccessSessionId), as the former should be a more human readable name for the 'terminal' that the access session is connected to.!!!

3.3.4 Usage related types.

3.3.4.1 t_SessionId

```
// module TINACommonTypes
typedef unsigned long t_SessionId;
```

All operations on the party domain interfaces, (incl all Exe's and Info's) include a t_SessionId parameter.

This allows the party domain to identify the service session sending each operation request. It is a long (32 bits). The t_SessionId is the same as the sessionId provided by the startService(), or joinSession() operation for this session. (ie. the id appearing on the listSessions list in the access part matching this t_SessionId will refer to the same session.) If the party domain does not recognise the t_SessionId, it may raise a PD_InvalidSessionId error code in the e_PartyDomainError exception.

3.3.4.2 t_ParticipantSecretId

```
// module TINACommonTypes
typedef sequence<octet, 16> t_ParticipantSecretId;
```

All operations on the provider domain interfaces of the service session, (incl. all Requests) include a t_ParticipantSecretId parameter. This type is also returned by requests to start, and join a service session.

This allows a service session to identify the sender of each operation request. It is a 128-bit key. The format of the key is not defined, other than all zero's assumes the participant does not know or does not require a key. The session may raise an InvalidParticipantSecretId error code in the e_UsageError exception, if a key is necessary to make a request.

The t_ParticipantSecretId is provided so that sessions can be implemented using only a single interface for all the participants. The session can still be reasonably assured that the request comes from the identified user, and not a different user.

It is not intended that the t_ParticipantSecretId is used as the primary security mechanism. CORBA security, or other security contexts should still be used to underly the party domain-provider domain interactions.

3.3.5 Invitations and Announcements

Invitations allow a session to ask a specific end-user to join a 'running' session. Invitations are delivered to the consumer domain for the end-user, if an access session exists. If no access session exists with the consumer domain, the invitation may be delivered to a 'pre-registered' interface, or stored until an access session is established. They contain sufficient information for the user to: identify the user that requested the invitation be sent; find and join the session, or refuse. (All of these operations are defined across the access part of Ret-RP, and the retailer is always involved in allowing the consumer to find and join the session.)

```
// module TINAAccessCommonTypes
typedef unsigned long t_InvitationId;
typedef TINACommonTypes::Istring t_InvitationReason;

struct t_InvitationOrigin {
      TINACommonTypes::t_UserId userId;
      TINACommonTypes::t_SessionId sessionId;
};

struct t_SessionInvitation {
      t_InvitationId id;
      TINACommonTypes::t_UserId inviteeId;
      t_SessionPurpose purpose;
      t_ServiceInfo serviceInfo;
      t_InvitationReason reason;
      t_InvitationOrigin origin;
};

typedef sequence<t_SessionInvitation> t_InvitationList;


// module TINACommonTypes
enum t_InvitationReplyCodes {
      SUCCESS, UNSUCCESSFUL, DECLINE, UNKNOWN, ERROR, FORBIDDEN,
      RINGING, TRYING, STORED, REDIRECT, NEGOTIATE, BUSY, TIMEOUT
};

typedef t_PropertyList t_InvitationReplyProperties;

struct t_InvitationReply {
      t_InvitationReplyCodes reply;
      t_InvitationReplyProperties properties;
};
```

t_SessionInvitation describes the service session to which the consumer has been invited, and provides an t_InvitationId to identify this invitation when joining. (It does not give interface references to the session, nor any information which would allow the consumer to join the session

without first establishing an access session with this retailer.) It also provides a `t_UserId` with the id of the invited user. The consumer domain can check that the invitation is for an 'end-user' that is known to this domain.

`t_SessionPurpose` is a string describing the purpose of the session. A session purpose may be defined when the session is started (through `t_StartServiceSSProperties`), or during the session.

`t_ServiceInfo` is the subscribed service that the consumer can use to join the session. It is described in Section 4.3.4.

`t_InvitationReason` is a string describing the reason that this invitation was sent to the invited user. It can be defined by the party which requested the invitation, or by the session.

`t_InvitationOrigin` is a structure defining where the invitation has been generated. It contains the userId of the user that started the session, and their session id for the session.

An `t_InvitationReply` is returned which allows the consumer to inform the retailer of the action they will take regarding the invitation. The following reply codes are defined:

- `SUCCESS` - the consumer agrees to join the service session. (The consumer will need to establish an access session before they can join the service session. (This does NOT have to be establish from the terminal that received the invitation.) They will then use `joinSessionWithInvitation`() on the `i_RetailerNamedAccess` interface to join the session.) The consumer can use `replyToInvitation`() to 'change their mind', and not join the session, but they should really have replied with `RINGING`, or another reply code rather than `SUCCESS`.
- `UNSUCCESSFUL` - the consumer couldn't be contacted through this operation. (They will not be joining the session due to this invitation. However, if the same invitation was sent to multiple interfaces, a reply from another interface may indicate that the consumer will join the session.)
- `DECLINE` - the consumer declines to join the session.
- `UNKNOWN` - the consumer that has been sent the invitation is not known by this interface. (The `t_SessionInvitation` contains a `t_UserId` to allow the consumer domain to check the invitation is for a user known to this domain.)
- `FAILED` - the consumer is unable to join the service session. (No reason is given. The invitation may be badly formatted, or the consumer may be unable to join sessions.)
- `FORBIDDEN` - the consumer domain is not authorised to accept the request.
- `RINGING` - the consumer is known by this domain and is being contacted. The retailer should not assume that the consumer will join the session. (If the consumer wishes to join the session then they can do so as describe in `SUCCESS` above. If they wish to inform the retailer about their status regarding this invitation, they can use `replyToInvitation`() on the `i_RetailerNamedAccess` interface.)
- `TRYING` - the consumer is known by this domain, but cannot be contacted directly. The consumer domain is performing some action to attempt to contact the consumer. The retailer can treat this as `RINGING`.
- `STORED` - the consumer is known by this domain, but is not being contacted at present. The invitation has been stored for retrieval by the consumer. (The retailer can treat this as `RINGING`, although it may be awhile before the consumer responds.)

- `REDIRECT` - the consumer is known by this domain, but they are not available through this interface. The retailer should use the address given in `t_InvitationReplyProperies`, to contact the consumer.
- `NEGOTIATE` - the consumer is known by this domain, but they are not being contacted at present. The `t_InvitationReplyProperies` contains a set of alternatives that the retailer could try in order to contact the consumer. (These alternative are not defined by Ret RP, and so are retailer specific at present.)
- `BUSY` - the consumer cannot be contacted because they are 'busy'. This code should be treated as for `UNSUCCESSFUL`.
- `TIMEOUT` - the consumer cannot be contacted, as the consumer domain has timed out while trying to contact them. i.e. the consumer domain has a time out value for contacting the consumer using the method for contacting them, (e.g. pop-up window, ringing phone.), and this time has expired. This code should be treated as for `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'.

Announcements allow a session to publicise itself to a 'group' of end-users. The announcements are not directed to a specific user, nor are they 'delivered' to the end-user. Announcements are stored by the retailer domain until the consumer domain requests for a list of announcements. Announcements are return to the consumer, depending upon the 'groups' to which the user belongs. (These are defined by user properties, but no specific mechanism for defining announcement groups has been specified by Ret-RP. Announcements contain sufficient information for the user to join the session. (This operation is defined across the access part of Ret-RP, and the retailer is always involved in allowing the consumer to find and join the session.)

Draft definition: The structure for announcements is draft only. We would be glad to receive any feedback concerning this structure and its semantics.

```
// module TINACommonTypes
typedef t_PropertyList t_AnnouncementProperties;

struct t_SessionAnnouncement {
      t_AnnouncementId announcementId;
      t_SessionPurpose sessionPurpose;
      t_ServiceInfo serviceInfo;
      t_AnnouncementProperties properties;
};

typedef sequence<t_SessionAnnouncement> t_AnnouncementList;


// module TINAAccessCommonTypes
typedef unsigned long t_AnnouncementId;
```

`t_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 announcementId, the sessionPurpose, the serviceInfo, and a list of announcement properties. No property names or values are defined by Ret-RP for announcements. The announcement properties allow the retailers to define

their own types for announcements, which can be passed using the announcement operations defined by Ret-RP. It is possible that the structure of announcements will change in future versions of Ret-RP.

`t_AnnouncementId` identifies an announcement to the consumer domain. The consumer domain can request a list of announcements which are associated with this end-user. The `t_AnnouncementId` is used by the consumer domain to distinguish between the announcements it receives. The ids for each announcement can only be used by this user. They do not uniquely identify the announcement for all consumers of a retailer.

4. Access Part

This section contains a definition and explanation of the access part of Ret-RP. The access part describes how a consumer accesses a retailer to make use of services it provides.

The access part of Ret-RP offers the following capabilities:

- initiation of dialogue between the consumer and retailer domains,
- identification of the domains to each other (either domain can remain anonymous dependent on the interaction requested),
- establishment of a secure association between the domains, (an access session),
- set up of the default context for the control and management of usage functionality (service sessions),
- discovery of service[1] offerings,
- listings of access sessions, service sessions and subscribed services
- initiation of usage between the domains, (starting a service session),
- control and management of service sessions (e.g., stop, suspend, resume, join, notify changes, etc.).

The access part largely addresses the establishment, and use of a secure association between the domains, termed an Access Session [5].

The access part of Ret-RP is defined in the following sections. Many of the interfaces and operations defined for Ret-RP will be applicable to the access parts of other inter-domain reference points (such as 3PTY, and RtR). In order to facilitate this re-use, a set of interfaces have been defined which can be re-used in other reference points. These interfaces can be recognized by the prefix `i_User` or `i_Provider`. These interface types correspond to the Access User and Access Provider roles identified in the Service Architecture [5].

Interfaces for the Ret-RP are designated with the prefixes: `i_Consumer` and `i_Retailer`. These correspond to the consumer and retailer business administrative domains from the TINA Business Model [7]. For Ret-RP, these domains take the access user and access provider roles. All of the `i_Consumer` and `i_Retailer` interfaces are inherited from corresponding `i_User` and `i_Provider` interfaces. Any specialisations for the Ret-RP are defined in the `i_Consumer` / `i_Retailer` interfaces. However, no specialisations are defined at present.

In summary, the Ret-RP interfaces are inherited from generic user-provider interfaces that can be reused in many other reference points.

NOTE: The main body of this document describes only interfaces and operations on interfaces. A complete listing of the IDLs, and how the interfaces are grouped into modules can be found in Annex A - Annex F:.

The remainder of the Access Part of Ret-RP is structured as follows:

1. These services can be primary (e.g. Video on Demand (VoD)), ancillary to the primary (e.g. configuration management for VoD) or administrative (e.g., subscriber management for VoD). See [2] for definitions.

Section 4.1, "Overview of Access interfaces for Ret-RP" contains a description of the access interfaces of Ret-RP, together with a short explanation of every operation. It identifies only those interfaces which are exported over Ret-RP.

Section 4.2, "User-Provider Interfaces" identifies the generic user-provider interfaces that are not exported over Ret-RP. It shows the inheritance hierarchy for the interfaces exported over Ret-RP. It also describes the generic interfaces so that they can be re-used for other inter-domain reference points.

Section 4.3, "Access Information View" gives an information view of Ret-RP. It describes the types of information passed over the Ret-RP.

Section 4.4, "Access Interface definitions" describes the operations of Ret-RP interfaces supported by the consumer and retailer domains in detail, including parameters and dynamics.

IDL definitions of each of the interfaces can be found in Annex C.

4.1 Overview of Access interfaces for Ret-RP

The Access Part of the Ret-RP is defined by a set of interfaces which are offered over the reference point. All of the interfaces in the Access Part are categorised by which side of the RP offers the interface: the consumer, or the retailer.

The interfaces are also categorised by whether they are assessable during an access session; always available outside of an access session; or may be registered to be available outside of an access session.

Registration of interfaces can only be done by the consumer on the retailer domain during an access session. The lifetime of registration depends on how the consumer registers his interfaces, i.e. only as long as the access session exists or permanent.

The following diagram names all of the interfaces defined by the Access Part, and categorizes them as above.

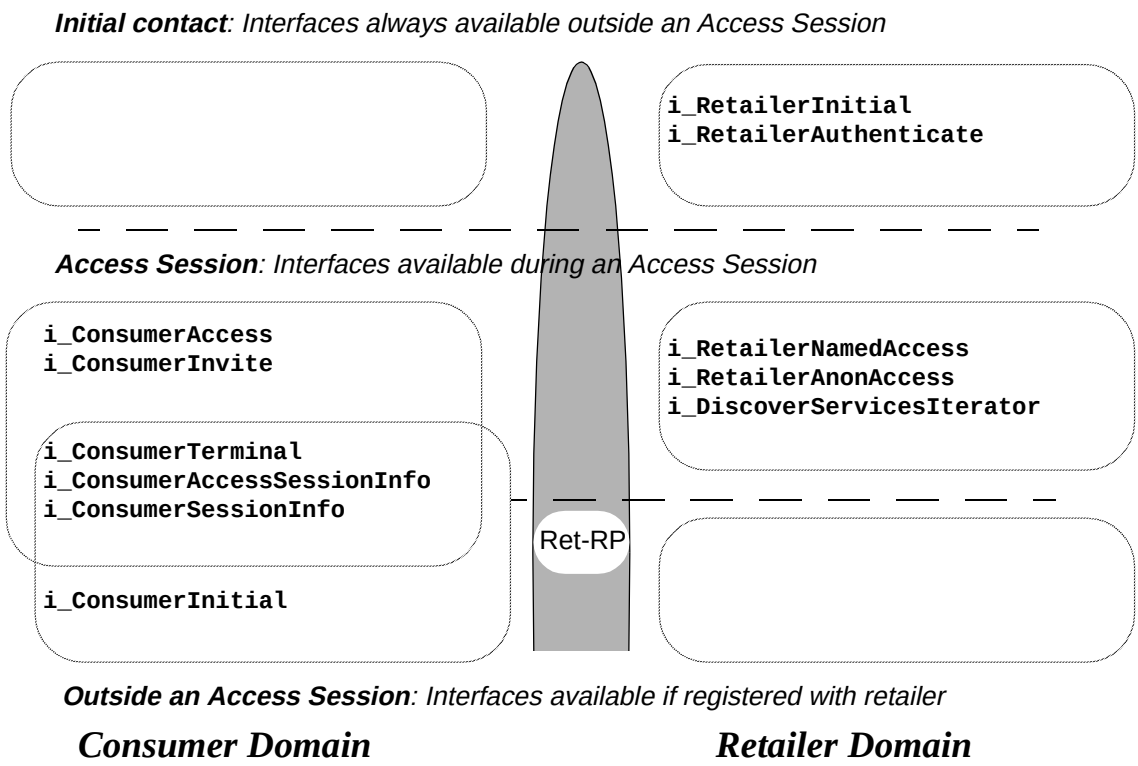


Figure 4-1. Interfaces in Access Part of Ret-RP.

The interfaces: “always available outside of an access session” are supported by the retailer to allow a consumer to request the establishment of an access session. They are the initial point of contact for the consumer, and allow him to authenticate himself and the retailer; establish the access session; and gain a reference to a `i_RetailerNamedAccess`, or `i_RetailerAnonAccess` interface.

The interfaces: “available during an access session” allow the consumer and retailer to interact during an access session. The interfaces on the retailer allow the consumer to discover services; initiate usage of those services; control and manage those services, (e.g. stop, suspend, resume, etc.) and register the consumer’s context and interfaces with the retailer. The interfaces supported by the consumer allow the retailer to discover interfaces and terminal configuration of the consumer; notify changes in access and service sessions; and send invitations to join service sessions. These capabilities are only possible during the access session.

The interfaces: “available if registered with retailer” are supported by the consumer. They must be registered with the retailer for use outside the access session to be accessible. With the appropriate interface registered, the retailer is able to perform all of the operations “available during an access session”, as well as request the consumer to initiate an access session with them.

The following sections will globally describe the interfaces and their operations. Detailed information about operations, their parameter lists and dynamics can be found in Section 4.4, "Access Interface definitions".

4.1.1 Example Scenario of Access part of Ret-RP

This section is an example of the use of the access Ret-RP interfaces. It describes a consumer making use of retailer interfaces to establish an access session; make use of retailer facilities, and register to receive invitations outside of an access session.

1. A consumer domain contacts the retailer by gaining a reference to a **i_RetailerInitial** interface[2].

2. The consumer domain calls the **requestNamedAccess**() operation on **i_RetailerInitial**, as he wishes to establish an access session with the retailer as a named user. (If the consumer wished to remain anonymous, he could use the **requestAnonymousAccess**() operation on that interface instead.)

   2.a If CORBA security services have been used, then both domain's credentials and other authentication information will have been exchanged, and both consumer and retailer will have been authenticated to each other. The call to **requestNamedAccess**() returns a reference to a **i_RetailerNamedAccess** interface. (An access session has been established between the consumer and retailer domains.)

   2.b If CORBA security services are not used, then the call to **requestNamedAccess**() fails, and an e_AuthenticationError exception is raised. This exception contains a reference to a **i_RetailerAuthenticate** interface, which the consumer can use to authenticate himself. After this, the consumer calls **requestNamedAccess**() on **i_RetailerInitial** in order to gain a reference to the **i_RetailerNamedAccess** interface.

3. At this point, an access session has been established, and the consumer domain has a reference to the **i_RetailerNamedAccess** interface.

4. The consumer domain informs the retailer domain of its interfaces and terminal configuration by calling the **setUserCtxt**( ) operation on **i_RetailerNamedAccess**. The retailer gains references to the **i_ConsumerAccess**, **i_ConsumerInvite**, **i_ConsumerTerminal**, and **i_ConsumerSessionInfo** interfaces for use within this access session.

5. The consumer can now, by invoking the appropriate operations on the **i_RetailerNamedAccess** interface:

   - discover services offered by the retailer (**discoverServices**());
   - subscribe to those services (by starting a subscription service);
   - list the access sessions and service sessions they are currently involved with (**listAccessSessions**(), **listServiceSessions**());
   - start a new service session (**startService**());
   - suspend, resume, join and end existing sessions (**suspendSession**(), **resumeSession**(), **endSession**());
   - gain references to retailer-specific interfaces (**getInterfaces**());
   - register interfaces for use outside of an access session (**registerInterfaceOutsideAccessSession**());
   - and more...

6. The retailer can:

2. The mechanism by which the consumer gains this interface is not prescribed by Ret-RP.

- gain references to retailer-specific interfaces (using **i_ConsumerAccess** interface);
- invite the consumer to join a session (using **i_ConsumerInvite** interface);
- discover the terminal configuration (using **i_ConsumerTerminal** interface);
- inform the consumer of changes in their access and service sessions (using **i_UserAccessSessionInfo**, and **i_UserSessionInfo** interfaces);

7. The consumer registers the **i_ConsumerInitial** interface for use outside of an access session, (using **registerInterfaceOutsideAccessSession**() on **i_RetailerNamedAccess**). Then he ends the access session (), and can no longer make requests to the retailer.
8. The retailer can still invite the consumer to join a service session, using **inviteUserOutsideAccessSession**() on the **i_ConsumerInitial** interface.
9. If the consumer wished to join the session they've been invited to, then they would have to establish another access session, (as in step 1.).

4.1.2 Always available outside an Access Session

Only retailer interfaces are always available outside an access session.

The following interfaces are provided by the retailer to allow the consumer and/or retailer to authenticate themselves, and establish an access session.

- **i_RetailerInitial -** This interface is the consumer's initial point of contact for the retailer. It can be used to request the establishment of an access session. The access session provides a consumer access to use his subscribed services, etc., through a `i_RetailerNamedAccess`, or `i_RetailerAnonAccess` interface, if the consumer is authenticated as a named, or anonymous user respectively. If the consumer is not authenticated, it returns a reference to the `i_RetailerAuthenticate` interface, to allow this authentication to occur.
- **i_RetailerAuthenticate** - This interface is used by the consumer to authenticate themselves and the retailer and for passing credentials that can be used to establish the access session.

4.1.2.1 i_RetailerInitial Interface

The `i_RetailerInitial` interface allows the consumer to request the establishment of an access session.

- **requestNamedAccess**() allows the consumer to identify himself to the retailer, and establish an access session. A secure context may have already been set-up between the consumer and the retailer using CORBA security services. In this case, this operation returns a reference to a `i_RetailerNamedAccess` interface. If the consumer has not already been authenticated, then an `e_AuthenticationError` exception will be raised. This contains a reference to a `i_RetailerAuthenticate` interface, which may be used to authenticate and set-up the secure context. Then this operation can be invoked again to retrieve the reference to the `i_RetailerNamedAccess` interface.
- **requestAnonymousAccess**() allows the consumer to establish an access session with the retailer without revealing his identity. The access session will provide access to some services, although the consumer may need to negotiate with the retailer over which

services are available. (The services will obviously not be specialised to the consumer.) The consumer interacts with the retailer through a `i_RetailerAnonAccess` interface. This operation is otherwise the same as `requestNamedAccess`().

4.1.2.2 i_RetailerAuthenticate interface

The `i_RetailerAuthenticate` interface allows the consumer and/or the retailer to be authenticated and acquire credentials, to set-up a secure context. The interface provides a generic mechanism for authentication which can be used to support a number of different authentication protocols.

The primary purpose of this interface is to verify to the consumer and retailer that they are indeed talking to the domain they have been told they are talking to. It is not intended to necessarily identify the consumer. (`requestNamedAccess`() is used to identify the consumer, and provide it access to its services.)

- **`getAuthenticationMethods`**() provides a list of the authentication methods supported by the retailer.
- **`authenticate`**() allows the consumer to select an authentication method, pass authentication data and request specific credentials that may be used for maintaining a secure context. The retailer then returns its authentication data (if required), challenge data for the consumer to respond using `continueAuthentication`() (if required), and the requested credentials (if possible). If further authentication protocol is required before credentials are returned then these can be returned by `continueAuthentication`**()**
- **`continueAuthentication`**() may be called one or more times after `authenticate`(). It allows the consumer to respond to the challenge data returned from `authenticate`() or previous `continueAuthentication`() call. At the first or subsequent calls of `continueAuthentication`() credentials requested by the consumer may be returned according to the protocol requirements.

4.1.3 Available during an Access Session

Both consumer and retailer interfaces are available during an access session.

The consumer supports the following interfaces for the retailer to use during the access session:

- **`i_ConsumerAccess` -** The retailer can find out about the interfaces in the consumer domain using this interface. It provides the retailer with interface references to other interfaces in the consumer domain.
- **`i_ConsumerInvite` -** This interface is used by the retailer to notify the consumer of invitations to join service sessions. The consumer can register an `i_ConsumerInitial` interface to receive invitations outside an access session.
- **`i_ConsumerTerminal`** - This interface is used by the retailer within an access session to access terminal configuration information, e.g. applications installed, hardware configuration, (NAPs), etc.
- **`i_ConsumerAccessSessionInfo`** - This interface is used by the retailer to inform the consumer of state changes to other access sessions which this consumer has with this retailer.
- **`i_ConsumerSessionInfo` -** This interface is used by the retailer to inform the consumer of state changes to service sessions which this consumer has with this retailer. Information is sent on all service sessions, used through all access sessions with this retailer.

All of the consumer supported may be registered with the retailer for use inside or outside of an access session, through operations on the `i_RetailerNamedAccess` interface. Other retailer-specific interfaces not defined by Ret-RP can also be registered. Registration of the first three mentioned here is mandatory, using the `setUserCtxt`() operation on the `i_RetailerNamedAccess` interface. The lifetime for this particular registration is the same as the lifetime of the access session.

The retailer supports two interfaces for use during access sessions. The consumer will only be given a reference to one of these interfaces. If they have authenticated as a named user and invoked the `requestNamedAccess`() operation, they will be given a reference to `i_RetailerNamedAccess`; otherwise if they have authenticated as an anonymous user, and invoked `requestAnonymousAccess`(), then they will be given a reference to `i_RetailerAnonAccess`:

- **`i_RetailerNamedAccess` -** This interface allows a known consumer to access his subscribed services, start and manage service sessions, etc.
- **`i_RetailerAnonAccess` -** This interface is used by the retailer to notify the consumer of invitations to join service sessions. The consumer can register an `i_ConsumerInitial` interface to receive invitations outside an access session.

During an access session, a consumer will have access to one of these interfaces, depending on whether they have authenticated as a named or anonymous user. The current definition of Ret-RP does not allow the change from anonymous to named user in the same access session.

The retailer also supports the following interface:

- **`i_DiscoverServicesIterator`** - A reference to this interface is returned to the consumer after invoking the `discoverServices`() operation on either of the interfaces above. It is used to retrieve the remaining service descriptions, which were not returned directly from `discoverServices`().

4.1.3.1 i_ConsumerAccess interface

The `i_ConsumerAccess` interface allows the retailer access to the consumer domain, during an access session. It allows the retailer to request references to interfaces supported by the consumer domain. These interfaces include those defined by Ret-RP, as well as other retailer specific interfaces.

- **`cancelAccessSession`**() - allows the retailer to cancel this access session. After this operation has been invoked, neither consumer nor retailer will make use of the other interfaces. (Interfaces registered for use outside the access session, or interfaces within another access session can still be used.)

This interface inherits the following operations from the `i_UserAccess` interface, for the retailer to gain references to other interfaces supported by the consumer:

- **`getInterfaceTypes`**() - allows the retailer to discover all of the interface types supported by the consumer domain.
- **`getInterface`**() - allows the retailer to retrieve an interface reference, giving the interface type, and properties.
- **`getInterfaces`**() - allows the retailer to retrieve a list of all the interfaces, supported by the consumer.

This interface is registered with the retailer using the `setUserCtxt()` operation, and is available for use during the current access session.

4.1.3.2 i_ConsumerInvite interface

The `i_ConsumerInvite` interface allows the retailer to send invitations to join service session, during an access session. It is only available during an access session to receive invitations. If the consumer wishes to receive invitations outside of an access session, then they must register the `i_ConsumerInitial` interface for use outside an access session.

- **`inviteUser`**`()` - allows the retailer to invite the consumer 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 `joinSessionWithInvitation()` operation on the `i_RetailerNamedAccess` interface.
- **`cancelInviteUser`**`()` - allows the retailer inform the consumer that an invitation previously sent to the consumer has been cancelled.

This interface is registered with the retailer using the `setUserCtxt()` operation, and is available for use during the current access session.

4.1.3.3 i_ConsumerTerminal interface

The `i_ConsumerTerminal` interface allows the retailer to gain information about the consumer domain's terminal configuration, and applications.

- **`getTerminalInfo`**() - allows the retailer to retrieve information about the consumer domain's terminal. Information on the terminal id, type, network access points, and user applications can be accessed.

This interface is registered with the retailer using the `setUserCtxt()` operation, and is available for use during the current access session.

4.1.3.4 i_ConsumerAccessSessionInfo interface

The `i_ConsumerAccessSessionInfo` interface allows the retailer to inform the consumer of changes of state in other access sessions with the consumer, (e.g. access sessions with the same consumer which are created or deleted). The consumer is only informed about access sessions which they are involved in.

- **`newAccessSessionInfo`**`()` - This (oneway) operation is used by the retailer to inform the consumer about a new access session in which the consumer is involved.
- **`endAccessSessionInfo`**`()` - This (oneway) operation is used by the retailer to inform the consumer that another access session has ended.
- **`cancelAccessSessionInfo`**`()` - This (oneway) operation is used to inform the consumer an access session has been cancelled by the retailer.
- **`newSubscribedServicesInfo`**`()` - This (oneway) operation is used by the retailer to inform the consumer that they have been subscribed to some new services.

This interface is not registered with the retailer using the `setUserCtxt()` operation. Instead the consumer domain must register this interface using the `i_RetailerNamedAccess` interface. It can be registered for use both inside and outside of an access session.

4.1.3.5 i_ConsumerSessionInfo Interface

The `i_ConsumerSessionInfo` interface allows the retailer to inform the consumer of changes of state in service sessions which the consumer is involved in. Information operations are invoked whenever a change to the service session affects the consumer, (i.e. the session is suspended), but not when the change does not affect the consumer, (i.e. another party in the session leaves). This interface is informed of changes in all service sessions involving the consumer, and not just those associated with this access session.

The following operations inform the consumer that:

- **`newSessionInfo`**() - a new service session has been started;
- **`endSessionInfo`**() - an existing service session has ended;
- **`endMyParticipationInfo`**() - the consumer's participation in the session has ended;
- **`suspendSessionInfo`**() - an existing service session has been suspended;
- **`suspendMyParticipationInfo`**() - the consumer's participation in the service session has suspended;
- **`resumeSessionInfo`**() - a suspended session has been resumed
- **`resumeMyParticipationInfo`**() - the consumer's participation in the session has resumed;
- **`joinSessionInfo`**() - the consumer has joined a service session.

This interface can be registered with the retailer using the `setUserCtxt()` operation. If so, the interface is available during the current access session only.

It can be registered at any other time with the retailer using the register interface operations on the `i_RetailerAccess` interface. It can be registered for use both inside and outside of an access session.

4.1.3.6 i_RetailerNamedAccess interface

`i_RetailerNamedAccess` interface allows a known consumer access to his subscribed services. The consumer uses it for all operations within an access session with the retailer. A reference to this interface is returned when the consumer has been authenticated by the retailer and an access session has been established. It is returned by calling `requestNamedAccess`() on the `i_RetailerInitial` interface.

It provides the following operations (which are inherited from `i_ProviderNamedAccess` interface:

- **`setUserCtxt()`** - allows the consumer to inform the retailer about interfaces in the consumer domain, and other consumer domain information. (e.g. user applications available in the consumer domain, operating system used, etc.). It should be called immediately after receiving the reference to this interface, or subsequent operations may raise an exception.
- **`listAccessSessions()`** - allows the consumer in this access session to find out about other access sessions that he has with this retailer. (e.g. A consumer is at work, but has an access session set up at home which runs an active security service session.)
- **`endAccessSession()`** - allows the consumer to end a specified access session, either the current one, or another, found using `listAccessSessions`(). The consumer can also specify some actions to do if there are active service sessions.

- **`getUserInfo`**`()` - gets the consumer's username, and other properties.
- **`listSubscribedServices`**`()`- lists the services to which the consumer is subscribed. Scoping of subscribed services can be done using property lists. The operation returns sufficient information for the consumer to start a particular (subscribed)service.
- **`discoverServices`**`()` - lists all the services available from the retailer. The consumer can scope the list by suppling some properties that the service should have, and a maximum number to return. A reference to an `i_DiscoverServicesIterator` interface can be used to retrieve the remaining services.
- **getServiceInfo**() - returns the service information for a particular service (identified in the invocation by its serviceId). Similar information (t_ServiceProperties) can be obtained with the listSubscribedServices or discoverServices, but the getServiceInfo is a simplified version, targeting on a single service, and independantly from the subscription status.
- **listRequiredServiceComponents**() - retrieves information on how to download the application software in case of Java applets. The terminalInfo is included as an IN parameter to avoid an explicit call of the getTerminalInfo operation. For example in case of Java applet download, the property list will contain an entry with a name-value pair describing the url of the Java applet; the name will be "URL" and the value the string value of the URL.
- **`listServiceSessions`**`()` - lists the service sessions of the consumer. The request can be scoped by the access session, and session properties, (e.g. active, suspended, service type, etc.).
- **`getSession`**`(Models/InterfaceTypes/Interface/Interfaces)()` - all retrieve information on a particular session.
- **`listSessionInvitations`**`()` - lists the invitations to join a service session that have been sent to the consumer.
- **`listSessionAnnouncements`**`()` - lists the service sessions with have been announced. It can be scoped by some announcement properties.
- **`startService`**`()` - allows the consumer to start a service session.
- **`endSession`**`()` - allows the consumer to end a service session.
- **`endMyParticipation`**`()` - allows the consumer to end his participation in a service session.
- **`suspendSession`**`()` - allows the consumer to suspend a service session.
- **`suspendMyParticipation`**`()` - allows the consumer to suspend his participation in a service session.
- **`resumeSession`**`()` - allows the consumer to resume a service session.
- **`resumeMyParticipation`**`()` - allows the consumer to resume his participation in a service session.
- **`joinSessionWithInvitation`**`()` - allows the consumer to join a service session, to which he has been invited.
- **`joinSessionWithAnnouncement`**`()` - allows the consumer to join a service session, which has been announced.
- **`replyToInvitation`**`()` - allows the consumer to reply to an invitation. It can be used to inform the service session to which they have been invited, that they will/will not be joining the session, or to send the invitation somewhere else. (It does not allow the consumer to join the session.)

It also supports the following operations inherited from `i_ProviderAccessInterfaces` interface. These are useful for accessing retailer specific interfaces.:

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

4.1.3.7 i_RetailerAnonAccess interface

The `i_RetailerAnonAccess` interface allows an anonymous consumer access to the retailer's services. The anonymous consumer uses it for all operations within an access session with the retailer. This interface is returned when the consumer calls `requestAnonymousAccess`() on the `i_RetailerInitial` interface.

Currently the operations for this interface are not defined. It will support operations similar to those of the `i_RetailerNamedAccess` interface.

4.1.3.8 i_DiscoverServicesIterator

The `i_DiscoverServicesIterator` interface is used returned by calls to the `discoverServices`() operation. This operation is used to retrieve a list of services supported by the retailer which match a set of properties. The list generated by this operation may be too large to return as an out parameter. This interface allows the list to be retrieved in digestible chunks by the consumer. Each call to `discoverServices`() returns a new instance of this interface.

- **`maxLeft`**() - The consumer can find out how many unseen services are left
- **`nextN`**() - The consumer can indicate that he wants to get information about the next n services.
- **`destroy`**() - The consumer informs the retailer that the interface is no longer needed.

4.1.4 Available outside an Access Session if Registered

The consumer can register some his interface for use by the retailer outside of the current access session. An interface can be registered using the `registerInterfaceOutsideAccessSession`() operation on the `i_RetailerNamedAccess`

interface. If registered, the retailer will retain a reference to the interface when the consumer/retailer end the current access session. The retailer can invoke operations on this interface without an access session being present.

The retailer will not use the interface registered until the access session in which it was registered has ended. They will continue to use the interface until the interface is unregistered. If another access session is established, the retailer will still invoke operations on the registered interface, in addition to new interfaces provided as part of the new access session.

- **`i_ConsumerInitial`.** This interface allows the retailer to initiate an access session with the consumer. It also allows the retailer to send invitations to the consumer outside of an access session.
- **`i_ConsumerTerminal`.** The retailer will use this interface to access terminal configuration information, if necessary. See previous description.
- **`i_ConsumerAccessSessionInfo`.** The retailer will use this interface to inform the consumer of changes to any of their access sessions. See previous description.
- **`i_ConsumerSessionInfo`.** The retailer will use this interface to inform the consumer of changes in state to any of their service sessions. See previous description.

4.1.4.1 i_ConsumerInitial Interface

The `i_ConsumerInitial` interface allows the retailer to contact the consumer outside of an access session. It can be used to request that the consumer establish an access session with the retailer; and to invite a user to join a service session.

This interface is only available to the retailer if the consumer has registered it during an access session, (using the `registerInterfaceUntilUnregistered()` operation, on the `i_RetailerNamedAccess` interface). It is NOT available through a broker, as the `i_RetailerInitial` interface is.

The following operations are available:

- **`requestAccess`**() - allows the retailer to request the consumer to set up an access session.
- **`inviteUserWithoutAccessSession`**() - allows the retailer to send an invitation to the consumer while he is not involved in an access session with the retailer.
- **`cancelInviteUserWithoutAccessSession`**() - allows the retailer to cancel an invitation sent to the consumer.

4.2 User-Provider Interfaces

The interfaces defined above are for use over the Ret-RP. The interface names use include the names Consumer and Retailer in order to identify that they are for use over the Ret-RP. The descriptions above include all of the operations that are used over Ret-RP.

Other reference points will want to use similar interfaces as those defined above, for access related activities, (e.g. establishing an access session, starting services, etc.). In order to allow other reference points to re-use interfaces and operations, a set of generic access interfaces are defined. These interface support the access session roles defined in the Service Architecture document [ref5.0]. The roles supported are access user and access provider. These interfaces can be recognized by the prefix `i_User` or `i_Provider`.

The interfaces defined for use over Ret-RP have been defined above, including all of the inherited operations. All of the `i_Consumer` and `i_Retailer` interfaces are inherited from corresponding `i_User` and `i_Provider` interfaces. Any specialisations for the Ret-RP are defined in the `i_Consumer` / `i_Retailer` interfaces. However, no specialisations are defined at present.

The figures below define the inheritance hierarchy for both Ret-RP interfaces, and the generic User-Provider interfaces.

4.2.1 User Interfaces

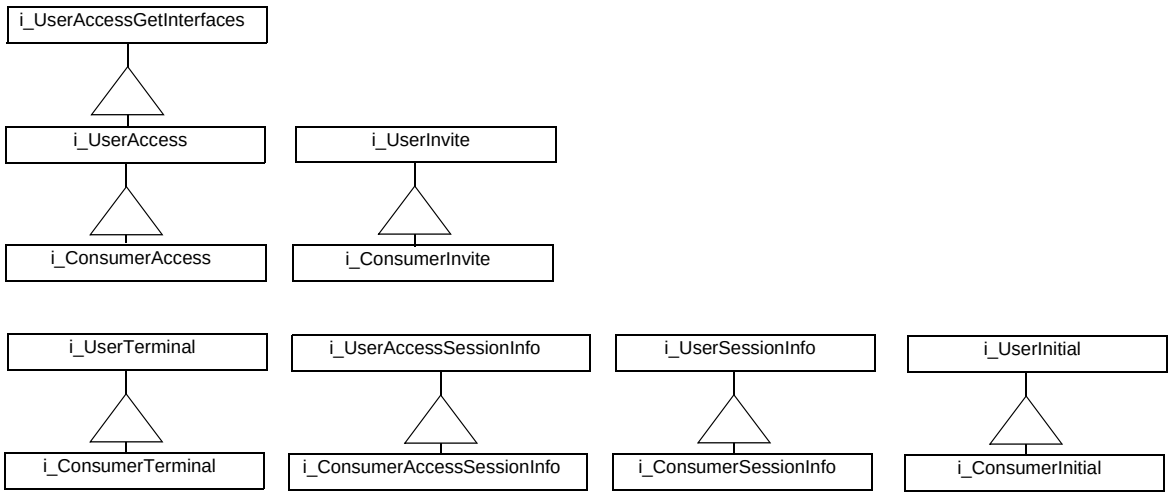


Figure 4-2. Consumer interfaces inherited from User interfaces.

Figure 4-2 shows the consumer interfaces, and the user interfaces that they inherit from. The user and consumer interfaces have a simple mapping, (all the consumer interfaces inherit from a correspondingly named user interface). All of the user interfaces define the operations that are described for the consumer interfaces in Section 4.1. The only exception to this is the `i_UserAccess` interface that inherits all of this operation from `i_UserAccessGetInterfaces`. This is to allow other interfaces to re-use the operations to retrieve interfaces.

4.2.1.1 i_UserAccess

This interface inherits from the abstract interface `i_UserAccessGetInterfaces`, and defines the following operation:

- **`cancelAccessSession`**`()`

4.2.1.2 i_UserInvite

This interface defines the following operations:

- **`inviteUser`**`()`
- **`cancelInviteUser`**`()`

4.2.1.3 i_UserTerminal

This interface defines the following operation:

- **getTerminalInfo**()

4.2.1.4 i_UserAccessSessionInfo

This interface defines the following operations:

- **newAccessSessionInfo**()
- **endAccessSessionInfo**()
- **cancelAccessSessionInfo**()
- **newSubscribedServicesInfo**()

4.2.1.5 i_UserSessionInfo

This interface defines the following operations:

- **newSessionInfo**()
- **endSessionInfo**()
- **endMyParticipationInfo**()
- **suspendSessionInfo**()
- **suspendMyParticipationInfo**()
- **resumeSessionInfo**()
- **resumeMyParticipationInfo**()
- **joinSessionInfo**()

4.2.1.6 i_UserInitial

This interface defines the following operations:

- **requestAccess**()
- **inviteUserOutsideAccessSession**()
- **cancelInviteUserOutsideAccessSession**()

4.2.2 Provider interfaces

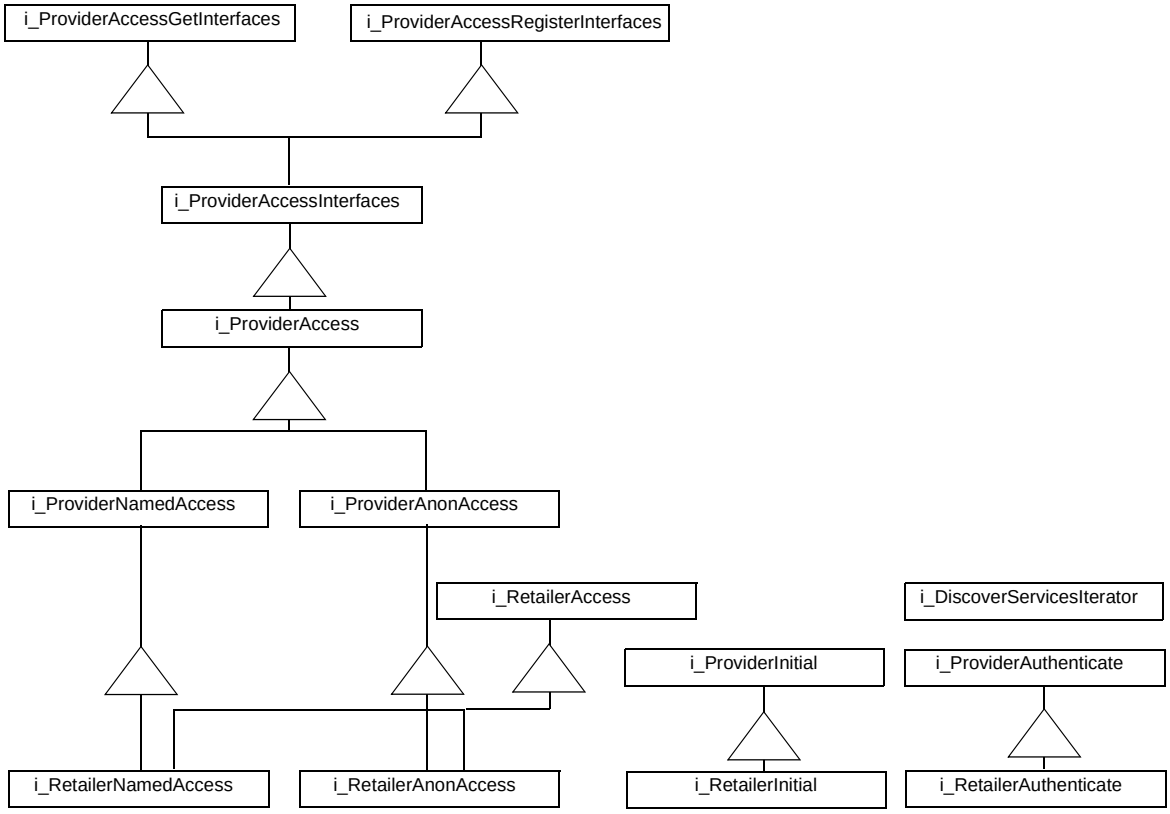


Figure 4-3. Retailer interfaces inherited from Provider interfaces

Figure 4-3 shows the retailer interfaces, and the provider interfaces that they inherit from.

The inheritance hierarchy for the `i_RetailerNamedAccess` and `i_RetailerAnonAccess` interface is complex. Both inherit from `i_RetailerAccess`. This interface defines Ret-RP specific operations that are common to both interfaces. (Currently no operations are defined here.)

`i_RetailerNamedAccess` also inherits from `i_ProviderNamedAccess`, which defines the operations available for the generic access provider role, where the user domain supports a known user. `i_ProviderNamedAccess` defines all of the operations offered by `i_RetailerNamedAccess`.

`i_RetailerAnonAccess` inherits from `i_ProviderAnonAccess`, which defines the operations available for the generic access provider role, where the user domain supports an anonymous user. Currently, `i_ProviderAnonAccess` only inherits operations from `i_ProviderAccess`.

`i_ProviderAccess` interface defines the generic access provider role, for re-use in other reference points. It is inherited by both `i_ProviderNamedAccess` and `i_ProviderAnonAccess`. Currently, no operations are defined for this interface. In the future, some of the operations defined for `i_ProviderNamedAccess` will be moved here, as they are common to both interfaces, being appropriate to known and anonymous users.

4.2.2.1 i_ProviderNamedAccess

This interface defines the following operations, and inherits others from `i_ProviderAccess`:

- **`setUserCtxt`**`()`
- **`listAccessSessions`**`()`
- **`endAccessSession`**`()`
- **`getUserInfo`**`()`
- **`listSubscribedServices`**`()`
- **`discoverServices`**`()`
- **`getServiceInfo`**`()`
- **`listServiceSessions`**`()`
- **`getSession`**`(Models/InterfaceTypes/Interface/Interfaces)()`
- **`listSessionInvitations`**`()`
- **`listSessionAnnouncements`**`()`
- **`startService`**`()`
- **`endSession`**`()`
- **`endMyParticipation`**`()`
- **`suspendSession`**`()`
- **`suspendMyParticipation`**`()`
- **`resumeSession`**`()`
- **`resumeMyParticipation`**`()`
- **`joinSessionWithInvitation`**`()`
- **`joinSessionWithAnnouncement`**`()`
- **`replyToInvitation`**`()`

4.2.2.2 i_ProviderAnonAccess

This interface defines no operations. It inherits from `i_ProviderAccess`.

4.2.2.3 i_ProviderAccess

This interface defines no operations. It inherits from `i_ProviderAccessInterfaces`.

4.2.3 Abstract interfaces

This section describes the abstract interfaces which are inherited in several retailer and consumer interfaces. They are not exported over Ret. The main purpose of these interfaces is to provide a generic mechanism for registration and retrieval of interfaces in a certain domain.

- **`i_UserAccessGetInterfaces`** - allows the provider to retrieve all interfaces, only interfaces that have certain properties or interface types of the current access session.
- **`i_ProviderAccessGetInterfaces`** - allow the user to retrieve all interfaces, only interfaces that have certain properties or interface types of the current access session.
- **`i_ProviderAccessRegisterInterfaces`** - allow the user to register interfaces for the lifetime of an access session or permanent. It also offers an operation to unregister interfaces.
- **`i_ProviderAccessInterfaces`** - inherits the previous two and does not offer additional functionality.

4.2.3.1 i_UserAccessGetInterfaces

This interface defines the following operations:

- **getInterfaceTypes**()
- **getInterface**()
- **getInterfaces**()

4.2.3.2 i_ProviderAccessGetInterfaces

This interface defines the following operations:

- **getInterfaceTypes**()
- **getInterface**()
- **getInterfaces**()

4.2.3.3 i_ProviderAccessRegisterInterfaces

This interface defines the following operations:

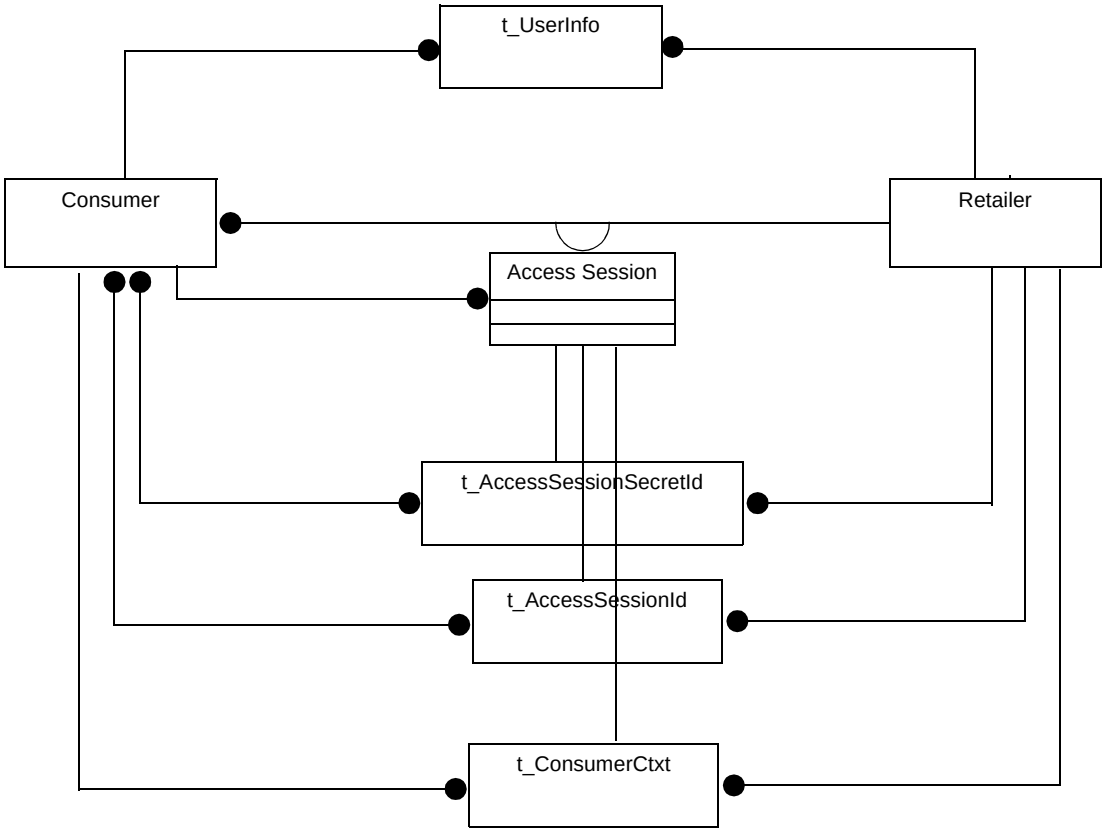
- **registerInterface**()
- **registerInterfaces**()
- **registerInterfaceOutsideAccessSession**()
- **registerInterfacesOutsideAccessSession**()
- **listRegisteredInterfaces**()
- **unregisterInterface**()
- **unregisterInterfaces**()

4.2.3.4 i_ProviderAccessInterfaces

This interface inherits from **i_ProviderAccessGetInterfaces** and **i_ProviderAccessRegisterInterfaces** with no additional operations

4.3 Access Information View

This section describes the types of information passed across the Access part of the Ret RP. The types are passed in operations defined in the access interfaces.



4.3.1 Access Session Information

```
// module TINAAccessCommonTypes
typedef unsigned long t_AccessSessionId;
typedef sequence<octet, 16> t_AccessSessionSecretId;
```

`t_AccessSessionId` is used to identify an access session. The `t_AccessSessionId` for the consumer's current access session is returned by `requestNamedAccess()` or `requestAnonAccess()`. The `t_AccessSessionId` for other access sessions can be found using `listAccessSessions()` on the `i_RetailerNamedAccess` interface. (Anonymous users can only have a single access session, and so only a single `t_AccessSessionId`). The `t_AccessSessionId` is scoped by the consumer, i.e. for a single consumer (`t_UserId`) all `t_AccessSessionId`'s are unique.

`t_AccessSessionSecretId` is used to identify within which access session a request on the `requestNamedAccess()` is made. Each access session of a consumer has a unique `t_AccessSessionSecretId`. It is returned by `requestNamedAccess()`.

All operations on `requestNamedAccess()` take an `t_AccessSessionSecretId` as their first parameter. This parameter can be checked by the retailer to determine within which access session of the consumer the request originated. This is useful when the behaviour of the request is dependant on the consumer context, (e.g. `startService()` checks the consumer context to determine if this service can be used).

`t_AccessSessionSecretId` is only known within the access session it is created. It is not known to other access sessions of the same consumer, and is not available through `listAccessSessions()`. This is because it is being used to determine the access session within which the request is made. If another access session gained the `t_AccessSessionSecretId` of this access session, then it could use it to pretend the request came from this access session. For this reason, it should not be displayed to a human consumer, or other applications in the consumer domain. `t_AccessSessionSecretId` is not itself a security mechanism, as CORBA security is still needed to set-up security contexts between the consumer and retailer domains. However, it does allow the retailer to easily discover the sender of a particular request.

4.3.2 User Information

Most of the user related information is described in the common types section (Section 3.3.2)

```
// module TINAAccessCommonTypes
struct t_UserInfo {
      TINACommonTypes::t_UserId userId;
      TINACommonTypes::t_UserName name;
      TINACommonTypes::t_UserProperties userProperties;
};
```

`t_UserInfo` describes the user. It contains the `t_UserId`, the user's name, and `t_UserProperties`. It is returned by `getUserInfo()` on the `i_ProviderAccess` interface.

4.3.3 User Context Information

```
// module TINACommonTypes
typedef Istring t_UserCtxtName;


// module TINAProviderAccess
struct t_UserCtxt {
      TINACommonTypes::t_UserCtxtName ctxtName;
      TINAAccessCommonTypes::t_AccessSessionId asId;
      Object accessIR;            // type: i_UserAccess[1]
      Object terminalIR;          // type: i_UserTerminal
      Object inviteIR;            // type: i_UserInvite
      Object sessionInfoIR;       // type: i_UserSessionInfo
      TINAAccessCommonTypes::t_TerminalConfig terminalConfig;
};
```

`t_UserCtxt` informs the retailer about the user and the consumer domain, including the name of the context, interfaces available during this access session, and terminal configuration information

1. The type written in the IDL of this parameter is the base class type. The actual type will depend on the reference point used. In this case the retailer can expect the i_Consumer<> type. See also remark at requestNamedAccess(), output namedAccessIR.

t_UserCtxtName is a name given to this consumer context. It is generated by the consumer domain. It is used to distinguish between access sessions to different consumer domains/terminals. When listing the access sessions, the t_UserCtxtName is returned, (along with the t_AccessSessionId), as the former should be a more human readable name for the 'terminal' that the access session is connected to.

accessIR is a reference to the i_UserAccess interface supported by the consumer domain for use in this access session.

terminalIR is a reference to the i_UserTerminal interface supported by the consumer domain for use in this access session.

inviteIR is a reference to the i_UserInvite interface supported by the consumer domain for use in this access session.

All of the preceeding three interface references should be set to valid interfaces in the consumer domain.

sessionInfoIR is a reference to the i_UserSessionInfo interface supported by the consumer domain for use in this access session. It is not necessary to supply a reference for this interface.

t_TerminalConfig is a structure containing the terminal id and type; the network access point id and type; and a list of terminal properties. Two property types have been defined t_TerminalInfo, described below, and t_ApplicationInfoList, a list of the user applications on the terminal.

t_TerminalInfo gives details on the type of terminal, operating system, etc.

```
// module TINAAccessCommonTypes
struct t_TerminalInfo {
      t_TerminalType terminalType;
      string operatingSystem;              // includes the version
      TINACommonTypes::t_PropertyList networkCards;
      TINACommonTypes::t_PropertyList devices;
      unsigned short maxConnections;
      unsigned short memorySize;
      unsigned short diskCapacity;
};
```

Draft definition: This structure is draft only. We would be glad to receive any feedback concerning this structure's signature.

t_TerminalType is an enumerated type, giving the type of the terminal.

operatingSystem provides the operating system type and version as a string.

networkCards and devices are property lists of the physical devices of the terminal. The property names and values are not defined at present, so their use is retailer specific.

maxConnections is the maximum number of network connections which can be supported by the terminal.

memorySize is the amount of RAM in Megabytes.

diskCapacity is the amount of disk storage in Megabytes.

4.3.4 Service and Session Information

```
// module TINAAccessCommonTypes
struct t_ServiceInfo {
      t_ServiceId id;
      t_UserServiceName name;
      t_ServiceProperties properties;
};
```

t_ServiceInfo is a structure which describes a subscribed service of the consumer.

t_ServiceId is the identifies for the service. t_ServiceId is unique among all the consumer's subscribed services. (Other consumer's may be subscribed to the same service, but will have a different t_ServiceId.). The t_ServiceId value persists for the lifetime of a subscription.

t_UserServiceName is the name of the service as a string. The name is chosen by the subscriber when they subscribe to the service. It is the name of the service displayed to the user.

t_ServiceProperties 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, e.g. using discoverServices() on i_RetailerNamedAccess.) Currently, no properties have been defined for t_ServiceProperties, and so its use is retailer specific.

```
// module TINAAccessCommonTypes
struct t_SessionInfo {
      TINACommonTypes::t_SessionId id;
      t_SessionPurpose purpose;
      TINACommonTypes::t_ParticipantSecretId secretId;
      TINACommonTypes::t_PartyId myPartyId;
      t_UserSessionState state;
      TINACommonTypes::t_InterfaceList itfs;
      TINACommonTypes::t_SessionModelList sessionModels;
      TINACommonTypes::t_SessionProperties properties;
};
```

t_SessionInfo is a structure, which contains information which allows the consumer domain to refer to a particular session, when using interfaces within an access session (e.g. i_RetailerNamedAccess). It also contains information for the usage part of the session, including the interface references to interact with the session.

id is the identifier for this session. It is unique to this session, among all sessions that this consumer interacts with through this retailer. (i.e. if the consumer interacts with multiple retailers concurrently, then they may return t_SessionId's which are identical.)

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

secretId is an identifier that the consumer must use when interacting with interfaces on the session which are defined by the TINA Session Model. (See Usage Part of the Ret-RP for more details.)

myPartyId is the party identifier of this consumer. If the session is using the TINA Session Model, with the MultipartyFS feature set, then this identifier will be used to identify this party. The t_PartyId's of other parties in the session are also available through MultipartyFS interfaces.

`state` is the session state as perceived by this consumer. It can be: `UserUnknownSessionState`, `UserActiveSession`, `UserSuspendedSession`, `UserSuspendedParticipation`, `UserInvited`, `UserNotParticipating`. But as the session has just been started, it is likely to be `UserActiveSession`.

`itfs` is a list of interface types and references supported by the session. (It may include service specific interfaces for the consumer to interact with the session.

`sessionModels` is a list of the session models and feature sets that are supported by the session. It may include interface references to interfaces supported for each feature set.

`properties` is a list of properties of the session. Its use is retailer specific.

4.4 Access Interface definitions

This section describes in detail the interfaces supported over Ret RP. First all of the consumer interfaces are described, followed by the retailer interfaces. Each interface and all its supported operations are defined.

Many of the operations are inherited from other interfaces. However they are described here as though they were defined on the Ret RP specified interfaces. Only the interfaces defined here must be supported for the Ret RP. It is not necessary to support the same inheritance hierarchy as defined previously in Section 4.2.

4.4.1 Consumer Domain Interfaces

These are the interfaces supported by the consumer domain, which are available across the Ret RP:

- `i_ConsumerInitial`
- `i_ConsumerAccess`
- `i_ConsumerInvite`
- `i_ConsumerTerminal`
- `i_ConsumerSessionInfo`
- `i_ConsumerAccessSessionInfo`

4.4.1.1 i_ConsumerInitial Interface

```
// module TINARetConsumerInitial

interface i_ConsumerInitial
                        : TINAUserInitial::i_UserInitial
{
};
```

This interface is provided to allow a Retailer initiate an access session with the consumer.It also allows the consumer to receive invitations outside of an access session.

The purpose of this interface is to provide an initial contact point for the retailer wishing to contact the consumer. (So its purpose is similar to that of `i_RetailerInitial` interface). However, this interface is only available to a retailer if the consumer had previously registered the interface for use outside an access session. This is achieved using `registerInterfaceOutsideAccessSession` operation on the `i_RetailerNamedAccess` interface.

The operations described in the following sections are all inherited into this interface from the `i_UserInitial` interface, which supports the generic user-provider roles. No Ret RP specific specialisations are defined for this interface.

The following operation signatures are taken from the module `TINAUserInitial`. All unscoped types need to scoped by `TINAUserInitial::` when used by clients of the `i_ConsumerInitial` interface.

4.4.1.1.1. requestAccess()

```
void requestAccess (
      in t_ProviderId providerId,
```

```
        out t_AccessReply reply
    );
```

Draft definition: This operation is draft only. We would be glad to receive any feedback concerning this operations signature and semantics. See Section 6.1.1 for details on the stability of this operation.

This operation allows the retailer to request that an access session is established between the consumer and the retailer.

This operation only allows the retailer to request that an access session is established with the consumer. It does not allow the access session to actually be established. In order to set up an access session the consumer must contact the retailer, using the `i_RetailerInitial` interface, and request that an access session is established.

The retailer passes his `t_ProviderId` to the consumer. The consumer uses this to contact the provider, and gain a reference to an `i_RetailerInitial` interface.

The `t_AccessReply` parameter allows the consumer to inform the retailer of the action they will take in response to the request. The following reply codes are defined:

- `SUCCESS` - the consumer agrees to establish an access session. (The consumer will establish the access session as described above.)
- `DECLINE` - the consumer declines to initiate an access session.
- `FAILED` - the consumer is unable to establish an access session.
- `FORBIDDEN` - the consumer domain is not authorised to accept the request.

4.4.1.1.2. inviteUserAccessSession()

```
    void inviteUserOutsideAccessSession (
         in t_ProviderId providerId,
         in TINAAccessCommonTypes::t_SessionInvitation invitation,
         out TINACommonTypes::t_InvitationReply reply
    );
```

Draft definition: This operation is draft only. We would be glad to receive any feedback concerning this operations signature and semantics. Specifically, the `t_SessionInvitation` and `t_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 retailer to send an invitation to join a service session, to a consumer that is not involved in an access session.

This operation is used if the consumer has previously registered this interface for use outside of an access session, and the consumer is not currently in an access session. If the consumer is in an access session with this retailer, then invitations will not be sent using this operation, but will be delivered to the `i_ConsumerAccess` interface involved in the access session.

In order to join the service session described by the invitation, the consumer must establish an access session with the retailer, and use `joinSessionWithInvitation()` operation on the `i_RetailerNamedAccess` interface. The service session cannot be joined without an access session with the retailer.

`t_ProviderId` identifies the retailer to the consumer.

t_SessionInvitation describes the service session to which the consumer has been invited, and provides an t_InvitationId to identify this invitation when joining. (It does not give interface references to the session, nor any information which would allow the consumer to join the session without first establishing an access session with this retailer.) It also provides a t_UserId with the id of the invited user. The consumer domain can check that the invitation is for an 'end-user' that is known to this domain. (For more details, see Section 3.3.5, "Invitations and Announcements").

An t_InvitationReply is returned which allows the consumer to inform the retailer of the action they will take regarding the invitation. (For more details, see Section 3.3.5).

4.4.1.1.3. cancelInviteUserOutsideAccessSession()

```
void cancelInviteUserOutsideAccessSession (
      in t_ProviderId providerId,
      in TINAAccessCommonTypes::t_InvitationId id
) raises (
      TINAAccessCommonTypes::e_InvitationError
);
```

Draft definition: This operation is draft only. We would be glad to receive any feedback concerning this operations signature and semantics. Specifically, the t_SessionInvitation and t_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 retailer to cancel an invitation to join a service session which has been sent to a consumer. The operation can be used to cancel invitations which have been sent both within an access session, (using inviteUser() on i_ConsumerInvite), and outside of an access session, (using inviteUserOutsideAccessSession on i_ConsumerInitial).

t_ProviderId identifies the retailer to the consumer.

t_InvitationId is used in together with the t_ProviderId in order to determine the invitation to be cancelled. (t_InvitationId's are unique to a retailer only. If a consumer has received invitations from several retailers, then invitations from different retailers may have the same id.)

If the t_InvitationId list is unknown to the consumer, then the operation should raise an e_InvitationError exception with the InvalidInvitationId error code.

4.4.1.2 i_ConsumerAccess Interface

```
// module TINARetConsumerAccess

interface i_ConsumerAccess

                        : TINAUserAccess::i_UserAccess

{
};
```

This interface allows the retailer access to the consumer domain, during an access session. It provides operations for the retailer to request references to interfaces supported by the consumer domain. These interfaces include those defined by Ret RP, as well as other retailer specific interfaces.

It is similar in purpose to i_RetailerAccess, in that it is available during the access session. It is passed to the retailer domain as part of setUserCtxt() on i_RetailerNamedAccess interface.

All the operations described below are inherited into this interface from the `i_UserAccess` interface, which supports the generic user-provider roles. No Ret RP specific specialisations are defined for this interface.

The following operation signatures are taken from the module `TINAUserAccess`. All unscoped types need to scoped by `TINAUserAccess::` when used by clients of the `i_ConsumerAccess` interface.

4.4.1.2.1. cancelAccessSession()

```
void cancelAccessSession(
      in t_CancelAccessSessionProperties options
);
```

Draft definition: This operation is draft only. We would be glad to receive any feedback concerning this operations signature and semantics.

`cancelAccessSession`() allows the retailer to end an access session with the consumer. The retailer can use this operation to terminate an access session without the consumer's permission.

When this operation is invoked, the secure and trusted relationship between the consumer and retailer has ended. Neither retailer nor consumer side interfaces available during the access session can be used to make requests. (Interfaces which have been registered for use outside an access session can still be used).

`options` is a property list describing retailer specific options or action taken by the retailer when cancelling the access session, (i.e. the retailer may have suspended the consumer's participation in their active service sessions). Currently no specific property names and values have been defined for `t_CancelAccessSessionProperties`, and so its use is retailer specific.

This operation does not affect any contractual relationship between the consumer and retailer. The consumer can still request the establishment of an access session, and other access sessions will not have been terminated.

4.4.1.2.2. getInterfaceTypes()

```
void getInterfaceTypes (
      out TINACommonTypes::t_InterfaceTypeList itfTypes
) raises (
      TINACommonTypes::e_ListError
);
```

This operation returns a list of the interface types supported by the consumer domain.

`itfTypes` are all the interface types supported by the consumer domain. It is a `sequence` of `t_InterfaceTypeName`'s, which are strings representing the interface types supported by the consumer. `itfTypes` should include all the interface types that can be supported by the consumer.

If the `itfTypes` list is unavailable, because the interface types supported by the session are not known, then the operation should raise an `e_ListError` exception with the `ListUnavailable` error code.

4.4.1.2.3. getInterface()

```
void getInterface (
      in TINACommonTypes::t_InterfaceTypeName itfType,
      in TINACommonTypes::t_MatchProperties desiredProperties,
```

```
        out TINACommonTypes::t_InterfaceStruct itf
    ) raises (
        TINACommonTypes::e_InterfacesError,
        TINACommonTypes::e_PropertyError
    );
```

This operation returns an interface, of the type requested, supported by the consumer domain.

`type` identifies the interface type of the interface reference to be returned.

The `desiredProperties` parameter can be used to identify the interface to be returned. `t_MatchProperties` identifies the properties which the sessions must match. It also defines whether a session must match one, all or none of the properties. Currently, no interface property names and values have been defined for Ret RP, and its use is retailer specific.

`itf` is returned by this operation. It contains the `t_InterfaceTypeName`, an interface reference (`t_IntRef`) and the interface properties (`t_InterfaceProperties`) of the interface type requested.

If the consumer does not support interfaces of `type`, then the operation should raise the `e_InterfacesError`, with the `InvalidInterfaceType` error code.

If an invalid property is passed, the operation should raise a `e_PropertyError`.

4.4.1.2.4. getInterfaces()

```
    void getInterfaces (
        out TINACommonTypes::t_InterfaceList itfs
    ) raises (
        TINACommonTypes::e_ListError
    );
```

This operation returns a list of all the interfaces supported by the consumer.

`itfs` is returned by this operation. It is a `sequence` of `t_InterfaceStruct` structures which contain the `t_InterfaceTypeName`, an interface reference (`t_IntRef`) and the interface properties (`t_InterfaceProperties`) of each interface.

If the operation cannot, or refuses to, return the interfaces, it should raise the `e_ListError` exception.

4.4.1.3 i_ConsumerInvite Interface

```
// module TINARetConsumerAccess

interface i_ConsumerInvite
                        : TINAUserAccess::i_UserInvite
{
};
```

This interface allows the retailer to send an invitation to the consumer requesting that they join a service session. It can only be used during an access session to receive invitations. It is passed to the retailer domain as part of `setUserCtxt()` on `i_RetailerNamedAccess` interface. If the consumer wishes to receive invitations outside of an access session, then they must register the `i_ConsumerInitial` interface.

The operations described in the following sections are all inherited into this interface from the `i_UserInvite` interface, which supports the generic user-provider roles. No Ret RP specific specialisations are defined for this interface.

The following operation signatures are taken from the module `TINAUserAccess`. All unscoped types need to scoped by `TINAUserAccess::` when used by clients of the `i_ConsumerInvite` interface.

4.4.1.3.1. inviteUser()

```
void inviteUser (
      in  TINAAccessCommonTypes::t_SessionInvitation invitation,
      out TINACommonTypes::t_InvitationReply reply
) raises (TINAAccessCommonTypes::e_InvitationError);
```

Draft definition: This operation is draft only. We would be glad to receive any feedback concerning this operations signature and semantics. Specifically, the `t_SessionInvitation` and `t_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 retailer to invite the consumer to join a service session. It can only be used during an access session.

`t_SessionInvitation` describes the service session to which the consumer has been invited, and provides an `t_InvitationId` to identify this invitation when joining. (It does not give interface references to the session, nor any information which would allow the consumer to join the session outside of an access session with this retailer.)

An `t_InvitationReply` is returned which allows the consumer to inform the retailer of the action they will take regarding the invitation. (For more details, see Section 3.3.5).

The consumer may join the service session described by the invitation, from within this access session, or they may establish another access session with this retailer. The same `t_InvitationId` will refer to this invitation in both access sessions. The consumer should use `joinSessionWithInvitation()` on the `i_RetailerNamedAccess` interface. The service session cannot be joined without an access session with the retailer.

4.4.1.3.2. cancelInviteUser()

```
void cancelInviteUser (
      in TINAAccessCommonTypes::t_InvitationId id
      ) raises (
      TINAAccessCommonTypes::e_InvitationError
);
```

Draft definition: This operation is draft only. We would be glad to receive any feedback concerning this operations signature and semantics. Specifically, the `t_SessionInvitation` and `t_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 retailer to cancel an invitation to join a service session which has been sent to a consumer. The operation can be used to cancel invitations which have been sent both within an access session, (using `inviteUser`() on `i_ConsumerInvite`), and outside of an access session, (using `inviteUserOutsideAccessSession` on `i_ConsumerInitial`).

`t_InvitationId` is used in together with the `t_ProviderId` in order to determine the invitation to be cancelled. (`t_InvitationId`'s are unique across all access sessions with the same retailer).

If the `t_InvitationId` list is unknown to the consumer, then the operation should raise an `e_InvitationError` exception with the `InvalidInvitationId` error code. (It is possible to receive a `cancelInviteUser` before a corresponding `inviteUser`, especially if the cancel is sent just after the access session is established. This operation should raise the exception anyway.)

4.4.1.4 i_ConsumerTerminal Interface

```
// module TINARetConsumerAccess

interface i_ConsumerTerminal
                          : TINAUserAccess::i_UserTerminal
{
};
```

This interface allows the retailer to gain information about the consumer domain's terminal configuration, and applications. It is passed to the retailer domain as part of `setUserCtxt`() on `i_RetailerNamedAccess` interface. If the consumer wishes to allow the retailer access to terminal information outside of an access session, then they must register this interface, using `registerInterfaceOutsideAccessSession`() on `i_RetailerNamedAccess`.

Draft definition: This interface is draft only. We would be glad to receive any feedback concerning operation which would be useful in determining the facilities of a terminal, or consumer domain. Specifically, we may enhance this interface to allow a retailer to ask more specific questions about the consumer domain.

The operations described in the following sections are all inherited into this interface from the `i_UserTerminal` interface, which supports the generic user-provider roles. No Ret RP specific specialisations are defined for this interface.

The following operation signatures are taken from the module `TINAUserAccess`. All unscoped types need to scoped by `TINAUserAccess::` when used by clients of the `i_ConsumerTerminal` interface.

4.4.1.4.1. getTerminalInfo()

```
void getTerminalInfo(
      out TINAAccessCommonTypes::t_TerminalInfo terminalInfo
);
```

Draft definition: This operation is draft only. We would be glad to receive any feedback concerning this operations signature and semantics.

This operation allows the retailer to receive all the information about the consumer domain's terminal configuration, that the consumer wishes the retailer to have access to.

The operation returns the `t_TerminalInfo` structure, giving details on the type of terminal, operating system, etc. See Section 4.3.3, "User Context Information" for details.

4.4.1.5 i_ConsumerAccessSessionInfo Interface

```
// module TINARetConsumerAccess

interface i_ConsumerAccessSessionInfo

                    : TINAUserAccess::i_UserAccessSessionInfo
{
};
```

This interface allows the retailer to inform the consumer of changes of state in other access sessions with the consumer, (e.g. access sessions with the same consumer which are created or deleted). The consumer is only informed about access sessions which they are involved in.

This interface is NOT automatically passed to the retailer, as part of `setUserCtxt`() on `i_RetailerNamedAccess` interface. If the consumer wishes to be informed of changes in other access session, then they must register this interface, using `registerInterface`() on `i_RetailerNamedAccess`. Then the retailer will tell the consumer about access session changes, until this interface is unregistered, or the current access session ends.

If the consumer wishes to be informed of access session changes outside of an access session, then they must register this interface, using `registerInterfaceOutsideAccessSession`() on `i_RetailerNamedAccess`. The operations do not include a `t_ProviderId`, so if this interface is registered for use outside an access session, a separate interface must be registered with each retailer. Retailers can not share this interface, because `t_AccessSessionId` is only unique within a retailer for this consumer.

The operations described in the following sections are all inherited into this interface from the `i_UserAccessSessionInfo` interface, which supports the generic user-provider roles. No Ret RP specific specialisations are defined for this interface.

The following operation signatures are taken from the module `TINAUserAccess`. All unscoped types need to scoped by `TINAUserAccess::` when used by clients of the `i_ConsumerAccessSessionInfo` interface.

4.4.1.5.1. newAccessSessionInfo()

```
oneway void newAccessSessionInfo (
      in TINAAccessCommonTypes::t_AccessSessionInfo accessSession
      );
```

This operation is used to inform the consumer that a new access session has been established.

t_AccessSessionInfo contains the t_AccessSessionId of the new access session, the t_UserCtxtName so the consumer can tell which consumer domain/terminal the access session has been established, and t_AccessSessionProperties which are a retailer specific property list that can be used to provide more information on the access session.

4.4.1.5.2. endAccessSessionInfo()

```
oneway void endAccessSessionInfo (
      in TINAAccessCommonTypes::t_AccessSessionId asId
      );
```

This operation is used to inform the consumer that an access session has ended.

t_AccessSessionId identifies which access session has ended.

4.4.1.5.3. cancelAccessSessionInfo()

```
oneway void cancelAccessSessionInfo (
      in TINAAccessCommonTypes::t_AccessSessionId asId
      );
```

This operation is used to inform the consumer that an access session has been cancelled by the retailer. See Section 4.4.1.2.1, "cancelAccessSession()" for details. t_AccessSessionId identifies which access session has been cancelled.

4.4.1.5.4. newSubscribedServicesInfo()

```
oneway void newSubscribedServicesInfo (
      in TINAAccessCommonTypes::t_ServiceList services
      );
```

This operation is used to inform the consumer that they have been subscribed to some new services. (The consumer may have subscribed to the services through a service in this, or another access session, or a consumer may have subscribed his users to a new service.)

t_ServiceList is a list of the services that the user has subscribed to. (It is a sequence of t_ServiceInfo structures, see Section 4.3.4, "Service and Session Information".)

4.4.1.6 i_ConsumerSessionInfo Interface

```
// module TINARetConsumerAccess

interface i_ConsumerSessionInfo
                        : TINAUserAccess::i_UserSessionInfo
{
};
```

This interface allows the retailer to inform the consumer of changes of state in service sessions which the consumer is involved in. Information operations are invoked whenever a change to the service session affects the consumer, (i.e. the session is suspended), but not when the change does not affect the consumer, (i.e. another party in the session leaves). This interface is informed of changes in all service sessions involving the consumer, and not just those associated with this access session.

This interface can be passed to the retailer, as part of `setUserCtxt()` on `i_RetailerNamedAccess` interface. If the consumer does NOT wish to be informed of changes in their service sessions, then this interface does NOT need to be passed in `setUserCtxt()`. (If it is not passed, the consumer can still register this interface, using `registerInterface()` on `i_RetailerNamedAccess`. Then the retailer will tell the consumer about service session changes, until this interface is unregistered, or the current access session ends.)

If the consumer wishes to be informed of service session changes outside of an access session, then they must register this interface, using `registerInterfaceOutsideAccessSession()` on `i_RetailerNamedAccess`. The operations do not include a `t_ProviderId`, so if this interface is registered for use outside an access session, a separate interface must be registered with each retailer. Retailers can not share this interface, because `t_SessionId` is only unique within a retailer for this consumer.

The operations described in the following sections are all inherited into this interface from the `i_UserSessionInfo` interface, which supports the generic user-provider roles. No Ret RP specific specialisations are defined for this interface.

The following operations are invoked when an action concerning this consumer is performed by the service session. (They correspond to info operations invoked by the service session on a user application. These info operations can be found the usage part of Ret RP, within the BasicFS (Section 5.5.2) and MultipartyFS (Section 5.5.5) feature sets.)

Only actions associated with this consumer produce info operations, i.e. consumer A receives a `endMyParticipationInfo()` invocation if they end their participation in a session, but do not receive any info if another consumer B ends their own participation. If B were to end A's participation, then A would receive the info.

All `i_ConsumerSessionInfo` interfaces receive info invocations when an action in a service session occurs. Usually one of these interfaces will be registered through each access session. It does not matter in which access session the service session is being used, all `i_ConsumerSessionInfo` interfaces will receive an info invocation.

The following operation signatures are taken from the module `TINAUserAccess`. All unscoped types need to scoped by `TINAUserAccess::` when used by clients of the `i_ConsumerSessionInfo` interface.

```
oneway void newSessionInfo (
      in TINAAccessCommonTypes::t_SessionInfo session
      );
```

- The consumer has started a new service session. `session` contains information about the new session that has been started.

```
oneway void endSessionInfo (
      in TINACommonTypes::t_SessionId sessionId
);
```

- A service session has been ended. `sessionId` identifies the ended session.

```
oneway void endMyParticipationInfo (
      in TINACommonTypes::t_SessionId sessionId
);
```

- The consumer's participation in a service session has been ended. `sessionId` identifies the session.

```
oneway void suspendSessionInfo (
      in TINACommonTypes::t_SessionId sessionId
      );
```

- A service session has been suspended. `sessionId` identifies the session.

```
oneway void suspendMyParticipationInfo (
      in TINACommonTypes::t_SessionId sessionId
      );
```

- The consumer's participation in service session has been suspended. `sessionId` identifies the session.

```
oneway void resumeSessionInfo (
      in TINAAccessCommonTypes::t_SessionInfo session
      );
```

- A suspended service session has been resumed. `sessionId` identifies the session. (The consumer may or may not have re-joined the service session, depending on whether they or another consumer resumed the session). `session` contains information about the session in which has been resumed.

```
oneway void resumeMyParticipationInfo (
      in TINAAccessCommonTypes::t_SessionInfo session
      );
```

- The consumer's participation in service session has been resumed. `session` contains information about the session in which the consumer has resumed their participation.

```
oneway void joinSessionInfo (
      in TINAAccessCommonTypes::t_SessionInfo session
      );
```

- The consumer has joined a service session. `session` contains information about the session that the consumer has joined.

4.4.2 Retailer Domain Interfaces

The following are the interfaces defined for the retailer domain of the Ret RP.

4.4.2.1 i_RetailerInitial Interface

```
// module TINARetRetailerInitial

interface i_RetailerInitial:

    TINAProviderInitial::i_ProviderInitial
{
     // Inherited operations shown in following subsections.
};
```

The `i_RetailerInitial` interface is a consumer's initial contact point with the retailer. It allows the consumer to request an access session is established between himself and the retailer.

This interface is returned when the consumer contacts the retailer. Ret-RP does not specify how the consumer contacts the retailer. Some examples could be: through the DPE naming service; through another type of directory service, such as a trader; through the TINA Broker business domain and Bkr reference point; or through a URL and retailer home page. An interface of this type is returned to the consumer as part of this contact the retailer scenario.

This interface inherits from `i_ProviderInitial` interface. It defines all of the operations which are generic to access user-provider roles, and can be re-used in other inter-domain reference points.

This interface has a role in security, and may use DPE security for message encryption, and domain authentication. That is message passing through the DPE is protected through encryption to varying, agreed levels and that both domain's credentials are exchanged for authentication. However, it does not mandate that authentication and credential acquisition occurs through the DPE, and so provides the `i_RetailerAuthenticate` interface to allow authetication of the user, outside of DPE security. A reference to the `i_RetailerAuthenticate` interface is passed to the consumer domain by the `requestNamedAccess()` and `requestAnonymousAccess()` operations if the user is not authenicated by DPE security.

The following operation signatures are taken from the module `TINAProviderInitial`. All unscoped types need to scoped by `TINAProviderInitial::` when used by clients of the `i_RetailerInitial` interface.

4.4.2.1.1. requestNamedAccess()

```
void requestNamedAccess (
      in TINACommonTypes::t_UserId userId,
      in TINACommonTypes::t_UserProperties userProperties,
      out Object  namedAccessIR,  // type: i_ProviderNamedAccess
      out TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      out TINAAccessCommonTypes::t_AccessSessionId asId
) raises (
      e_AccessNotPossible,
      e_AuthenticationError,
      TINAAccessCommonTypes::e_UserPropertiesError
);
```

The `requestNamedAccess`() allows the consumer to identify himself and request the establishment of an access session with the retailer. The access session provides access to use his subscribed services, etc, through a `i_RetailerNamedAccess` interface.

If CORBA security services are being used by both the consumer and retailer domains' DPEs, then both domain's credentials and other authentication information will be exchanged through the DPE before this operation is invoked on the retailer. This means a secure context for messages may have already been set-up between the domains, and the identify of the consumer will have been authenticated. In this case, an access session is established, and a reference to the `i_RetailerNamedAccess` interface will be returned. Along with this, an `t_AccessSessionSecretId` is returned, to be used in all requests on the new interface.

If CORBA security services are not being used, then no secure context for messages will have been set-up, and DPE messages could potentially be intercepted and read by third-parties.

If the consumer has not already been authenticated, and the DPE is unable to perform the authentication and establish an access session when this operation is invoked, then the operation will fail. An `e_AuthenticationError` exception will be raised, which contains a reference to a `i_RetailerAuthenticate` interface. This interface may be used to authenticate and set-up the secure context. Then this operation can be invoked again to establish the access session.

`userId` identifies the consumer to the retailer. For details on the structure of the `userId`, see Section 3.3.2, "User Information".

`userProperties` are a sequence of user properties associated with this consumer. In general the consumer would not send sensitive information to the retailer until an access session has been established. However this parameter can be used to pass the consumer's password to the retailer, if both domains use DPE security to encrypt the messages. Security context, and other information which is understood by the specific retailer can also be sent. For more details, see Section 3.3.2, "User Information".

If the request is successful, and the consumer has been authenticated, then the following out parameters are returned:

`namedAccessIR` is the reference to the `i_RetailerNamedAccess` interface, which the consumer domain uses during the access session. NOTE: Although the IDL specifies the type (in text) as `i_ProviderNamedAccess`, it is only to state the base reference type. An abstract interface (reference) is never exported over a reference point. The `requestNamedAccess()` operation is defined inside the `i_ProviderNamedAccess` interface which is later inherited into `i_RetailerNamedAccess` and thus uses interface (references) of type `i_Retailer<...>`. The reason for stating the base reference type in the IDL is to allow re-use in other reference point definitions. For that matter, the `namedAccessIR` could just as well be of i_3ptyNamedAccess type.

`asSecretId` is an `t_AccessSessionSecretId` used whenever the consumer domain invokes an operation on the `namedAccessIR` within this access session. The `asSecretId` identifies the consumer domain from which invocations on `namedAccessIR` are made. This parameter should be used during this access session only , and only by the consumer domain to which it was returned. See Section 4.3.1, "Access Session Information".

`asId` is an `t_AccessSessionId` used to identify this access session. It is available to all the access sessions for this consumer. It can be used identity this access session when making requests on any `i_RetailerNamedAccess` interface between this consumer and retailer, e.g. using `listServiceSessions()`, an `t_AccessSessionId` can be used to scope the list to those started from a specific access session.

If the request is unsuccessful, either the consumer has not been authenticated, or the authentication has failed.

An e_AccessNotPossible exception is raised if the retailer is unable, or refuses to allow the consumer domain to establish an access session with them.

An e_AuthenticationError exception is raised if the retailer has not authenticated the consumer. This contains a list of authentication methods that can be used with the i_RetailerAuthenticate interface. The interface is returned as either an interface reference, or a stringified object reference, depending on the retailer. This reference is used to authenticate the consumer with the retailer. Once the consumer has been successfully authenticated, (using one of the authentication methods indicated), then the consumer can call this operation again to request the establishment of an access session, and get a reference to the i_RetailerNamedAccess interface.

If an e_UserPropertiesError exception is raised, then there is a problem with the userProperties. The errorCode provides the reason for the error.

4.4.2.1.2. requestAnonymousAccess()

```
void requestAnonymousAccess (
      in TINACommonTypes::t_UserProperties userProperties,
      out Object anonAccessIR,     // type: i_ProviderAnonAccess
      out TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      out TINAAccessCommonTypes::t_AccessSessionId asId
) raises (
      e_AccessNotPossible,
      e_AuthenticationError
      TINAAccessCommonTypes::e_UserPropertiesError
);
```

The requestAnonymousAccess() allows the consumer to request the establishment of an access session with the retailer. It is used when the consumer does not have a user identity with the retailer. This may be because they have not previously contacted this retailer, or they wish to remain anonymous to this retailer.

This operation returns a reference to a i_RetailerAnonAccess interface, through which the consumer can access services, and register as a named user with the retailer, if they wish to do so.

If CORBA security services are being used by both the consumer and retailer domains' DPEs, then both domain's may exchange credentials through the DPE before this operation is invoked on the retailer. This means a secure context for messages may have already been set-up between the domains, but the credentials will not contain any information about the identity of the specific consumer.

If CORBA security services are not being used, then no secure context for messages will have been set-up, and DPE messages could potentially be intercepted and read by third-parties.

userProperties are a sequence of user properties associated with this consumer. They may contain security context and other information which is understood by the specific retailer. For more details, see Section 3.3.2, "User Information".

If the request is successful, an access session has been established with the consumer. The following out parameters are returned:

anonAccessIR is the reference to the i_RetailerAnonAccess interface, which the consumer domain uses during the access session.

asSecretId is an t_AccessSessionSecretId used whenever the consumer domain invokes an operation on the namedAccessIR within this access session. The asSecretId identifies the consumer domain from which invocations on namedAccessIR are made. This parameter should be used during this access session only , and only by the consumer domain to which it was returned. See Section 4.3.1, "Access Session Information".

asId is an t_AccessSessionId used to identify this access session. It is available to all the access sessions for this consumer. It can be used identity this access session when making requests on any i_RetailerNamedAccess interface in an access session between this consumer and retailer. (In general, anonymous users can only have one access session with the retailer, as each access session with each anonymous user must be treated separately. Since the consumers are anonymous to the retailer, each consumer appears to be a separate individual, even if they are, in fact, the same person.)

If the request is unsuccessful, either the consumer has not been authenticated, or the authentication has failed.

An e_AccessNotPossible exception is raised if the retailer is unable, or refuses to allow the consumer domain to establish an access session with them.

An e_AuthenticationError exception is raised if the retailer requires that the consumer domain is authenticated. This contains a list of authentication methods that can be used with the i_RetailerAuthenticate interface. (Authentication methods may authenticate the domains only, and not the specific consumer). The interface is returned as either an interface reference, or a stringified object reference, depending on the retailer. This reference is used to authenticate the consumer domain with the retailer. Once the consumer domain has been successfully authenticated, (using one of the authentication methods indicated), then the consumer can call this operation again to request the establishment of an access session, and get a reference to the i_RetailerAnonAccess interface.

If an e_UserPropertiesError exception is raised, then there is a problem with the userProperties. The errorCode provides the reason for the error.

4.4.2.2 i_RetailerAuthenticate Interface

```
// module TINARetRetailerInitial

interface i_RetailerAuthenticate:
        TINAProviderInitial::i_ProviderAuthenticate
{
      // Inherited operations shown in following subsections.
};
```

The i_RetailerAuthenticate interface allows the consumer and the retailer to be authenticated. It provides a generic mechanism for authentication which can be used to support a number of different authentication protocols.

The purpose of this interface is to verify to the consumer and retailer that each domain is interacting with the domain they have been told they are talking to. This mutual authentication of both domains. Other authentication schemes which authenticate only one of the domains is also possible using this

interface. The interface provides a set of generic operations, that can be used in authentication. However, the operations only provide a mechanism for 'transporting' authentication information. Both domains must know and use a common authentication protocol, and perform this protocol using these operations in order to authenticate the domains. Ret-RP does not specify any particular authentication protocol. The `getAuthenticationMethods()` operation on this interface can be used to determine the authentication protocols supported by the retailer, and a protocol chosen for authentication. The authentication protocol may, or may not identify the individual consumer. It may only identy and authenticate the consumer's domain.

The following operation signatures are taken from the module `TINAProviderInitial`. All unscoped types need to scoped by `TINAProviderInitial::` when used by clients of the `i_RetailerInitial` interface.

4.4.2.2.1. getAuthenticationMethods()

```
void getAuthenticationMethods {
      in  t_AuthMethodSearchProperties desiredProperties,
      out t_AuthMethodDescList authMethods
) raises (
      e_AuthMethodPropertiesError,
      TINACommonTypes::e_ListError
);
```

The `getAuthenticationMethods()` allows the consumer to ask the retailer for a list of the authentication methods supported. A particular authentication method can then be chosen by the consumer to use in `authenticate()`.

`desiredProperties` is a list containing the properties that the consumer wishes the authentication method to support. (See `t_MatchProperties` in Section 3.3.1). For example, the consumer can request that the authentication methods returned support mutual authentication, or retailer authentication only. Currently no specific property names and values have been defined for `t_AuthMethodSearchProperties`, and so its use is retailer specific.

`authMethods` is a list of authentication methods which match the `desiredProperties`, and which the retailer supports. The `t_AuthMethodDesc` structure contains the authentication method identifier, and a list of properties of the method. It is assumed that both the consumer and retailer both know the protocol to follow in order to use the authentication method defined.

The `authMethods` list may be empty. This may occur if the retailer does not support any methods matching the properties requested, or if the retailer does not wish to allow the consumer to authenticate using a method with the desired properties. e.g. if the consumer requests a method for retailer only authentication, and the retailer wishes to have mutual authentication.

If the `desiredProperties` parameter is wrongly formatted, or provides an invalid property name or value, the `e_PropertyError` exception should be raised. (Property names which are not recognised can be ignored, if `desiredProperties` requires that only some, or none of the properties are matched.)

If the `authMethods` list is unavailable, then raise an `e_ListError` exception with the `ListUnavailable` error code.

4.4.2.2.2. authenticate()

```
void authenticate(
      in t_AuthMethod authMethod,
```

```
        in string securityName,
        in t_opaque authenData,
        in t_opaque privAttribReq,
        out t_opaque privAttrib,
        out t_opaque continuationData,
        out t_opaque authSpecificData,
        out t_AuthenticationStatus authStatus
    ) raises (
        e_AuthMethodNotSupported
    );
```

`authenticate`() allows the consumer to select an authentication method, pass authentication data to the retailer.

Once the consumer domain has determined an authentication method with the retailer, this operation is used to transport authentication data, and other credentials to the retailer. This data is used to perform the type of authentication appropriate to the authentication method, (this may be mutual authentication, or authentication of the consumer/retailer domain only, etc.).

The retailer then returns its authentication data (if required), challenge data for the consumer to respond using `continueAuthentication`() (if required), and the requested credentials (if possible). If further authentication protocol is required before credentials are returned then these can be returned by `continueAuthentication()`.

The following parameters are sent by the consumer to the retailer:

`authMethod` is used to identify the authentication method proposed by the consumer. It affects the composition and generation method of the other opaque data parameters. Currently no specific authentication methods values have been defined for `t_AuthMethod`, and so its use is retailer specific.

`securityName` is the name assumed by consumer for authentication. It may be an empty string according to the authentication method used.

`authenData` is opaque data containing consumer attributes to be authenticated. Its format depends upon the authentication method used.

`privAttribReq` is opaque data which is used to specify the rights and privileges which the consumer domain requests from the retailer domain. This data may correspond to levels of security to access different areas of the retailer domain. Its format depends upon the authentication method used.

The following parameters are returned by the retailer to the consumer:

`privAttrib` is opaque data which defines the privilege attributes granted to the consumer, based upon the `privAttribReq`, and their authentication data. Its format depends upon the authentication method used.

`continuationData` is opaque data which is used to challenge the consumer. The consumer has not yet been authenticated, and must process this data and return the result to the retailer using the `continueAuthentication()` operation. Its format depends upon the authentication method used. This parameter may be ignored if the value of `authStatus` is not `SecAuthContinue`.

`authSpecificData` is opaque data which is specific to the authentication method used.

`authStatus` identifies the status of the authentication process. It is an enumerated type with the following values:

- `SecAuthSuccess`
  Authentication has completed successfully. No (more) calls to `continueAuthentication()` are necessary. The consumer can call `requestNamedAccess()` on `i_RetailerInitial` interface to gain a reference to the `i_RetailerNamedAccess` interface. (Or if they wish to be an anonymous user, call `requestAnonymousAccess()` for a `i_RetailerAnonAccess`interface.)
- `SecAuthFailure`
  Authentication has completed unsuccessfully. The consumer has not been authenticated, and will not be able to establish an access session. Calls to `requestNamedAccess()` will continue to raise an `e_AccessNotPossible`, or `e_AuthenticationError` exception.
- `SecAuthContinue`
  Authentication is continuing, and the consumer must reply to this result by calling `continueAuthentication()`.
- `SecAuthExpired`
  Authentication has timed out. The consumer did not make this invocation of `continueAuthentication`() quickly enough, after the reply from `authenticate`(), or the previous call to `continueAuthentication`(). Authentication must be started again from the beginning by calling `authenticate`(). (This enumeration should not be returned by `authenticate`().

4.4.2.2.3. continueAuthentication()

```
void continueAuthentication{
      in t_opaque responseData,
      out t_opaque privAttrib,
      out t_opaque continuationData,
      out t_opaque authSpecificData,
      out t_AuthenticationStatus authStatus
);
```

`continueAuthentication`() allows the consumer to continue an authentication protocol, started using `authenticate`(), and pass authentication data to the retailer.

This operation should be invoked by the consumer if the `authStatus` returned from `authenticate`(), or a previous call to `continueAuthentication`(), is `SecAuthContinue`. The `authStatus` is used by both operations to indicate if the consumer needs to make another call to this operation. Parameters returned by this operation must be processed by the consumer according to the authentication method, and the results provided as in parameters to the subsequent call to this operation.

`responseData` is opaque data from the consumer. This data has been generated by the consumer according to the authentication method, based on the `continuationData` returned by the previous call to `authenticate`() or `continueAuthentication`(). Precisely how this data is generated, and formatted is specific to the authentication method used.

`continuationData` is opaque data which is used to challenge the consumer. The consumer has not yet been authenticated, and must process this data and return the result to the retailer using the `continueAuthentication()` operation. Its format depends upon the authentication method used. This parameter may be ignored if the value of `authStatus` is not `SecAuthContinue`.

`authSpecificData` is opaque data which is specific to the authentication method used.

`authStatus` identifies the status of the authentication process. It has the same values as for `authenticate`().

4.4.2.3 i_RetailerAccess Interface

```
// module TINAProviderAccess

interface i_RetailerAccess
{
};
```

`i_RetailerAccess` interface is an abstract interface, used to inherit common operations in to the `i_RetailerNamedAccess`, and `i_RetailerAnonAccess` interfaces.

The purpose of this interface is for inheritance, as described above. It should not be available over the Ret RP. No instances of this interface type should be created.

Currently no operations are defined for this interface. It will be contain operations which are shared between the `i_RetailerNamedAccess`, and `i_RetailerAnonAccess` interfaces. Currently all operations are defined on the `i_RetailerNamedAccess` interface, and no operations have been identified for the `i_RetailerAnonAccess` interface.

4.4.2.4 i_RetailerNamedAccess Interface

```
// module TINAProviderAccess

interface i_RetailerNamedAccess
        : i_ProviderNamedAccess, i_RetailerAccess
{
// Inherited operations shown in following subsections.
};
```

`i_RetailerNamedAccess` interface allows a known consumer access to his subscribed services. The consumer uses it for all operations within an access session with the retailer.

This interface is returned when the consumer has been authenticated by the retailer and an access session has been established. It is returned by calling `requestNamedAccess`() on the `i_RetailerInitial` interface.

This interface inherits from `i_ProviderNamedAccess` and `i_RetailerAccess` interfaces. `i_ProviderNamedAccess` defines all of the operations which are generic to access user-provider roles, and can be re-used in other inter-domain reference points. All the operations on this interface are inherited from there. The `i_RetailerAccess` interface is currently blank. It will be contain operations which are shared between the `i_RetailerAnonAccess` interface, and this interface, that are specific to the Ret RP.

The following operation signatures are taken from the module `TINAProviderAccess`. All unscoped types need to scoped by `TINAProviderAccess::` when used by clients of the `i_RetailerAccess` interface.

4.4.2.4.1. setUserCtxt()

```
void setUserCtxt (
      in TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in t_UserCtxt userCtxt
) raises (
      TINAAccessCommonTypes::e_AccessError,
      e_UserCtxtError
);
```

The `setUserCtxt()` allows the consumer to inform the retailer about interfaces in the consumer domain, and other consumer domain information. (e.g. user applications available in the consumer domain, operating system used, etc).

`userCtxt` is a structure containing consumer domain configuration information and interfaces.

This operation should be called immediately after receiving the reference to this interface. If this operation has not been called successfully, subsequent operations may raise an `e_AccessError` exception with a `UserCtxtNotSet` error code.

If there is a problem with `userCtxt`, then `e_UserCtxtError` should be raised with the appropriate error code.

4.4.2.4.2. getUserCtxt()

```
void getUserCtxt (
      in TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in TINACommonTypes::t_UserCtxtName ctxtName,
      out t_UserCtxt userCtxt
) raises (
      TINAAccessCommonTypes::e_AccessError,
      e_UserCtxtError
);
```

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

`ctxtName` is the name of the context that the consumer wishes to retrieve user context information about. (`ctxtName` is set by the consumer when registering a user context, and is the consumer term for the context, e.g. “Home”, “Work”, “Mum’s House”, etc.

`userCtxt` is a structure containing consumer domain configuration information and interfaces.

4.4.2.4.3. getUserCtxts()

```
void getUserCtxts (
      in TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in t_SpecifiedUserCtxt ctxt,
      out t_UserCtxtList userCtxts
) raises (
      TINAAccessCommonTypes::e_AccessError,
      e_UserCtxtError,
      TINACommonTypes::e_ListError
);
```

4.4.2.4.4. listAccessSessions()

```
void listAccessSessions (
      in TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      out TINAAccessCommonTypes::t_AccessSessionList asList
) raises (
      TINAAccessCommonTypes::e_AccessError,
      TINACommonTypes::e_ListError
);
```

The `listAccessSessions()` returns a list of access sessions. The list contains all the access sessions the consumer currently established with this retailer. It is a `sequence` of `t_AccessSessionInfo` structures, which consist of the `t_AccessSessionId`, `t_UserCtxtName`, and `t_AccessSessionProperties`. The last of these is a `t_PropertyList`. Currently no specific property names and values have been defined for `t_AccessSessionProperties`, and so its use is retailer specific.

The information returned by this operation can be used by the consumer to found out which other access sessions are currently established; end some of those access sessions (see `endAccessSession()`); list the service sessions of those access sessions (see `listServiceSessions()`); and be informed of changes to those access sessions and service sessions (see `i_ConsumerAccessSessionInfo` and `i_ConsumerSessionInfo` interfaces).

If the `asList` list is unavailable, because the consumer's access sessions are not available, then the operation should raise an `e_ListError` exception with the `ListUnavailable` error code.

4.4.2.4.5. endAccessSessions()

```
void endAccessSession(
      in TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in TINAAccessCommonTypes::t_SpecifiedAccessSession as,
      in t_EndAccessSessionOption option
) raises (
      TINAAccessCommonTypes::e_AccessError,
      TINAAccessCommonTypes::e_SpecifiedAccessSessionError,
      e_EndAccessSessionError
);
```

The `endAccessSession()` allows the consumer to end an 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 `t_SpecifiedAccessSession` parameter.

`t_EndAccessSessionOptions` allows the consumer to choose the actions the retailer should take, if there are active or suspended service sessions, when the access session ends. The actions are only used as part of this invocation. The retailer does not remember the action chosen. (Retailers may define a default policy for service sessions when a consumer ends the access session in which they were created, or allow the consumer to define the policy. Currently, Ret RP does not support the definition of such a policy by the consumer.)

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

e_EndAccessSessionError is raised if option is invalid, or service sessions remain active, or suspended, which are not allowed by the retailer. (A consumer may end an access session, leaving active or suspended sessions if this is allowed as a policy of the retailer for this consumer.)

4.4.2.4.6. getUserInfo()

```
void getUserInfo(
      in TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      out TINAAccessCommonTypes::t_UserInfo userInfo
) raises (
      TINAAccessCommonTypes::e_AccessError
);
```

The getUserInfo() allows the consumer to request information about himself.

This operation returns a t_UserInfo structure as an out parameter. This contains the consumer's t_UserId, their name, and a list of user properties. Currently no specific property names and values have been defined for t_UserProperties, and so its use is retailer specific.

4.4.2.4.7. listSubscribedServices()

```
void listSubscribedServices (
      in TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in t_SubscribedServiceProperties desiredProperties,
      out TINAAccessCommonTypes::t_ServiceList services
) raises (
      TINAAccessCommonTypes::e_AccessError,
      TINACommonTypes::e_PropertyError,
      TINACommonTypes::e_ListError
);
```

The listSubscribedServices() returns a list of the services to which the consumer has previously been subscribed.

The desiredProperties parameter can be used to scope the list of subscribed services. t_SubscribedServiceProperties identifies the properties which the suscribed services must match. It also defines whether a subscribed service must match one, all or none of the properties. (See t_MatchProperties in Section 3.3.1). Currently no specific property names and values have been defined for t_SubscribedServiceProperties, and so its use is retailer specific.

The list of services subscribed to by the consumer, and matching the desiredProperties, is returned in the t_ServiceList. This is a sequence of t_ServiceInfo structures, which contain the t_ServiceId, t_UserServiceName (consumers name for the service), and a sequence of service properties, t_ServiceProperties. Currently no specific property names and values have been defined for t_ServiceProperties, and so its use is retailer specific.

If the desiredProperties parameter is wrongly formatted, or provides an invalid property name or value, the e_PropertyError exception should be raised. (Property names which are not recognised can be ignored, if desiredProperties 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 e_ListError exception with the ListUnavailable error code.

4.4.2.4.8. discoverServices()

```
void discoverServices(
      in TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in t_DiscoverServiceProperties desiredProperties,
      in unsigned long howMany,
      out TINAAccessCommonTypes::t_ServiceList services,
      out Object iteratorIR       // type: i_DiscoverServicesIterator
) raises (
      TINAAccessCommonTypes::e_AccessError,
      TINACommonTypes::e_PropertyError,
      TINACommonTypes::e_ListError
);
```

The discoverServices() returns a list of the services available from this retailer.

This operation is used to discover the services provided by the retailer, for use by the consumer. It can be used to retrieve information on all of the services provided, or be scoped by the desiredProperties parameter. (See t_MatchProperties in Section 3.3.1).

The list of retailer services matching the desiredProperties is returned in services. This is a sequence of t_ServiceInfo structures, which contain the t_ServiceId, t_UserServiceName (consumers name for the service), and a sequence of service properties, t_ServiceProperties. Currently no specific property names and values have been defined for t_ServiceProperties, and so its use is retailer specific.

The howMany parameter defines the number of t_ServiceInfo structures to return in the services parameter. The length of services will not exceed this number. Any remaining services which match the desiredProperties, but which aren't included in services are accessible through iteratorIR, the i_DiscoverServicesIterator interface. If there are no remaining services, then iteratorIR should be null.

If the desiredProperties parameter is wrongly formatted, or provides an invalid property name or value, the e_PropertyError exception should be raised. (Property names which are not recognised can be ignored, if desiredProperties 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 e_ListError exception with the ListUnavailable error code.

4.4.2.4.9. getServiceInfo()

```
void getServiceInfo (
      in TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in TINAAccessCommonTypes::t_ServiceId serviceId,
      in TINAProviderAccess::t_SubscribedServiceProperties
               desiredProperties,
      out TINAAccessCommonTypes::t_ServiceProperties serviceProperties
) raises (
      TINAAccessCommonTypes::e_AccessError,
      TINAProviderAccess::e_ServiceError
);
```

The getServiceInfo() returns information on a specific service, identified by the serviceId. The desiredProperties list can scope the information which is requested to be returned.

4.4.2.4.10. listServiceSessions()

```
void listServiceSessions (
      in TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in TINAAccessCommonTypes::t_SpecifiedAccessSession as,
      in t_SessionSearchProperties desiredProperties,
      out TINAAccessCommonTypes::t_SessionList sessions
) raises (
      TINAAccessCommonTypes::e_AccessError,
      TINAAccessCommonTypes::e_SpecifiedAccessSessionError,
      TINACommonTypes::e_PropertyError,
      TINACommonTypes::e_ListError
);
```

The `listServiceSessions()` returns a list of the service sessions, which the consumer is involved in. This includes active and suspended sessions.

The `as` 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 `desiredProperties` parameter can be used to scope the list of sessions. `t_SessionSearchProperties` identifies the properties which the sessions must match. It also defines whether a session must match one, all or none of the properties. (See `t_MatchProperties` in Section 3.3.1). The following property names and values have been defined for `t_SessionSearchProperties`:

- name: "`SessionState`"
  value: `t_SessionState`
  If a property in `t_SessionSearchProperties` has the name "`SessionState`", then the session must have the same `t_SessionState` as given in the propery value.
- name: "`UserSessionState`"
  value: `t_UserSessionState`
  If a property in `t_SessionSearchProperties` has the name "`UserSessionState`", then the session must have the same `t_UserSessionState` as given in the propery value.

Other retailer specific properties can also be defined in `desiredProperties`.

The list of sessions matching the `desiredProperties` and the `accessSession` are returned in `sessions`. This is a `sequence` of `t_SessionInfo` structures, which define the `t_SessionId`, `t_ParticipantSecretId`, `t_PartyId`, `t_UserSessionState`, `t_InterfaceList`, `t_SessionModelList`, and `t_SessionProperties` of the session.

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

If the desiredProperties parameter is wrongly formatted, or provides an invalid property name or value, the e_PropertyError exception should be raised. (Property names which are not recognised can be ignored, if desiredProperties requires that only some, or none of the properties are matched.)

If the sessions list is unavailable, because the consumer's sessions are not known, then the operation should raise an e_ListError exception with the ListUnavailable error code.

4.4.2.4.11. getSessionModels()

```
void getSessionModels (
      in TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in TINACommonTypes::t_SessionId sessionId,
      out TINACommonTypes::t_SessionModelList sessionModels
) raises (
      TINAAccessCommonTypes::e_AccessError,
      e_SessionError,
      TINACommonTypes::e_ListError
);
```

The getSessionModels() returns a list of the session models supported by a service session. It can be used on active and suspended sessions.

sessionId identifies the session whose session models are retrieved.

sessionModels are the session models supported by the session. It is a sequence of t_SessionModel structures, which contain the name of the session model, and a list of properties for that session model. For the "TINA Session Model", a number of properties have been defined, (see ??Sesscion Model section in the Usage part of Ret RP??) including a list of the feature sets supported by the session.

e_SessionError is raised if the sessionId is invalid; or the session state precludes access to the session models (e.g. the session is suspended); or the session refuses to return sessionModels.

If the sessionModels list is unavailable, because the session models supported by the session are not known, then the operation should raise an e_ListError exception with the ListUnavailable error code.

4.4.2.4.12. getSessionInterfaceTypes()

```
void getSessionInterfaceTypes (
      in TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in TINACommonTypes::t_SessionId sessionId,
      out TINACommonTypes::t_InterfaceTypeList itfTypes
) raises (
      TINAAccessCommonTypes::e_AccessError,
      e_SessionError,
      TINACommonTypes::e_ListError
);
```

The getSessionInterfaceTypes() returns a list of the interface types supported by a service session. It can be used on active and suspended sessions.

sessionId identifies the session whose interface types are retrieved.

itfTypes are all the interface types supported by the session. It is a sequence of t_InterfaceTypeName's, which are strings representing the interface types supported by the session. itfTypes should include all the interface types that can be supported by the session.

e_SessionError is raised if the sessionId is invalid; or the session state precludes access to the session models (e.g. the session is suspended); or the session refuses to return itfTypes.

If the itfTypes list is unavailable, because the interface types supported by the session are not known, then the operation should raise an e_ListError exception with the ListUnavailable error code.

4.4.2.4.13. getSessionInterface()

```
void getSessionInterface (
      in TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in TINACommonTypes::t_SessionId sessionId,
      in TINACommonTypes::t_InterfaceTypeName itfType,
      out TINACommonTypes::t_InterfaceStruct itf
) raises (
      TINAAccessCommonTypes::e_AccessError,
      e_SessionError,
      TINACommonTypes::e_InterfacesError
);
```

The getSessionInterface() returns an interface, of the type requested, supported by a service session. It can be used on active sessions.

sessionId identifies the session whose interface are retrieved.

itfType identifies the interface type of the interface reference to be returned.

itf is returned by this operation. It contains the t_InterfaceTypeName, an interface reference (t_IntRef) and the interface properties (t_InterfaceProperties) of the interface type requested.

e_SessionError is raised if the sessionId is invalid; or the session state precludes access to the session models (e.g. the session is suspended); or the session refuses to return itfTypes.

If the session does not support interfaces of itfType, then the operation should raise the e_SessionInterfacesError, with the InvalidSessionInterfaceType error code.

4.4.2.4.14. getSessionInterfaces()

```
void getSessionInterfaces (
      in TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in TINACommonTypes::t_SessionId sessionId,
      out TINACommonTypes::t_InterfaceList itfs
) raises (
      TINAAccessCommonTypes::e_AccessError,
      e_SessionError,
      TINACommonTypes::e_ListError
);
```

The getSessionInterfaces() returns a list of all the interfaces supported by a service session. It can be used on active sessions only.

`sessionId` identifies the session whose interface types are retrieved.

`itfs` is returned by this operation. It is a `sequence` of `t_InterfaceStruct` structures which contain the `t_InterfaceTypeName`, an interface reference (`t_IntRef`) and the interface properties (`t_InterfaceProperties`) of each interface.

`e_SessionError` is raised if the `sessionId` is invalid; or the session state precludes access to the session models (e.g. the session is suspended); or the session refuses to return `itfTypes`.

If the `itfs` list is unavailable, because the interface supported by the session are not known, then the operation should raise an `e_ListError` exception with the `ListUnavailable` error code.

4.4.2.4.15. listSessionInvitations()

```
void listSessionInvitations (
      in TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      out TINAAccessCommonTypes::t_InvitationList invitations
) raises (
      TINAAccessCommonTypes::e_AccessError,
      TINACommonTypes::e_ListError
);
```

The `listSessionInvitations()` returns a list of the invitations to join a service session, which have been sent to the consumer through this retailer.

`invitations` is returned by this operation. It is a `sequence` of `t_SessionInvitation` structures:

```
struct t_SessionInvitation {
      t_InvitationId id;
      t_UserId inviteeId;
      t_SessionPurpose purpose;
      t_InvitationReason reason;
      t_InvitationOrigin origin;
};
```

`id` identifies the particular invitation. It uniquely identifies this invitation from others for this consumer at this retailer. (Other consumers with this retailer may have invitations with the same id). This id is used in `joinSessionWithInvitation`() to join the session refered to by this invitation.

`inviteeId` is the user id of this consumer. (It is not necessary here, as the user id is known through the access session. It is included in this structure to allow invitations to be deliverable outside of an access session, and allow the receiptant to check that the invitation was for them.)

`purpose` is a string containing the purpose of the session.

`reason` is a string containing the reason this consumer has been invited to join this session.

`origin` is a structure containing the userId of the consumer that requested that the invitation was sent to this consumer, and their sessionId for the session that this consumer has been invited to join. (The sessionId is provided so that if the invited consumer contacts the inviting consumer, he is able to tell which session the invited consumer is refering to).

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

4.4.2.4.16. listSessionAnnouncements()

```
void listSessionAnnouncements (
      in TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in t_AnnouncementSearchProperties desiredProperties,
      out TINACommonTypes::t_AnnouncementList announcements
) raises (
      TINAAccessCommonTypes::e_AccessError,
      TINACommonTypes::e_PropertyError,
      TINACommonTypes::e_ListError
);
```

The `listSessionAnnouncements()` returns a list of the session announcements, which have been announced through this retailer.

Sessions can be announced due to requests from session participants (see Multiparty Feature Set), or due to properties of the session intialisation, service factory or policies of the user starting the service. The process by which sessions are announced is not defined by Ret-RP. However, this operation is provided in order to allow a consumer to request a list of sessions which have been announced. (Announcements may be scoped in order to restrict the distribution of the announcement to particular groups). This operation returns a list of announcements which match the `desiredProperties`, as specified by the consumer.

The `desiredProperties` parameter can be used to scope the list of announcements. `t_AnnouncementSearchProperties` identifies the properties which the announcements must match. (See `t_MatchProperties` in Section 3.3.1). Currently no specific property names and values have been defined for `t_AnnouncementSearchProperties`, and so its use is retailer specific.

`announcements` is a list of announcements available to the consumer, and matching the `desiredProperties`. This is a `sequence` of `t_SessionAnnouncement` structures, which contain the properties od the announcement, `t_AnnouncementProperties`. Currently no specific property names and values have been defined for `t_AnnouncementProperties`, and so its use is retailer specific.

If the `desiredProperties` parameter is wrongly formatted, or provides an invalid property name or value, the `e_PropertyError` exception should be raised. (Property names which are not recognised can be ignored, if `desiredProperties` requires that only some, or none of the properties are matched.)

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

4.4.2.4.17. startService()

```
void startService (
      in TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in TINAAccessCommonTypes::t_ServiceId serviceId,
      in t_ApplicationInfo app,
      in TINACommonTypes::t_SessionModelReq sessionModelReq,
      in t_StartServiceUAProperties uaProperties,
      in t_StartServiceSSProperties ssProperties,
      out TINAAccessCommonTypes::t_SessionInfo sessionInfo
) raises (
      TINAAccessCommonTypes::e_AccessError,
```

```
        e_ServiceError,
        e_ApplicationInfoError,
        TINACommonTypes::e_SessionModelError,
        e_StartServiceUAPropertyError,
        e_StartServiceSSPropertyError
    );
```

The `startService()` starts a service session for the consumer.

`serviceId` is the service type identifier, which indicates the service type of the session which the consumer wishes to start.

`app` is a structure containing information on the application, which will be used to interact with the service session. It includes: application name, version, serial number, property list, etc. It also includes: a list of interfaces supported by the application, which can optionally include references to some of those interfaces if they are available; a list of session models, and feature sets, again including interface references if appropriate; and a stream interface description list.

`sessionModelReq` defines the session models and feature sets that the consumer domain wishes the session to have. It allows the consumer to request that some, all or none of the session models are supported by the session.

`uaProperties` is a property list that will be interpreted by the retailer domain before the service session is started. No property names or values are defined, so it use it retailer-specific. Its purpose is to allow the consumer to define some preferences or other constraints that they wish to be applied to this service session only, and that the retailer needs to know before the session is started. (These properties may affect the choice of service factory for the session.)

`ssProperties` is a property list that will be interpreted by the service session, as soon as it has started. (i.e. before the references to the session are returned to the consumer domain). No property names or values are defined. Its use is entirely service specific, and only the service session is intended to interpret the properties given. (This parameter allows the consumer domain/application to pass service specific information to the service session, which is not intended for the retailer domain to interpret.)

`sessionInfo` is a structure, which contains information which allows the consumer domain to refer to this session using other operations on this interface. It also contains infomation for the usage part of the session, including the interface references to interact with the session. (See Section 4.3.4).

The following are exceptions which are raised by this operation:

`e_ServiceError` is raised if the `serviceId` is invalid/unknown by the retailer, or if a service session cannot be created.

`e_ApplicationInfoError` is raised if there are unknown or invalid values for `t_ApplicationInfo`, or if the application is incompatable with the type of service being started.

`e_SessionModelError` is raised if invalid session models and/or feature sets are required for the service session.

`e_StartServiceUAPropertyError` is raised if there is an error in the properties for `uaProperties`. It has the same properties error codes as `e_PropertyError`. (See Section 3.3.1 for more details.)

`e_StartServiceSSPropertyError` is raised if there is an error in the properties of `ssProperties`. It has the same properties error codes as `e_PropertyError`.

4.4.2.4.18. endSession()

```
void endSession (
      in TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in TINACommonTypes::t_SessionId sessionId
) raises (
      TINAAccessCommonTypes::e_AccessError,
      e_SessionError
);
```

The `endSession()` ends a service session for the consumer. It can be used to end sessions which the consumer is currently active in, and sessions which have been suspended, or the consumer has suspended his participation.

`sessionId` is the identifier of the session to be ended.

The exception `e_SessionError` is raised if `sessionId` is invalid; or the session refuses to end because of the user's session state; or the user does not have permission.

4.4.2.4.19. endMyParticipation()

```
void endMyParticipation (
      in TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in TINACommonTypes::t_SessionId sessionId
) raises (
      TINAAccessCommonTypes::e_AccessError,
      e_SessionError
);
```

The `endMyParticipation()` ends the consumer's participation in a service session. It can be used on a session which the consumer is currently active in, or which has been suspended, or the consumer has suspended his participation.

`sessionId` is the identifier of the session to end this user's participation.

The exception `e_SessionError` is raised if `sessionId` is invalid; or the session refuses to end this user's participation because of their session state; or the user does not have permission.

4.4.2.4.20. suspendSession()

```
void suspendSession (
      in TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in TINACommonTypes::t_SessionId sessionId
) raises (
      TINAAccessCommonTypes::e_AccessError,
      e_SessionError
);
```

The `suspendSession()` suspends a service session for the consumer. It can be used to suspend sessions which the consumer is currently active in, and sessions which the consumer has already suspended his participation.

`sessionId` is the identifier of the session to suspend.

The exception e_SessionError is raised if sessionId is invalid; or the session refuses to suspend because of this user's session state; or the user does not have permission.

4.4.2.4.21. suspendMyParticipation()

```
void suspendMyParticipation (
      in TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in TINACommonTypes::t_SessionId sessionId
) raises (
      TINAAccessCommonTypes::e_AccessError,
      e_SessionError
);
```

The suspendMyParticipation() suspends the consumer's participation in a service session. It can be used on a session which the consumer is currently active in.

sessionId is the identifier of the session to suspend this user's participation.

The exception e_SessionError is raised if sessionId is invalid; or the session refuses to suspend this user's participation because of their session state; or the user does not have permission.

4.4.2.4.22. resumeSession()

```
void resumeSession (
      in TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in TINACommonTypes::t_SessionId sessionId,
      in t_ApplicationInfo app,
      out TINAAccessCommonTypes::t_SessionInfo sessionInfo
) raises (
      TINAAccessCommonTypes::e_AccessError,
      e_SessionError,
      e_ApplicationInfoError
);
```

The resumeSession() resumes a service session. It is used on a session which is suspended.

sessionId is the identifier of the session to resume.

app is a structure containing information on the application, which will be used to interact with the service session. This application may be different to the user's original application that they were using when the session was suspended .

sessionInfo is a structure, which contains information which allows the consumer domain to refer to this session using other operations on this interface. It also contains infomation for the usage part of the session, including the interface references to interact with the session. (See Section 4.3.4).

The exception e_SessionError is raised if sessionId is invalid; or the session refuses to resume because of the user's session state; or the user does not have permission.

The exception e_ApplicationInfoError is raised if there are unknown or invalid values for t_ApplicationInfo, or if the application is incompatable with the type of service being resumed.

4.4.2.4.23. resumeMyParticipation()

```
void resumeMyParticipation (
      in TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
```

```
        in TINACommonTypes::t_SessionId sessionId,
        in t_ApplicationInfo app,
        out TINAAccessCommonTypes::t_SessionInfo sessionInfo
    ) raises (
        TINAAccessCommonTypes::e_AccessError,
        e_SessionError,
        e_ApplicationInfoError
    );
```

The `resumeMyParticipation()` resumes the consumer's participation in a service session. It can be used on a session which the consumer has previously suspended his participation from.

`sessionId` is the identifier of the session to resume the user's participation.

`app` is a structure containing information on the application, which will be used to interact with the service session. This application may be different to the user's original application that they were using when they suspended their participation.

`sessionInfo` is a structure, which contains information which allows the consumer domain to refer to this session using other operations on this interface. It also contains infomation for the usage part of the session, including the interface references to interact with the session. (See Section 4.3.4).

The exception `e_SessionError` is raised if `sessionId` is invalid; or the session refuses to resume the user's participation because of their session state; or they do not have permission.

The exception `e_ApplicationInfoError` is raised if there are unknown or invalid values for `t_ApplicationInfo`, or if the application is incompatable with the type of service being resumed.

4.4.2.4.24. joinSessionWithInvitation()

```
    void joinSessionWithInvitation (
        in TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
        in TINAAccessCommonTypes::t_InvitationId invitationId,
        in t_ApplicationInfo app,
        out TINAAccessCommonTypes::t_SessionInfo sessionInfo
    ) raises (
        TINAAccessCommonTypes::e_AccessError,
        e_SessionError,
        TINAAccessCommonTypes::e_InvitationError,
        e_ApplicationInfoError
    );
```

The `joinSessionWithInvitation()` allows the consumer to join an existing service session, to which the consumer has received an invitation.

`invitationId` is the identifier of the invitation. The invitation, kept by the retailer, contains sufficient information for retailer to contact the service session, and request that the consumer be allowed to join the session.

`app` is a structure containing information on the application, which will be used to interact with the service session.

`sessionInfo` is a structure, which contains information which allows the consumer domain to refer to this session using other operations on this interface. It also contains infomation for the usage part of the session, including the interface references to interact with the session. (See Section 4.3.4).

The exception e_SessionError is raised if the session refuses to allow the consumer to join it.

The exception e_InvitationError is raised if the invitationId is invalid.

The exception e_ApplicationInfoError is raised if there are unknown or invalid values for t_ApplicationInfo, or if the application is incompatable with the type of service being joined.

4.4.2.4.25. joinSessionWithAnnouncement()

```
void joinSessionWithAnnouncement (
      in TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in TINAAccessCommonTypes::t_AnnouncementId announcementId,
      in t_ApplicationInfo app,
      out TINAAccessCommonTypes::t_SessionInfo sessionInfo
) raises (
      TINAAccessCommonTypes::e_AccessError,
      e_SessionError,
      e_AnnouncementError,
      e_ApplicationInfoError
);
```

The joinSessionWithAnnouncement() allows the consumer to join an existing service session, to which the consumer has discovered an announcement. Session announcements may be gained in a number of ways (not described in Ret RP), including through a specialised service session.

announcementId is the identifier of the announcement. The announcement, kept by the retailer, contains sufficient information for retailer to contact the service session, and request that the consumer be allowed to join the session.

app is a structure containing information on the application, which will be used to interact with the service session.

sessionInfo is a structure, which contains information which allows the consumer domain to refer to this session using other operations on this interface. It also contains infomation for the usage part of the session, including the interface references to interact with the session. (See Section 4.3.4).

The exception e_SessionError is raised if the session refuses to allow the consumer to join it.

The exception e_AnnouncementError is raised if the announcementId is invalid.

The exception e_ApplicationInfoError is raised if there are unknown or invalid values for t_ApplicationInfo, or if the application is incompatable with the type of service being joined.

4.4.2.4.26. replyToInvitation()

```
void replyToInvitation (
      in TINAAccessCommonTypes::t_AccessSessionSecretId asSecretId,
      in TINAAccessCommonTypes::t_InvitationId invitationId,
      in TINACommonTypes::t_InvitationReply reply
) raises (
      TINAAccessCommonTypes::e_AccessError,
      TINAAccessCommonTypes::e_InvitationError,
      TINACommonTypes::e_InvitationReplyError
);
```

The `replyToInvitation()` allows the consumer to reply to an invitation, which the consumer has received. It allows the consumer to inform the retailer of his intention to join, or not, the session, or of a different location to look for the consumer. (Joining the session cannot be accomplished through this operation.)

`invitationId` is the identifier of the invitation.

`reply` is a structure which contains the information about the consumer's reply. (For details, see Section 3.3.5, "Invitations and Announcements").

The exception `e_InvitationError` is raised if the `invitationId` is invalid.

The exception `e_InvitationReplyError` is raised if there is an error in `reply`.

4.4.2.5 i_RetailerAnonAccess Interface

```
interface i_RetailerAnonAccess
        : i_RetailerAccess
    {
    // No operations defined at present
    };
```

`i_RetailerAnonAccess` interface allows an anonymous consumer access to services. The consumer uses it for all operations within an access session with the retailer.

This interface is returned when the consumer has established an anonymous access session with the retailer. It is returned by calling `requestAnonAccess()` on the `i_RetailerInitial` interface.

This interface inherits from `i_RetailerAccess` interface. The `i_RetailerAccess` interface is currently blank. It will be contain operations which are shared between the `i_RetailerNamedAccess` interface, and this interface. This means that the operations offered by this interface will change in the future.

4.4.2.6 i_DiscoverServicesIterator Interface

```
interface i_DiscoverServicesIterator
    {
    // Operations defined in the following subsections
    };
```

This interface returned by the `discoverServices`() operation on the `i_RetailerNamedAccess` interface. It is used to access remaining services, which were not returned by the `discoverServices`() operation.

The `discoverServices()` operation returns a list of services which matched some properties defined by the consumer. This interface allows the consumer to access the remaining services which were not returned by the call to `discoverServices()`. This is necessary because the list of services matching the properties could be very large, and include large amounts of information, potentially too much for the consumer's application to handle.

Using the `discoverServices()` operation, following by possibly multiple calls on the `nextN()` operation on this interface, allows the consumer to access all of the services matching the properties, without having to receive all of them at once

4.4.2.6.1. maxLeft()

```
void maxLeft (
      out unsigned long n
) raises (
      e_UnknownDiscoverServicesMaxLeft
);
```

The `maxLeft()` returns the maximum number of services which will be returned through this interface. These services can be accessed through multiple calls on the **`nextN`**() operation.

`maxLeft()` raises the if it is not possible for the retailer to determine the number of maximum number of services which could be returned.

4.4.2.6.2. nextN()

```
void nextN (
      in unsigned long n,
      out TINAAccessCommonTypes::t_ServiceList services,
      out boolean moreLeft
) raises (
      TINACommonTypes::e_ListError
);
```

The `nextN()` allows the consumer to access the remaing services which were not returned by the `discoverServices()` operation, or by previous calls to this operation. These services can be accessed through multiple calls on the `nextN()` operation.

The `n` parameter determines the maximum number of services to be returned. The length of the `services` list will not exceed `n`.

The remaining services are returned as the `t_ServiceList services`. This is a `sequence` of `t_ServiceInfo` structures, which contain the `t_ServiceId`, `t_UserServiceName` (consumer's name for the service), and a `sequence` of service properties, `t_ServiceProperties`.

The `moreLeft` parameter is a `boolean` to inform the consumer if there are any remaining services, after this call to `nextN()`.

4.4.2.6.3. destroy()

```
void destroy ();
```

The `destroy()` operation is used to inform the retailer that the consumer has finished with the `i_DiscoverServicesIterator` interface. It may be called at any time by the consumer, (i.e. the consumer does not has to have retrieved all the services before destroying the interface). After it has returned, the consumer will not be able to use their reference to the `i_DiscoverServicesIterator` interface again.

5. Usage Part

This section contains a definition and explanation of the usage part of Ret-RP. The usage part describes the interactions between the consumer and retailer domains during the use of a service session.

In fact, the usage part of Ret-RP defines interactions between usage party and usage provider session roles, (refer to Section 3). This means the usage part of Ret-RP may be used for usage across other reference points. Parties are users of the service, (human or otherwise), and providers are the providers of the service. For Ret-RP, a consumer corresponds to a party session role; a retailer to a provider session role.

Also a service session can involve multiple parties. The usage part defines interactions between a single party and a provider. However, it recognizes that there may be many instances of this reference point for a single service session, and so defines interactions that cope with actions from other parties.

Services offered by a provider will come in many different types, each offering functions which a user wants to use. The usage part of Ret-RP is not about these service-specific functions, and does not want to restrict the sort of functionality offered by services. It is about sets of generic operations which many services, and classes of services, will want to offer, in a consistent and interoperable manner.

The usage part of Ret-RP defines a set of profiles in terms of Session Models. A session model defines feature sets to encompass these generic operations. Each feature set provides some facet of session control, and defines interfaces to make this accessible across an interoperable reference point. They can be combined to provide the specific functionality required by the service.

Similar to the access part, usage part interfaces are first defined informally using plain text and diagrams, then by means of semi-formal OMG-IDL specifications; behavior is described in plain text.

NOTE: The main body of this document describes only interfaces and operations on interfaces. A complete listing of the IDL specifications, and how the interfaces are grouped into modules can be found in Annex A - Annex F:

The remainder of the usage part of Ret-RP is structured as follows

Section 5.1 contains a description of session models. Session Models define a set of interactions between the usage party and usage provider session roles. A number of session models might be defined by TINA and by other organisations.

Section 5.2 describes the TINA Service Session Model. It defines feature sets which are related to service session control.

Section 5.3 describes the TINA Communication Session Model. It defines feature sets which are related to communication session control.

IDL definitions of each of the interfaces can be found in Annex E:.

5.1 Session Models

Session models define how service session components in each domain can interact in a generic manner. These session models allow components which have been designed and implemented separately to interact to support the service session. A session model also defines an information

model for the session, and relates how operations on interfaces affect the information model, and so the behavior of the session. Sessions may support one or more session models to describe their behavior.

Currently defines a 2 session models: TINA Service Session Model, and TINA Communication Session Model.

The TINA Service Session Model is related to service session control. It defines interfaces which allow service session components to make requests to end and suspend the session; list the parties involved; set of stream bindings between parties, for example. The interfaces are grouped into 'self-contained' feature sets, allowing a session to support only those generic operations that are applicable to it.

The TINA Communication Session Model is related to communication session control. It defines interfaces which support requests required to set up Stream Flow Connections that support stream bindings initiated by service sessions (or their members). These interfaces are grouped into a single feature set.

The TINA session models enable interoperability between the domains, for a set of generic session control operations. These operations are offered on a set of interfaces. The interfaces are grouped into 'self-contained' feature sets, allowing a session to support only those generic operations that are applicable to it.

Service session related components may support one or more of a variety of session models. These session models may be defined by a variety of organizations. Each session may support a number of session models, or may only support a single model. A session may support either the TINA Service Session Model, or the TINA Communication Session Model, or may support both session models. It may also support other session models (not defined by TINA, or Ret-RP) in addition to, or in place of the TINA defined session models. Services may also decide not to support a session model. All of these alternatives are acceptable, following the statement in [7], that "the access part includes the possibility to negotiate alternative usage interfaces".

The Ret-RP allows services to support one of more of a variety of session models. These session models may be defined by a variety of organisations, or be specific to a particular provider. (e.g. T.120 defines a specific session model.)

Each session may support a number of session models, or may only support a single model. The models supported may be defined by the service type, (e.g. the service type identifies the type of service as well as the session models it supports), or unrelated to the service type. The usage part of Ret-RP allows sessions to support any models they wish. However, currently the only session models defined for the Ret-RP, are the TINA Service Session Model, and the TINA Communication Session Model. We will assume that the session models act independently, and will only describe TINA session model interactions.

In addition to the session models, a session may also support other service specific interfaces.

In the access part of Ret-RP, when the consumer domain requests to start a service, it can ask that the service session supports a particular session model. Each session model is identified by a specific string, e.g. for the TINA Service Session Model, it is "TINAServiceSessionModel". This string is passed as part of the request to start the service. If successful, the new session will support the specified session model. If unsuccessful, the request raises an exception, indicating that the session model cannot be supported by the service.

In the usage part of Ret, it is possible to ask a session which session models it supports. The same identifying strings are used as in the access part. If the session supports the TINA Communication Session Model, then it will return “TINACommSessionModel” as a value in the list of session models. (The list may contain a one or more session model names, or none at all.)

The following sections describe the TINA session models. Other session models, and how they may be requested through the access part of Ret-RP, are described in the Access part of Ret-RP.

5.2 TINA Service Session Model

The TINA Service Session Model provides generic service session control operations, applicable to single and multi-party services. It defines interfaces which allow service session components to make requests to end and suspend the session; list the parties involved; set of stream bindings between parties, for example. The interfaces are grouped into ‘self-contained’ feature sets, allowing a session to support only those generic operations that are applicable to it.

The TINA service session model defines a number of feature sets and interfaces which can be used to interact with a session. The session may also support other service specific interfaces, (and probably will), and may support other session models. We will assume that the session models act independently, and will only describe TINA service session model interactions.

It also defines the Session Graph [2] information model. This information model can be used to define the behavior of operations on the interfaces. It is not necessary for the service session components to implement the session graph information model. The session graph information model is used purely to describe the behaviour of the session model operations, with Ret-RP. Ret-RP does not prescribe how to implement the interfaces it defines.

The TINA service session model is identified by the string “TINAServiceSessionModel”[1], in the access and usage parts of Ret-RP.

The TINA service session model defines a basic feature set, (BasicFS). It must be supported by all sessions which support the service session model. It provides sufficient functionality to control a single party session.

The TINA service session model also defines additional feature sets which may also be supported by the session. The feature sets which a session supports is available through the access part of Ret-RP, and through operations on the interface defined by the basic feature set.

5.2.1 TINA Service Session Model Feature Sets

The table below provides a brief description of each of the feature sets. All feature sets are ‘optional’, (except BasicFS). This means a session can support the TINA service session model, and only support the feature sets that are useful to the session. For example, a single party service (e.g. Video-on-demand) may not wish to support the interfaces associated with the multiparty feature set, but may wish to support the functionality of multimedia streams from the participant-oriented stream binding feature set.

1. Previous versions of the Ret-RP used: TINA Session Model, for the TINA Service Session Model, and the identifying strings: “TINASessionModel” and “TINA Session Model”. This name and identifying strings are no longer used.

The feature sets define interfaces that must be supported by the appropriate domain. The session role of a domain determines which interfaces of a feature set supported by the domain. A domain in a usage provider role offers usage provider type interfaces, and requires usage party interfaces from the other domain in the service session.

Current feature sets are defined for the usage party and provider roles only. Feature sets for peer roles have not yet been defined. Other more specialised roles may have specific feature sets to support those roles. E.g. a domain in a manager role may require a management feature set (not defined).

The behavior of the operations on each interface is also defined. However, the precise behavior of an operation can be affected by the service, and other feature sets supported. For example, a request to end the session, if it returns successfully, will always mean that the session has ended. However, if the multiparty feature set is supported, it also means that the other parties are told to execute any actions required to end the session. If MultipartyIndFS is supported, then the other parties will be informed that a party has requested that the session end, and may be allowed to vote (VotingFS), refuse (through service specific interactions). Or the request may be automatically refused because the party doesn't have permission to end the session (ControlSRFS).

Feature sets generally rely on other feature sets. E.g. the multiparty feature set provide operations to support inviting new users, and ending a user's participant. But it doesn't provide operations to end the session, as these are supported by the basic feature set. This means a session that supports MultipartyFS must also support BasicFS. (MultipartyFS is dependant upon BasicFS).

Table 5-1. TINA Service Session Model Feature Sets

Feature Set	Description	Dependant on
BasicFS	Support end and suspend session requests. Allows the party domain to discover interfaces and session models supported by the session.	Mandatory for TINASessionModel
BasicExtFS	Allows the provider domain to discover interfaces and session models supported by the party domain components.	BasicFS
MultipartyFS	Allows the session to support multiparty services. Supports requests for: - information on other parties - ending/suspending a party in the session - inviting a user to join session - announcing the session	BasicFS
MultipartyIndFS	Allows the session to indicate requests that are to be processed, to the party components.	MultipartyFS
VotingFS	Supports parties voting to determine if a request should be accepted, and executed.	MultipartyIndFS
ControlSRFS	Supports parties having ownership, and read/write rights on session entities, (i.e. parties, resources, streams, etc.)	MultipartyFS
ParticipantSBFS	Participant type stream binding feature set: provides high level support for setting up stream bindings. Stream bindings are described in terms of session members participation.	BasicFS

Table 5-1. TINA Service Session Model Feature Sets

Feature Set	Description	Dependant on
ParticipantSBIndFS	Participant type stream bindings with indications	ParticipantSBFS

5.2.1.1 BasicFS

The basic feature set (“BasicFS”) must be supported by all sessions which support the TINA service session model.

BasicFS provides sufficient functionality to control a single party session. The basic feature set enables a party domain to:

- discover the interfaces, session models and feature sets supported by a session;
- retrieve the interfaces supported by the session, (including service-specific, and ‘feature set interfaces’);
- register the client’s own interfaces and session models with the session;
- end and suspend the session.

Table 5-2. BasicFS Interfaces

BasicFS interfaces on:	
Party domain components	(none)
Provider domain components	i_ProviderBasicReq

The basic feature set allows domains to exchange and agree the use of additional non-standardized interfaces. This is essential to obtain service-specific interfaces and value added interfaces above or instead of TINA standard session control.

BasicFS supports a client-server paradigm, where the party domain is the client, and the provider domain is the server. All requests using BasicFS interfaces are initiated from the client applications, and are serviced by the provider domain components. This means that the simplest service that can be implemented using BasicFS is a single party service, that has interfaces on the provider domain components, with no interfaces on the party domain applications. This does not mean that services using BasicFS are restricted to single-party services, or only client-server interactions. Service sessions may support additional feature sets, and service-specific interfaces that support multiple parties in the session, and peer-to-peer interactions.

5.2.1.2 BasicExtFS

BasicExtFS allows provider domain components to discover interfaces and session models supported by the party domain.

BasicExtFS is an optional feature set, which requires that the session also support BasicFS.

Table 5-3. BasicExtFS Interfaces

BasicExtFS interfaces on:	
Party domain components	i_PartyBasicExtReq
Provider domain components	(none)

BasicExtFS provides the inverse of the BasicFS. It supports the opposite of the client-server paradigm, in that the provider domain can gain information about the session models and interfaces supported by the party domain.

It does not support any session control operations, such as ending or suspending the session, from the provider's domain.

5.2.1.3 MultipartyFS

MultipartyFS allows the session to support multiparty services.

MultipartyFS is an optional feature set, which requires that the session also support BasicFS.

It supports the party domain making requests for generic multiparty control actions, such as suspending a party's participation in the session. It also supports the session providing information on events that have happened to other participants, e.g. another party has suspended; and the session asking the party domain to execute an action, (e.g. the party is being suspended, and the party domain components need to perform some actions before the party is suspended.)

Table 5-4 MultipartyFS Interfaces

MultipartyFS interfaces on:	
Party domain components	i_PartyMultipartyExe i_PartyMultipartyInfo (optional)
Provider domain components	i_ProviderMultipartyReq

MultipartyFS provides operation to request and execute generic multiparty control actions, such as suspending a party's participation in the session. It does not define whether a particular party is allowed to perform the action. The operations are defined with exceptions that can be raised, if the session determines that a request for an action it not allowed. It is up to the session, and possibly parties to decide if an action should be allowed, or should not be performed.

The session may use entirely service specific mechanisms to decide if an action should be performed. Alternatively, ControlSRFS may be used to associate owners to session entities, which determine if an action is allowed. Also MultipartyIndFS and VotingFS allow the session to indicate to party domain that an action has been requested, and to vote on whether an action is performed. All three may also be used together to determine which parties are indicated about the action, and which can vote.

i_PartyMultipartyInfo interface allows the session to inform the party domain of changes in the state of the session and its participants.

5.2.1.4 MultipartyIndFS

MultipartyIndFS allows the session to indicate that an action will be taken shortly. (e.g. a party is going to be suspended.) The party may be able to vote on whether they wish this action to be taken, if the session supports the Voting feature set.

Table 5-5. MultipartyIndFS Interfaces

MultipartyIndFS interfaces on:	
Party domain components	i_PartyMultipartyInd
Provider domain components	(none)

This feature set is dependant on the session supporting MultipartyFS.

5.2.1.5 VotingFS

VotingFS allows the parties to vote on whether an action should occur. (The party domain finds out that action is going to occur by receiving an indication through the MultipartyIndFS.)

Table 5-6. VotingFS Interfaces

VotingFS interfaces on:	
Party domain components	i_PartyVotingInfo
Provider domain components	i_ProviderVotingReq

This feature set is dependant on the session supporting MultipartyIndFS.

5.2.1.6 ControlSRFS

The Control Session Relationships feature set (ControlSRFS) is used when Session Relationships need to be modified. Session relationships define the ownership and permission of parties, resources, etc. in a service session. This feature set allows parties to change the ownership and permissions of other parties to perform operations that affect the owned entities.

Table 5-7. ControlSRFS Interfaces

ControlSRFS interfaces on:	
Party domain components	i_PartyControlSRInd i_PartyControlSRInfo
Provider domain components	i_ProviderControlSRReq

This feature set is dependant on the session supporting MultipartyFS.

5.2.1.7 ParticipantSBFS

This feature set provides capabilities to setup, modify, and delete stream bindings between members of the session. It supports a high level description of the feature set, based on members participating in the stream binding. Session members use these descriptions to determine their participation and return suitable SI information to allow the stream binding to proceed.

Table 5-8. ParticipantSBFS Interfaces

Participant SBFS interfaces on:	
Party domain components	i_PartyPaSBExe i_PartyPaSBInfo (i_ConnInfo)
Provider domain components	i_ProviderPaSBReq

5.2.1.8 ParticipantSBIndFS

ParticipantSBIndFS allows the session to indicate that an action will be taken shortly. (e.g. a stream binding will be deleted.) The party may be able to vote on whether they wish this action to be taken, if the session supports the Voting feature set.

Table 5-9. ParticipantSBIndFS Interfaces

ParticipantSBIndFS interfaces on:	
Party domain components	i_PartyPaSBInd
Provider domain components	(none)

5.2.2 Types of Operations and Interfaces.

The feature sets support 4 main types of operations:

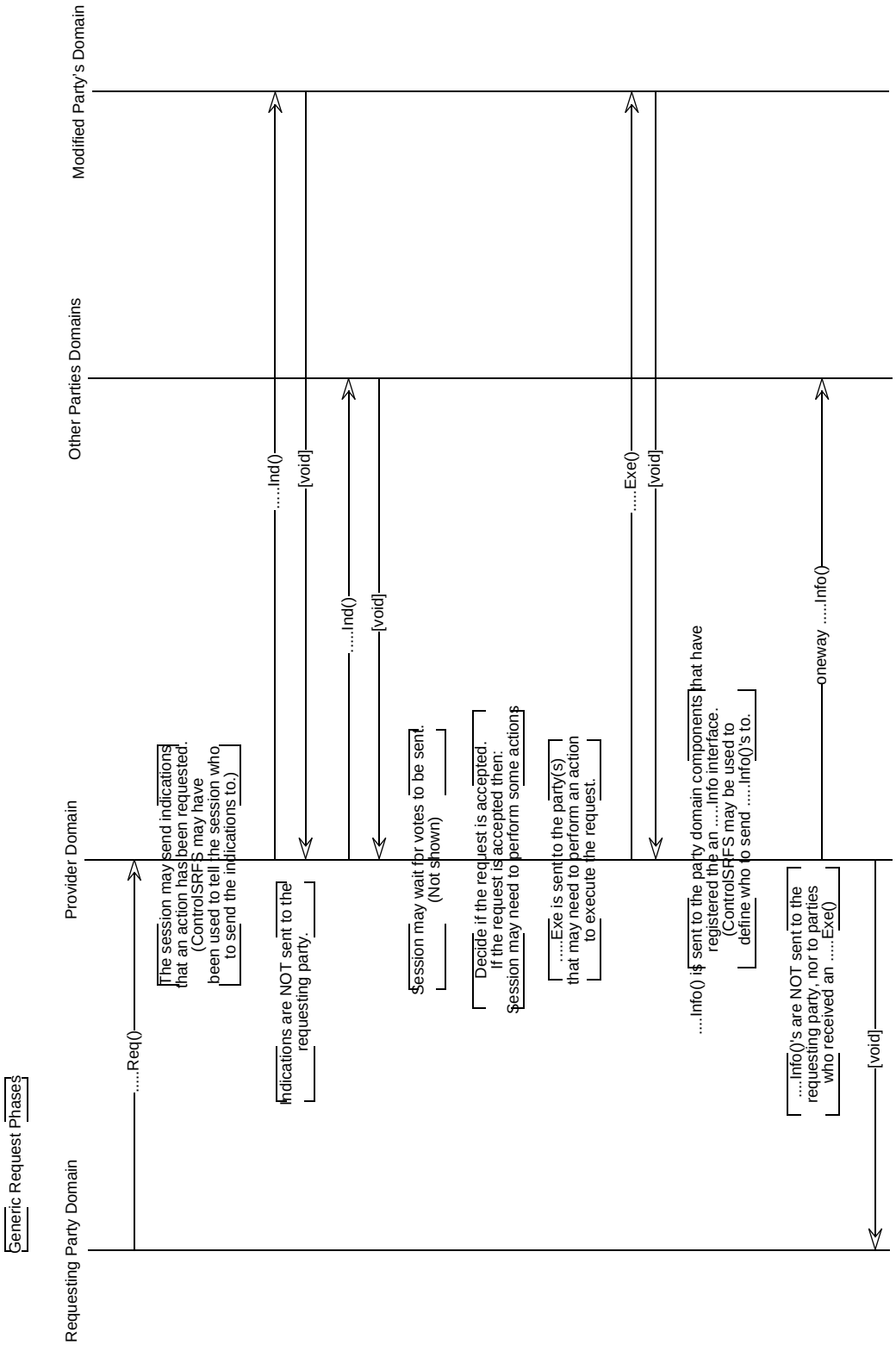
- Request operations ....Req()
- Indication operations ....Ind()
- Execution operations ....Exe()
- Information operations ....Info()

Not all operations support this convention, but the following describe how to interpret operations which do. (Operations which don't support the convention are all described individually in the feature set description.)

In general, each type of operation is supported on a separate interface. Interfaces supporting request operations are suffixed by Req; interfaces supporting information operations are suffixed by Info, etc.

5.2.2.1 Request operations.

Request operations (Req()) are generally supported by interfaces on the provider domain. They allow the party to request for a particular generic control operation to be performed, (e.g. end the session). Each feature set supports requests for generic control operations which are applicable to the purpose



of the feature set. (E.g. BasicFS supports generic operations to end and suspend the session;

MultipartyFS supports operations to end and suspend my participation in the session, and invite others to join the session).

All Req() operations include a `t_ParticipantSecretId` as the first parameter. This allows the session to distinguish which participant has sent the request. (This is useful even if requests from each participant arrive on different instances of the interface.) (See Section 3.3.4.2, "t_ParticipantSecretId" for more details.)

Req interfaces support Req()s to perform generic session control actions, such as suspending a party's participation in the session. The interfaces do not define whether a particular party is allowed to perform the action. The operations are defined with exceptions that can be raised, if the session determines that a Req() for an action is not allowed. It is up to the session, and possibly parties to decide if an action should be allowed, or should not be performed.

A Req() operation will eventually return to the requesting party. If it returns successfully. (i.e. no exception is raised,) then the request has been agreed by the session, and actions have been taken to perform the request. It is no longer possible for the request to be 'rolled back'. The request has been executed by the session, and the requesting party MUST now perform any action necessary for it to complete the request, (i.e. remove a party whose participation has been ended from the party's model of the session). Equally the session cannot return successfully to the requesting party until there is no possibility for the request to be 'rolled back', i.e. the session must both agree to the action, and perform the actions required to ensure the request is performed, before success is returned to the client. This effectively means that a Req() cannot return until all corresponding execution operations have returned successful to the session. (However, info operations do not have to return

Requests may not be performed because the requesting party does not have permission to perform the request, (e.g. a request to end another party's participation in a session may be denied because only the other party have the right to end their participation; or all the parties in the session have voted on this request and it has been denied.)

The session may use entirely service specific mechanisms to decide if an action should be performed. Alternatively, ControlSRFS may be used to associate owners to session entities, which determine if an action is allowed. Also MultipartyIndFS and VotingFS allow the session to indicate to the Party domain that an action has been requested, and to vote on whether an action is performed. All three may also be used together to determine which parties are indicated about the action, and which can vote.

All Req() operations have a common set of exception codes which can be raised to indicate that the request has failed. They include failure due to invalid `t_ParticipantSecretId`; failure due to not allowed; and failure due the operation not being supported, (e.g. BasicFS mandates a suspendSessionReq() operation to be available, but a particular service that has not implemented suspend, should always raise the 'not supported' error code in response to this Req().)

If this session additionally supports the MultipartyIndFS, then Indication operations (Ind()) will be invoked on the other parties in the session. (Precisely which parties receive indications is decided by the session, and may be defined by use of the ControlSRFS.)

If this session supports MultipartyFS, then an Execution operation will be sent to the party to which the action is being applied. (No Execution operation will be sent if the operation pertains to the requesting party, see Execution operation for more details). After the Execution operation has

successfully completed, a Information operation will be sent to the other party's, (except the requesting party, and the party receiving the Execution operation, see Information operation for more details.)

5.2.2.2 Indication operations

Indication operations (Ind()) are supported by interfaces on the party domain. They allow a session to indicate to the party that a generic control action is about to be taken.

The Ind()s are sent out to party domain components before the generic control action actually occurs. This allows the component to potentially perform some (internal) action before the action is executed. The party may also be able to send a vote to the session indicating whether they wish the action to be completed, (if both support the voting feature set).

Indications are never sent to the requesting party. It is always assumed they are aware that they have sent the Req() to the session. (In the voting feature set the requesting party is not able to vote, but is automatically assumed to have sent an 'Agreed' vote response.)

After receiving an indication, the party domain component will either receive an Information (Info()), Execution (Exe()), or a operationCancelled() operation. Info() and Exe() operations are described in the following sections.

5.2.2.3 Execution operations

Execution operations (Exe()) are supported by interfaces on the party domain. They are invoked on parties which have to perform explicit actions, due to a request for a generic control operation.

These operations are invoked when the session has determined that the request is allowed. The party must execute the operation, and should only raise an exception if it is unable to execute the operation, (e.g. if the party is suspended, due to a request from another party, they must suspend.)

If the party does raise an exception, then the whole request and action must be 'rolled back' by sendig an operationCancelled() operation to all the parties that have received and performed Exe() operations. This allows the party's to update their internal representation of the session to the state before the Exe() was sent.

Since Exe()'s can raise exceptions, it effectively means that the request cannot be considered to have been successful, until all the Exe() operations have returned successfully to the session.

The requesting party does not receive executions. They are assumed to have performed the required actions before issuing the request (for 'creations'), or after the request has been successfully completed, (for 'deletions').

5.2.2.4 Information operations

Information operations (Info()) are supported by interfaces on the party domain. They are invoked after a generic session control operation has successfully completed. (i.e. after all Indications, and Executions have returned successfully, and the action has been successfully completed in the session, but just before the Req() operation returns to the requesting party.)

They are invoked on parties which have previously registered their interfaces to the session. (The ControlSRFS may determine which parties receive Information operations; otherwise all parties should receive Information operations when a generic control operation request is successful.)

The Information operation are always ‘oneway’ operations. It is not possible for a party to raise an exception due to receiving an Information operation about an action that has already been successfully completed.

5.3 TINA Communication Session Model

The TINA Communication Session Model describes the interfaces required to support communication session level interactions across the Ret-RP. The communication session does not (currently) support requests for connections from party domains: these types of requests are handled by the stream binding feature sets of the TINA service session model. Instead, the communication session supports lower level requests required to set up Stream Flow Connections that support stream bindings initiated by service sessions (or their members).

The TINA communication session model is identified by the string “TINACommSessionModel”, in the access and usage parts of Ret-RP.

The communication session interface specifications here are not mandatory. The interface described here only supports basic functionality. Other interfaces may be used, but should at least support the basic functionality specified here. Extra abilities and communications from the party to the provider may be desired. A list of desirable extra functionality can be found in Section 6.2.2.1, "TINA Communication Session Model additional functionality".

Currently the communication session model does not define its interfaces in terms of feature sets. This is because the functionality defined is equal to a basic feature set for the communication session. When interfaces to support the extra functionality are defined, the session model will be structured according to feature sets. The interfaces defined here will become the basic feature set for the TINA communication session model.

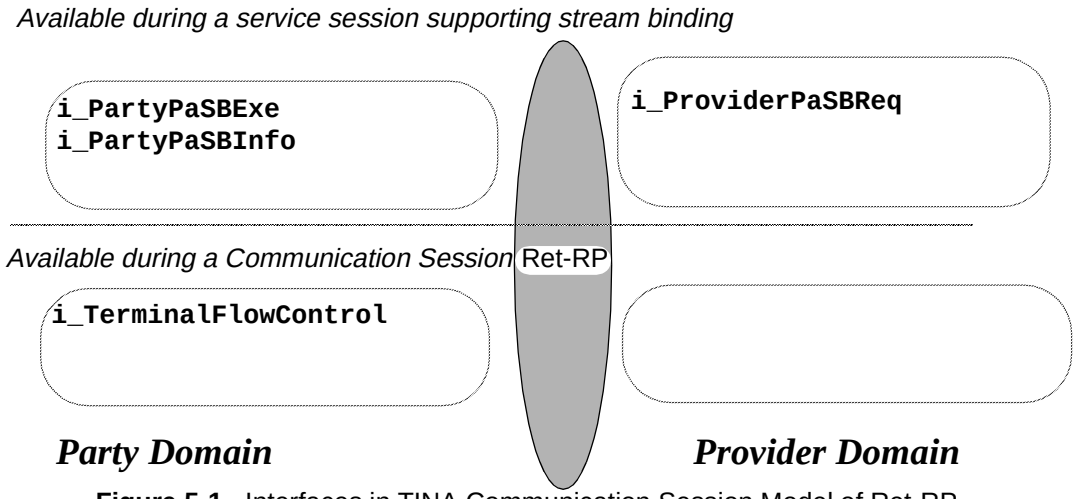


Figure 5-1. Interfaces in TINA Communication Session Model of Ret-RP.

Definitions of these interfaces and the operations they support can be found in Section 5.6.2, "Communication Session Model Interfaces".

5.4 Usage Information View

The information model of the usage part of the Ret-RP is described in 4 parts:

- **TINA Service Session Model related information**. This is mainly defined by a Session Graph, a network of related information objects, that describe the session and its participants. The session graph provides an overview of the session, and chnges to it descibe changes in the session. Section 5.4.1 describes the session graph concept, as well as common types related to many of the feature set of the TINA service session model.
- **Stream Binding Terminology**.
- **Common Communication Session and Stream Binding Data Types**.
- **Communication Session Model Information View**.

5.4.1 TINA Service Session Model related Information

The basic concept used to describe the information contained in and exchanged within the service session control is the Service Session Graph (SSG). It is important to make the difference between a particular instance of a service session graph and the concept itself: the concept provides the tool with potential operations and elements which are generic (service independent) while a particular instance is a specific instantiation or activation of one or more of those elements. The SSG information model is valid for both the local (user) views and the central (provider) view.

The service session control's SSG information model is described below using OMT notation, by means of the diagram in Figure 5-2.

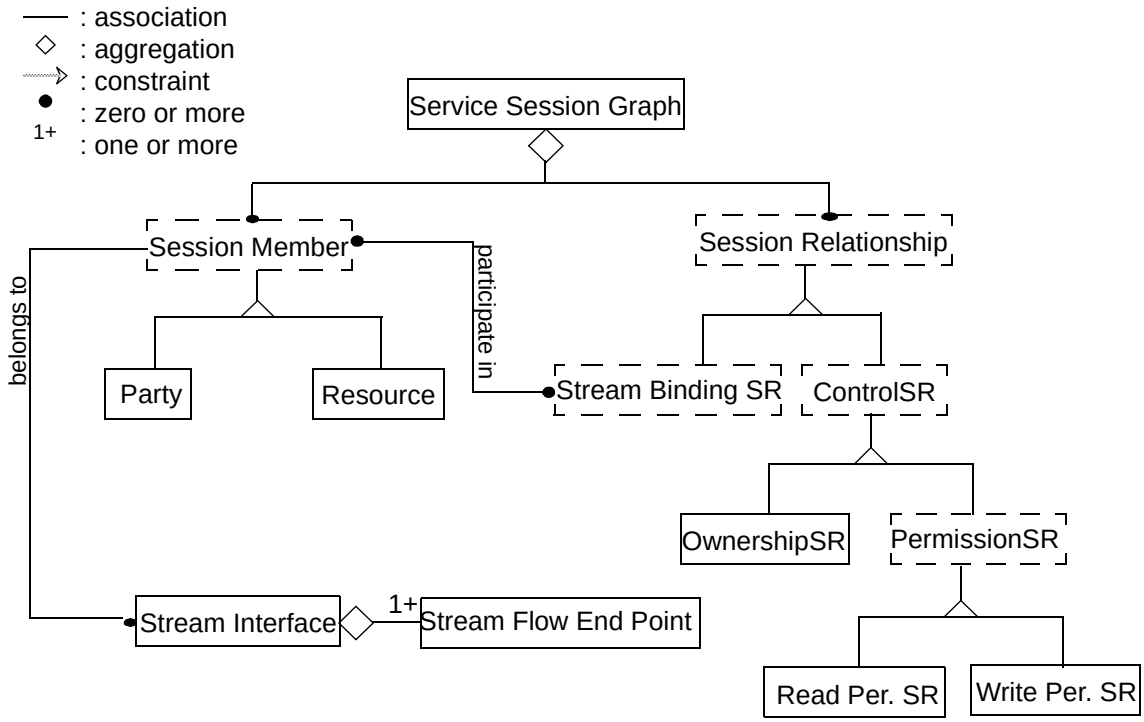


Figure 5-2. Service Session Graph aggregation.

First of all, the SSG is composed of (aggregates) several objects as shown in Figure 5-2. The figure just gives a *first idea* of the model and *is neither complete nor self explanatory*: the classes, relations and cardinalities and their usage are explained in more detail in the document “Service Component Specification” [19].

The abstract classes (for which there is no object instance) are shown in dash style: these are the session member and the session relationship classes: they have no other purpose but to simplify the object oriented modeling.

The Service Session Graph (SSG) model defines all the generic classes that are necessary to fulfil its mission: offering a generic platform to services. The concept is used to model and control the state of a TINA service session. An instance at a certain point of time of an SSG models a “snapshot” of the resources, the parties and the relationships established into the service session. Note that in Figure 5-2 the cardinality of most relationships is defined as “zero or more” for reasons of flexibility. Most of the classes described in the model are optional (zero instances when not used), or can be used many times (for multi-party and multi-media requirements, as many instances of the classes will be required as parties and medias involved). This flexibility allows to reuse the model for every features set that has been defined. Details of the information model are given in the following sections.

The Session Graph concept, which provides an overall framework, that is described in Section 5.4.1 through Section 5.4.1.4. The session graph has to be considered as the overall framework on which the different Feature Sets of the TINA Session model are based. A description of the Feature sets can be found in Section 5.2.1. The relationship between TINA session model Features Sets and the session graph information objects are explained in Section 5.4.1.5.

5.4.1.1 Service Session Graph Object Classes

The service session information object is manipulated by the components in the consumer and retailer domains, which are defined in [19]: they provide the operations for modifying the service session. Every feature set allows the modification and manipulation of subsets of the Information Classes here described. In the session graph model, this corresponds to operations defined for each IO class in the model.

The detailed operations specification is done in TINA in the computational viewpoint and can be found in the individual descriptions of TINA Session Model Feature Sets. Nevertheless, it is useful to define high-level operations on IO classes.

These high-level operations[1] are:

- **create**: this type of operation will initiate the creation of a new object in the service session; it includes its configuration, i.e. the need for specifying the objects from which this new object depends, and to set its attributes. Default value mechanisms will be foreseen. The respect of the relations cardinality is checked at creation time;
- **modify**: this type of operation will be used to modify object's attribute values at any time in the service session's lifetime;
- **delete**: this type of operation will be used to remove one or more objects and their dependant objects from the SSG;
- **suspend**: to put an active object into the suspended state;
- **resume**: to put a suspended object into the active state.

1. Sometimes these operations are not offered to external clients

5.4.1.2 Service Session Graph Information Model

This section gives an high level description of all SG Information objects classes apart from those related to control session relationships that will be described in the following section.

The **Service Session Graph (SSG)** class is the top class of the information model, and represents the service session as a whole. It contains (aggregates) all other objects in the model. It can contain information for scheduling the entire service session.

Party. It is defined as a negotiating entity taking part in the service session, not to be confused with the usage party session role: it can be either an end-user, a subscriber or a service or resource provider. It models a (potential) “user” in the service session. The party object is used to maintain information about a user in the service session and/or any of the user service sessions participating into the service session. In supplement of the generic attributes already described, an attribute that is maintained for the party is its name, which uniquely identifies the user.
A user can request to take part in the service session by requesting to become a party in the SSG. A party can invite another user to join the service session by requesting such a party to be created for that invited user. If there are already other parties involved in the same service session, they might have to confirm the invitation while the service session negotiation is taking place. A party can request to remove another party (or himself) from the service session by requesting the corresponding party object to be deleted. This might involve negotiation again, depending on the presence of control session relationships. In general, any negotiation of any operation handled by the service session will have a behavior that depends on the presence of control session relationships: this is further elaborated in Section 5.4.1.4.
The SSG contains zero or more party objects: there will be as many party objects as negotiating entities in the service session. What is meant by negotiation in this context is explained in detail in Section 5.4.1.4.1. The case with zero party will be transitory and is shown for completeness.
The party is the only class that is present in every feature set.

Resource. It models a source of support for the execution of the service (session). It can model many types of resources to be identified or shared in the service session (e.g. a file to be retrieved, a shared pointer in that file (global cursor), a conference bridge, a service or service subscription file, a VOD server, etc.). The resource is identified as a part of the service session upon request of parties or upon request of the service logic itself. The interaction held between a party and a resource is “not negotiated”. Resources do not take an active part in the negotiation, in the sense that resources cannot initiate negotiations but only react to requests: resources answer positively or negatively to requests whenever they are involved in a service session negotiation[2].
The SSG contains zero or more resource objects: there will be as many resource objects as resource used in the service session.
Resource is the main information object class of the Resource Feature Set.

Session member abstract class: The party objects and the resource objects that have been described above have several commonalities. First they both will require to be interconnected: connections between Parties for audio and video communication, connections between Parties and resources for information retrieval, connections between resources for information transfer, etc. This shows the need for defining means of communication for both the party class and the resource class. Another commonality is that the party and the resource will both require scheduling. In order to exploit

2. This does not imply any assumption on the behavior of the entities modelled as resources outside of the scope of the model. For example, entities that are modelled as resources in the SSG model (such as video bridges) can emit notifications, within or outside of a service session, according to service specific or context specific policies. This has nothing to do with the behavior of such entities during a negotiation process based on the SSG model, which is a “passive” role by definition. Negotiation is presented in Section 5.4.1.4.1.

fully the strength of Object Oriented Modelling (OOM) and enable the service designers and implementors to reduce redundancy, a class generalization of the party and resource classes will be defined: the Session Member (SM) class. This class is abstract because it cannot be instantiated: only the specialized classes can. Since party and resource classes are now defined as specializations of the session member class, they will be more completely named as respectively Party Session Member (PSM) and Resource Session Member (RSM).
Sometimes it is useful to group Party Session Member and/or Resource Session Member in order to be able to define relations in the service session that are valid for each member of such a group. The following group are defined: Session Member Group[3], Party Session Member Group[4] and Resource Session Member Group[5]. These classes are not depicted in the overall diagram of the TINA session model, more details can be found in the Service Component Specification [19].

5.4.1.3 Stream binding related parts of the SSG

In order for a session member to express its ability to communicate with other session members by means of streams[6], the session member is associated to stream interfaces belonging to it, as shown in the overall figure (Figure 5-2). The stream binding session relationship class represents the binding between parties or their corresponding stream interfaces and stream flow endpoints.

Stream Flow Endpoint (SFEP): It represents the termination of a single flow by an application. In other words, it can be seen as the logical representation of a physical stream device I/O port of a terminal (e.g. the outlet of a camera). Further details can be found in NRA [9].

Stream Interface (SI): It represents a dynamic grouping of SFEPs. The session member will associate as many stream interfaces as required (eventually none, when the stream binding feature set is not supported); the cardinality is shown in the Figure 5-3. It's worth noting that the session member - stream interface relationship is inherited by the SM's specialized classes: party, peer and resource. The stream interface class used here is defined in NRA (see [9]).

Stream flow connection (SFC): It describes point-to-point or point-to-multipoint connections between SFEPs. Figure 5-3 describes stream flow connections and the relationships between SM, SIs, SFEPs and SFCs.

Stream binding session relationship(SBSR)[7]: It can be described by a number of different information models. The stream binding information model is related to the setup of the binding (i.e., it is used to describe the stream binding we want to set up) and any explicit setup and modification control operations. Many models are possible, but we are most interested in those that relate the stream binding to SI and SFEPs, since these have computational aspects that may be supported by a DPE. TINA has defined three different ways (below described) to provide stream binding; all of them are based on the concept of stream flow connections.

---

3. **Session Member Group (SMG)**: it can contain any number of session member of any kind (indifferently PSMs or RSMs, even mixed together; see Service Component Specification.
4. **Party Session Member Group (PSMG)**: it is a specialization of the SMG because it will contain PSMs only; note that this class could carry Closed User Group (CUG) identification; see Service Component Specification.
5. **Resource Session Member Group (RSMG)**: it is a specialization of the SMG because it will contain RSMs only; see Service Component Specification.
6. As opposed to using operational interfaces.
7. Stream binding and stream binding session relationship (SBSR) will be used as synonyms. Normally 'session relationship' will be suffixed when the emphasis is on SBSR as a subclass of session relationship (SR).

Participant oriented stream binding: In this high level approach, stream bindings are described in terms of participating session members, type and QoS. Each participant is then requested for suitable stream interfaces which are to be bound by the stream binding. This high level description can be mapped by the service logic to a number of stream flow connections. These stream flows may be of different types and QoS. Special resources, such as bridges, may also be introduced to support the binding. The stream binding may be manipulated by modifying the type, QoS and participants.
This type of stream binding specification allows a high level request of multi-party-multi-media stream bindings[8]. The type and QoS parameters may be used to implicitly specify multimedia requirements[9].

Depending on the type of service, there may be different needs for explicit manipulation of individual SFEPs and SFCs. Hence, TINA supports all three models. There might also be cases where SBSR groups may be defined[10]. The different feature sets for the three different approaches are listed in Table 5-1. Detailed descriptions of these Feature Sets can be found in Section 5.2.1.7, "ParticipantSBFS" and Section 5.2.1.8, "ParticipantSBIndFS".

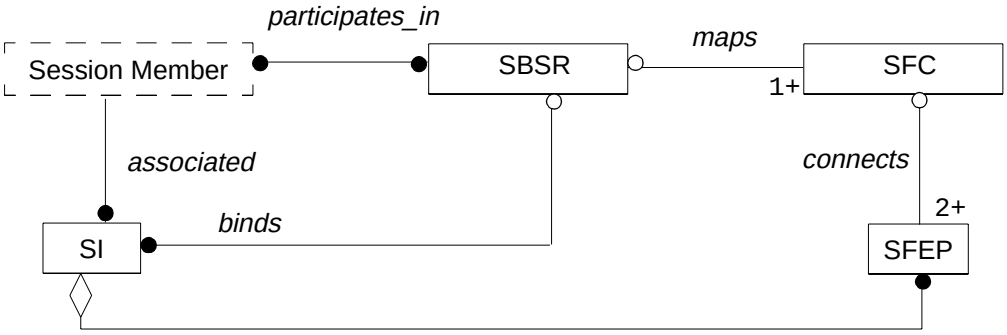


Figure 5-3. Stream binding model

In addition to the SSG related objects, the following terms related to the communication session are also useful in understanding stream binding concepts. These terms are briefly introduced here to aid later discussions of stream binding.

- **Network Flow Connection** (NFC): It describes point-to-point or point-to-multipoint connections between NFEPs.
- **Network Flow End Point** (NFEP): It describes a network termination in a technology independent manner.
- **Network Flow End Point Pool** (NFEPPool): It identifies either a group of existing NFEPs or a resource that can dynamically create NFEPs on request.
- **Resource Flow End Point** (RFEP): It represents either a NFEP or a NFEPPool.
- **Terminal Flow Connection** (TFC): It describes a point-to-point connection between an SFEP and an NFEP.

8. (i.e., one stream binding representing a multipoint-to-multipoint binding, that maps onto a number of stream flow connections, each representing point-to-multi-point connections between SFEPs).
9. However, it is also suitable for simple stream bindings with little overhead (i.e., this approach allows for a computationally 'lightweight' way of setting up stream bindings).
10. SBSRs may also be grouped together with CtrSRs into SRG.

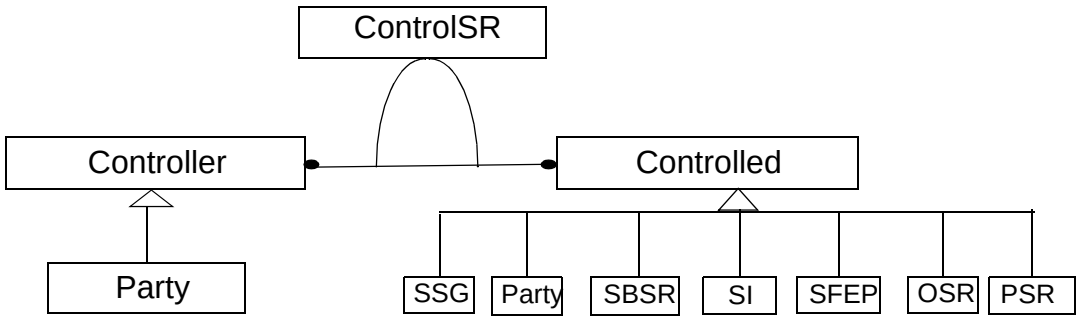
5.4.1.4 Expressing Control Relationships in the Service Session

The control session relationship (ControlSR) class allows to perform enhanced control on the service session objects. They support the mechanism of negotiation and voting, therefore they are the basis for the Control Session Relationships (ControlSRFS, Section 5.5.8) and Voting Feature Sets (Section 5.5.7). In other words if ControlSRFS is not supported, this information object is not present and no negotiation can take place.

Several specialized classes are defined for the ControlSR, for each specific control requirement, as shown in Figure 5-4. Their usage is explained below.

When a control session relationship has to be established between two objects in the SSG, there will always be one object having the “controller” role, and another having the “controlled” role.

In a similar way as was done for the PSM and the RSM classes, it might be useful to define a grouping



Note:
The ‘zero or more’ cardinality bullets in OMT do not relate to the describing class for the relation (in this case control SR class), so the correct reading of this diagram is:

- a control SR instance must always have one controller and one controlled class instance;
- a controller instance can be related to zero, one or more control SR instances;
- a controlled instance can be related to zero, one or more control SR instances.

Figure 5-4. The Control SR.

class for both the ControlSR, and the SBSR already introduced in the previous section. This new grouping class is called Session Relationship Group (SRG) and relates to the SSG, the ControlSR and the SBSR.

The following sections describe the ownership session relationship and the permission session relationships, which are both specializations of the ControlSR.

5.4.1.4.1. Ownership Session Relationship (OSR)

The Ownership SR (OSR) associates a partySM or partySMG (a controller class object) to a controlled class object in the SSG. It is used to force multi-party negotiation when an operation is requested on an object that is owned by one or more other Parties. The Ownership SR specifies which party SM or party SMG owns that object instance in the SSG. The ownership of that object

instance includes its eventual attributes, its attachment to associations / relations with other objects, the objects it eventually aggregates, and the OSR itself. The OSR allows multiple PSMs and/or PSMGs to be owners of an object instance at the same time (multi-party ownership).

When one or more controller objects have an OSR relation with a controlled object, they will be called “owners” of that object, and the controlled object will be “owned”. As a consequence of this OSR relation, the behavior of the negotiations in the service session will be modified: when a party (e.g. an end-user) requests a modification of the controlled (owned) object, the service session will:

- authorize the modification if the requesting party is one of the owners;
- start a negotiation with the owners if the requesting party is not an owner.

If a negotiation is required, each owner will be informed of the request, and will have to answer either positively or negatively. An attribute of the OSR will indicate the voting rule in effect for this OSR instance.

The possible “modifications” of the owned objects include the following:

- a modification of one or more of the attributes of the owned object;
- a modification of the associations of the owned object (adding or removing such an association);
- adding or removing any object aggregated by the owned object;
- adding to or removing the owned object from a “group” object.

The OSR allows services to customize negotiation on a service session, and enables the concept of “third party control” for any operation taking place in the service session.

The OSR is meant to be established by the owning entities themselves.

Default situation: when no OSR is specified in the SSG it means that all parties (and party groups) have a shared ownership of all the object instances in the service session: no negotiation is required. The only exception to that rule is that a party SM implicitly owns itself, which includes its eventual attributes, its attachment to associations / relations with other objects and the objects it eventually aggregates. These default rules are important because it means that the initial situation in a service session (when no OSR has been defined yet) is a shared ownership. This way most of the operations can take place without negotiation.

5.4.1.4.2. Permission Session Relationship (PSR) specialization

The Permission SR class can be further specialized in read-permission and write-permission so that they can be modelled separately (see Figure 5-2). Their usage is here briefly described below.

- **Read Permission Session Relationship (RPSR)**: it associates a party SM or party SMG (a controller class object) to a controlled class object in the SSG. The read permission SR specifies whether yes or no that party SM or party SMG has the ability to see that object instance in the SSG. The visibility (or non-visibility) of that object instance includes its eventual attributes, its attachment to associations / relations with other objects, the objects it eventually aggregates, and the RPSR itself. This means that if the RPSR specifies a “read: no”, the RPSR itself will not be visible to the concerned party SM or party SMG, just like the controlled class objects it is hiding. The RPSR is meant to be established by a “third party” entity (PSM), in order for that third party to be able to hide parts of information from other parties in the service session.

Default situation: when no RPSR is specified it means that all parties (and party groups) have read permission on all the object instances in the service session. If information hiding is required, specific RPSRs have to be established.

- **Write Permission Session Relationship (WPSR):** it associates a party SM or party SMG (a controller class object) to a controlled class object in the SSG. The write permission SR specifies whether yes or no that party SM or party SMG has the ability to write on (modify) that object instance in the SSG. The modification ability (or non-ability) on that object instance means the ability to modify or delete it, which includes its eventual attributes, its attachment to associations / relations with other objects, the objects it eventually aggregates, but not the WPSR itself.The WPSR can be established by a “third party” entity (PSM), in order for that third party to be able to inhibit other parties to modify some parts of the service session.
  Default situation: when no WPSR is specified it means that all parties have write permission on all the object instances. If write inhibition is required, specific WPSRs have to be established.

5.4.1.5 Relationship between Features sets and SG information Objects

Table 5-10 summarizes the relationship between the information objects described above and the Features Set of TINA session model as described in Section 5.2.1. The possible dependencies between Feature Sets are shown in Table 5-1.

Table 5-10. Relationships between FSs and IOs

Feature Set	SSG information object classes	Allowed operation
BasicFS	Party (1)	suspend delete resume
BasicExtFS	Party (1)	suspend delete resume
MultipartyFS	Party (1+)	create suspend delete modify
MultipartyIndFS (with indications)	Party (1+)	create suspend delete modify resume
VotingFS	Party (1+)	create suspend delete modify

Feature Set	SSG information object classes	Allowed operation
ControlSRFS	ControlSR OwnershipSR PermissionSR (Write & Read) Party (1+)	create delete modify
ParticipantSBFS (Participant oriented stream binding)	SB session Relationship Stream Flow End Point Stream Interface Party (Resource)	create suspend delete modify resume
ParticipantSBIndFS (with Indications)	SB session Relationship Stream Flow End Point Stream Interface Party (Resource)	create suspend delete modify resume

5.4.1.6 Specific Types:

5.4.1.7 Exception Types for the TINA Session Model

5.4.1.7.1. e_UsageError Exception

```
// module TINAUsageCommonTypes
enum t_UsageErrorCode {
      UnknownUsageError,
      InvalidParticipantSecretId,
      UsageNotAllowed,
      UsageNotAccepted,
      UsageOpNotSupported
};

exception e_UsageError {
      t_UsageErrorCode errorCode;
};
```

The `e_UsageError` exception can be raised by all operations on interfaces defined by the TINA Session Model. It is used to inform the client of the basic problems which are experienced by the object servicing in the operation. If the exception is raised then the operation has failed, and has not performed the required action

The following error codes can be used to define the problem encountered:

- `UnknownUsageError`
  An error of an unknown type has occurred during the processing of the operation. The operation has failed, and has not performed the required action. (This error code should only be used when an error has occured which is not covered by the other error codes or exceptions, which the operation can raise. That means it shouldn't be raised frequently.)

- `InvalidParticipantSecretId`
  The `t_ParticipantSecretId` parameter does not contain a valid `t_ParticipantSecretId` for this interface. This error code should be used when the session determines that the value of the `t_ParticipantSecretId` parameter does not match the value expected.

  Some sessions do not check the `t_ParticipantSecretId` parameter and so will not raise use this error code under any circumstances.

  Other sessions that use a single interface for all request operations may use the parameter to distinguish requests from different client applications. In this case the error code will be used when the value sent does not match any of the clients values.

  Sessions that use a separate interface for each client, may check the security context, passed by the ORB along with the request, in addition to the `t_ParticipantSecretId` parameter. If the value of `t_ParticipantSecretId` does not match expected, due to the security context, then this error code will be raised.
- `UsageNotAllowed`
  The client application is not allowed to request this operation. The operation has failed, and has not performed the required action. (The reason the client is not allowed to request the operation may be due to the ControlSRFS, or may be service specific. The client may be able to use the operations defined by ControlSRFS, if the session supports them, or other service specific operations in order to allow them to request the operation subsequently.)
- `UsageNotAccepted`
  The client application request has not been accepted by the sesssion. The operation has failed, and has not performed the required action. (The reason the request is not accepted is that 'owners' of the entities the operation affects have declined to allow the operation. This is defined by the ControlSR feature set. The client may be able to use the operations defined by ControlSR, if the session supports them, or other service specific operations in order to change the ownership of the entities, or 'persuade' the owners to change their decision, to allow them to successfully complete the operation subsequently.)
- `UsageOpNotSupported`
  The operation is not supported by the session. The operation has failed, and has not performed the required action. All subsequent requests for the same operation will raise the `e_UsageError` exception with this error code.

  (This error code allows the session to support a feature set, but not to support specific operations if there is a service specific reason they cannot. e.g. for some sessions, there is no concept of suspending the session, so the session may use this error code to inform the client that the session cannot be suspended.)

5.4.1.7.2. e_PartyDomainError Exception

```
// module TINAUsageCommonTypes
enum t_PartyDomainErrorCode {
      PD_UnknownError,
      PD_InvalidSessionId,
      PD_OpNotSupported
};

exception e_PartyDomainError {
```

```
        t_PartyDomainErrorCode errorCode;
};
```

The e_PartyDomainError exception is defined for the Execution (Exe) operations in the Multiparty feature set, and other interfaces supported by the user domain. It is used by the user's application to inform provider domain session of problems performing the Exe. If this exception is raised then the application has been unable to perform the Exe, and the required action has not been performed.

Party domain components should should only raise this exception due to a problem which means they are unable to comply with the Exe operation. It should not be used to avoid performing the Exe operation because the 'user' doesn't want to. (This is as described for Exe operations in general.)

The following error codes can be used to define the problem encountered:

- PD_UnknownError
  An error of an unknown type has occurred during the processing of the operation. The operation has failed, and has not performed the required action. (This error code should only be used when an error has occurred which is not covered by the other error codes or exceptions, which the operation can raise. That means it shouldn't be raised frequently.)
- PD_InvalidSessionId
  The t_SessionId parameter does not contain a valid t_SessionId for this interface. This error code should be used when the user domain application determines that the value of the t_SessionId parameter does not match the value expected.

  Some applications may not check the t_SessionId parameter and so will not raise use this error code under any circumstances. (Though they probably should always check that the Exe operations are coming from the session they think they are.)

  Other applications that use a single interface for Exe operations from more than one provider domain session may use the parameter to distinguish operations from different sessions. In this case the error code will be used when the value sent does not match any of the sessions.

  Applications that use a separate interface for each provider domain session, may check the security context, passed by the ORB along with the request, in addition to the t_SessionId parameter. If the value of t_SessionId does not match expected, due to the security context, then this error code will be raised.
- PD_OpNotSupported
  The operation is not supported by the application. The operation has failed, and has not performed the required action. All subsequent requests for the same operation will raise the e_PartyDomainError exception with this error code.

  This error code allows the application to support a feature set, but not to support specific operations if there is a service specific reason they cannot. e.g. for some applications, there is no concept of suspending the session, so if the session invoked a suspendSessionExe() on the application, it could use this error code to inform the session that it doesn't know how to suspend. (This should be used with caution, as in general the application should perform the Exe invoked. Frequently, if the application doesn't need to perform any action, e.g. due to suspendSessionExe(), it should return successfully and not raise any exception.)

5.4.1.7.3. e_PartyError Exception

```
// module TINAUsageCommonTypes
enum t_PartyErrorCode {
      InvalidPartyId,
      InvalidPartyType,
      PartySuspended
};

exception e_PartyError {
      t_PartyErrorCode errorCode;
      t_PartyId id;
      t_PartyType type;
};
```

The e_PartyError exception is defined for operations which require a t_PartyId parameter. The t_PartyId parameter indicates the party upon which the action should be performed. If the t_PartyId is invalid, then the exception is raised with the appropriate error code. This exception can also be raised when an invalid t_PartyType parameter is passed, or the t_PartyId parameter refers to a party upon which this action cannot be performed

The following error codes can be used to define the problem encountered:

- InvalidPartyId:
  The t_PartyId parameter does not contain a valid party identifier. (This can be because the t_PartyId is wrongly formatted, or the t_PartyId given does not refer to a party in the session.)

  The t_PartyId id variable in the exception contains the value of the t_PartyId parameter passed in the operation invocation.
- InvalidPartyType:
  The t_PartyType parameter does not contain a valid party identifier. (This can be because the t_PartyType is wrongly formatted, or it isn't a party type recognised by the session.)

  The t_PartyId id variable in the exception contains the value of the t_PartyId parameter passed in the operation invocation. (The party whose party type is trying to be modified.)

  The t_PartyType type variable in the exception contains the value of the t_PartyType parameter passed in the operation invocation.
- PartySuspended:
  This operation cannot be performed because the party is suspended. It may be raised if the operation cannot be completed because the party is suspended. (e.g. a suspendPartyReq() on a party that is already suspended will raise the exception with this error code.)

  Some operations may be able to be completed even though the party identified is suspended, (e.g. endPartyReq() may succeed, and remove the party from the session, even though the party is already suspended). Such operations do not need to raise the exception with this error code. (See specific operation descriptions for details.)

5.4.1.7.4. e_AnnouncementError Exception

```
// module TINAUsageCommonTypes
enum t_AnnouncementErrorCode {
      UnknownAnnouncementError,
      InvalidAnnouncementProperty
};

exception e_AnnouncementError {
      t_AnnouncementErrorCode errorCode;
      TINACommonTypes::t_PropertyErrorStruct propertyError;
};
```

The e_AnnouncementError exception is defined for the announceSessionReq() operation. The exception is raised if the t_AnnouncementProperties are invalid.

The following error codes can be used to define the problem encountered:

- UnknownAnnouncementError:
  An error in announcing the session of an unknown type has occurred during the processing of the operation. The operation has failed, and has not performed the required action. (This error code should only be used when an error has occurred which is not covered by the other error codes or exceptions, which the operation can raise. That means it shouldn't be raised frequently.)
- InvalidAnnouncementProperty:
  The t_AnnouncementProperties parameter is in error.

  The propertyError element of the exception describes the type of error in the announcement property.

  If the propertyError contains InvalidPropertyName, then the property name is not legal for this operation. If it contains InvalidPropertyValue, then the value is not a legal value for the property name.

  If the propertyError contains UnknownPropertyName, then the session does not recogise the property name. Some sessions may ignore t_PropertyName's that they do not recognise. They should not process t_PropertyValue associated with the t_PropertyName but may process the other t_Property's in the t_AnnouncementProperties parameter. Such sessions do not need to raise the exception with this error code.

5.4.1.7.5. e_IndError Exception

```
// module TINAUsageCommonTypes
enum t_IndErrorCode {
      UnknownIndError,
      UnknownIndId,
      InvalidIndId};

exception e_IndError {
      t_IndErrorCode errorCode;};
```

This exception is defined for the Voting feature set, and other feature sets, that accept an t_IndId in order to identify a previously sent indication.

5.4.2 Stream binding terminology

Stream bindings represent multipoint-to-multipoint, multimedia connections and are used to describe service level communication requirements. TINA's stream binding model has been introduced in Section 5.4.1.3. The terminology and parameters used to describe stream bindings derive from those models and the service session graph.

This section introduces particular terminology used to describe stream bindings, expands on supporting concepts, and introduces new parameters. The stream binding feature set also uses the common parameter types described in Section 5.4.1.6 and exceptions described in Section 5.4.1.7.

5.4.2.1 Terminology:

- **Media type:** A description of the data flow types which may be associated with a stream binding or stream binding component such as a SFC or SFEP. Examples include video, audio, and data. A media type can be associated with a number of attributes, such as service quality, for a more precise description or requirements.
- **Session member:** An existing resource or party associated with a service session (from SSG). As stream bindings may be established between resources as well as parties, this term is used to indicate elements in a service session that can potentially be bound.
- **Stream binding (SB) or Stream Binding Session Relation (SBSR):** Describes a multipoint-to-multipoint multimedia connection between parties and resources.
    - **Participant Oriented SB:** A stream binding description which describes communication requirements in terms of type and participants rather than SFEPs to be connected.
- **SB controller:** Any party that establishes a control relation with the stream binding but does not offer SIs or SFEPs to be bound.
- **SB member:** Any session member associated with the stream binding as a SB participant or SB controller.
- **SB participant:** Any party or resource that offers SIs or SFEPs to be bound by the stream binding.
- **Stream binding (Overall) type:** A type label used by SB participants to determine which SFEPs they need to offer to the stream binding. The stream binding type is service dependent and is interpreted by the participants' UAPs where it may be associated with particular media types and stream binding configurations.
- **Stream flow connection (SFC):** Describes point-to-point or point-to-multipoint connections between SFEPs. A SFC supports a particular media type, and is described by branches each terminated by an SFEP. Each SFEP in an SFC must support the same media type (e.g. video) but may support different attributes of the media type, e.g. different levels of service quality or different data formats. In the latter case, the provider is responsible for locating suitable resources for translating and bridging between formats.
    - **Associated SFCs:** SFCs associated with a particular media type or participant. Each media type with in a stream binding requires the support of a group of SFCs as determined by the service logic. Any SFC that has the attributes of a media type is said to be associated with it. An SFC is also associated with all participants whose SFEPs it connects.
    - **Implicit SFCs:** The set of SFCs that support a participant oriented stream binding's communication requirements. This set of SFCs is determined by a stream binding algorithm and the service logic's knowledge of extra resources such as bridges. It is not explicitly defined by the SB members.

- **(SFEP) Bind tags:** tags associated with SFEPs and used by the service logic's stream binding algorithm to help determine which SFEPs need to be connected.
- **Service quality:** Expresses the acceptable levels of service performance. This concept is most usefully associated with particular media type: i.e. it is easier to consider performance in relation to video or audio rather than the overall performance of a stream binding which may be associated with many different media types.
- **Success criteria:** The criteria that need to be met in order for an operation on a stream binding (including its creation) to be considered successful. If an operation is not successful, any completed steps should be reversed.

5.4.2.2 Stream Binding Algorithms

When a participant oriented stream binding description specifies communication requirements, a mapping from this model to one suited to the communication layer is required. In TINA, we assume the existence of a communication session supporting communication functionality which uses SFCs to model communication requirements. The service session logic needs an algorithm that will map the information from the stream binding model to SFCs. The stream binding algorithm uses information from stream binding request and exe operations to determine which SFEPs need to be connected and hence what SFCs to generate.

This section introduces a simple algorithm that is supported by existing parameters of PaSB feature set operations. This algorithm is not mandatory, but some algorithm that produces consistent behaviour, at least for a service type, is required. A binding algorithm needs to be run after stream binding creation, modification (including withdrawal and registration of SFEPs), and rebind requests.

Simple algorithm: When a stream binding is requested, it is associated with a service dependent stream binding type. This type is interpreted by the participants' software (not by the service logic) possibly in conjunction with a service specific role that participant has (e.g. teacher or student) to determine which SFEPs are required. When a participant registers its SFEPs for use in a stream binding, they are tagged with a bind tag. The service logic can then use a binding algorithm to determine which SFEPs need to be connected by:

- matching the bind tags;
- checking that SFEP media types match[11];
- ensuring that SFEP directions (i.e. sink and source) are consistent.

The provider may need to find additional resources (e.g. bridges) to make the mapping to SFCs (e.g. it is possible to have a multiparty-to-multiparty connection which is not directly supported by SFCs).

Variations: The TINA PaSB feature set allows for variations on this simple binding algorithm by allowing requesters to specify media types and their attributes and particular participation roles.

- Media types: Media type descriptions can be used to request particular media types be supported by the stream binding and specify particular requirements, such as service quality. If media types are specified, they should be used in conjunction with the overall type by a participant's UAP to determine which SFEPs to return.

11. The media type of each SFEP in an SFC must match (e.g. all should be video).If there are a number of similar media types (e.g. presentation video and participant video in a conference service) then there may be other attributes may that require matching.However, not all attributes need match (e.g. service quality - only the terminal knows what quality it can meet; data formats - e.g. PAL/NTCS). The provider determines if the SFC be can created.

- Participant roles: The TINA PaSB feature set allows a requester to modify SB participation roles from those imposed by service specific roles. This could be used where no specific roles are specified in the service logic or a where a service specific role needs to be overridden for a particular stream binding or particular session members.
- Stream binding configuration: Additional configuration information could be included in the media type descriptions if desired. This possibility has not been investigated fully.

5.4.2.2.1. Roles

It is useful to describe behaviour with in a stream binding in terms of particular roles. Across the Ret reference point, the consumer or retailer domains are constrained to particular roles. Roles include the SB participant and SB controller roles introduced earlier.

SB participant (or party) role: The party domain components (the UAP) usually support a participant role. A participant can supply SFEPs to a stream binding and also request the creation, modification or removal stream bindings. A participant requires request interfaces from the Retailer and supports exe and information type interfaces.

SB provider role: The retailer domain components (i.e. USM or SSM) support the provider role. A provider supports requests to establish, modify or remove a stream binding. After a request, it may optionally initiate negotiations, before completing the request. A provider component typically relies on another provider component or a communication session to establish the stream binding.

SB controller role: A controller can make requests to create, modify or remove a stream binding but can not actually have SFEPs bound in a stream binding. If consumer components support this role, then they require request interfaces from the SB provider, but need not support exe interfaces. This concept may also be used to model internal logic or a composing service requesting a stream binding.

Passive participant role: A passive participant can have SFEPs bound in a stream binding but can not make requests to create, modify or delete a stream binding. Components supporting this role do not require a request type interface from the provider, but do support exe and information interfaces like a normal participant. This role would usually be assumed by resources, but may be applicable to consumers in some service specific roles.

5.4.2.3 General Stream Binding Data Types

This section introduces some widely used types. Some are discussed at greater length in Section 5.4.3, "Common Communication Session and Stream Binding Data Types".

- `t_SBType:` `// module TINAStreamCommonTypes`
  A string used to identify the stream binding type to a party's UAP. The UAP can use it to determine how many SFEPs of which types are required to support the stream binding. It is not usually interpreted by the service logic. It may be supplemented by a number of media type descriptions.
- `t_AdministrativeState`: `// module TINASBCommSCommonTypes`
  An enumerated type that indicates the administrative state of an object, see Section 5.4.3.4. An object may have a `locked` (inactive) or `unlocked` (active) state. If a stream binding member is locked, then all associated SFEPs and associated SFC branches are locked. If an SFEP is locked then it and any associated SFC branch are locked. If an SFC branch is locked, the associated resources (including SFEPs) are reserved by not in use.

- `t_MediaDesc`: `// module TINASBCommSCommonTypes`
  A complex data type used to describe and set requirements for a media type within a stream binding. Media types include audio, video, and data. A media description includes a string type label, and a list of attributes. Attributes would normally describe service level requirements such as service quality[12] or format requirements. See also Section 5.4.3.3.
- `t_MediaDescList`: `// module TINASBCommSCommonTypes`
  A sequence of media type descriptors. (A sequence is a variable length list of items.)

5.4.2.4 Participant Description Data Types

Each participant is described by an identifier, control information (e.g. ownership of stream binding), stream binding membership role, initial administrative state, success and recovery criteria, and any additional information (e.g. particular media requirements).

`// module TINAPaSBTypes::`

- `t_SBFSParticipationType`:
  An enumerated type that indicates the participation role in a stream binding The participation role modifies the stream binding type and service specific session roles of the participant (if it has one). Possible values are:
  - SBFSApplicationSpecific: Use default stream binding type and service specific role.
  - SBFSSinkAll: Sink the SFCs of any associated media types (i.e. do not a act as source).
  - SBFSSourceAll: Source the SFCs of any associated media types (i.e. do not act as a sink).
  - SBFSSinkSourceAll: For each media type, act as a sink and source
  - SBFSSinkSource: Sink or source media types as applicable.
  - SBFSAssociate: A resource, such as a bridge, that may be used to support the stream binding if necessary.
  - SBFSInitiator: An SB controller who does not participate in the stream binding.
- `t_SBControlSRType`:
  An enumerated type that indicates the control relationship between the participant and a stream binding. Control relationships are discussed in Section 5.4.1.4. This parameter sets the initial ownership and read/write permissions of the stream binding. Ownerships permissions must be set when the stream binding is created to prevent other parties gaining control of it before control relations can be set up. Possible values are:
  - DefaultSBControl: The SB member's control relationship is determined by the service logic.
  - OwnershipSBControl: The SB member has an ownership relation to the stream binding
  - ReadSBControl: The SB member has read access to the stream binding. All SB participants must have at least read permission for the stream binding.
  - WriteSBControl: The SB member has write access to the stream binding.
- `t_ParticipantId`:
  This is an element identifier previously set by the session to identify some object. Only element identifiers relating to parties, resources, or groups of these are valid in this context[13]. Participant identifiers are part of the participant description and are also used

12. It would also be possible to include communication level information (e.g. the terminal's supported capabilities) if desired when returning the media type of a SFEP. This information would not be interpreted by the service logic but passed to the communication session or equivalent. The only reason for sending it with the service information would be to make the later communication set up more efficient.

to identify participants affected by a stream binding request when no description is required (e.g. when deleting participants). If a participant identifier identifies a group, then each member of that group must be participating in the stream binding. A stream binding action on a group affects all its members (e.g. deleting a group deletes all its elements from the stream binding; adding a group adds all its elements with the same participation requirements).

- `t_ParticipantIdList`: A sequence of participant identifiers
- `t_ParticipantDesc`:
  A complex data type that describes the requested participation requirements for a particular session member. This data type includes the participant's identifier, participation role, success criteria and recovery actions (see Section 5.4.2.6), control relation to the stream binding, and initial administrative state. A string acts as an additional information parameter.
- `t_ParticipantDescList`: A sequence of participant descriptions

5.4.2.5 Stream Flow Endpoint Service Description Data Types

SFEPs are described a name unique to the consumer domain, media description (e.g. type "video", video quality "broadcast"), direction (sink or source), current administrative state, an interface reference to the associated TCSM and a bind tag. This tag is used by the stream binding algorithm to determine which SFEPs should be connected. Each SFEP may be associated with an Stream Interface (SI) which groups SFEPs. The SFEP will be given a unique session identifier for use within the session as the local SFEP name may not be unique in the session.

SFEPs are returned by SB participants in response in response to join and modify stream binding requests. The SFEPs should match the given stream binding and media types, but some requirements, such as service quality, may be varied. Before it can be used in a stream binding, an SFEP must be known to the supporting Terminal Communication Session Manager (TCSM)[14].

- `t_SFEPId` `// module TINAStreamCommonTypes`
  An session element identifier used to identify objects to the service session. This is assigned once the SFEP is registered with a session.
- `t_SFEPBindName` `// module TINAStreamCommonTypes`
  A string bind tag used by the service logic to help determine which SFEPs need to be connected with each other.
- `t_SFEPName` `// module TINASBCommSCommonTypes`
  A name (see Section 5.4.3.1) unique to the consumer domain that identifies a SFEP, and used by the communication level SFEP description.
- `t_SIName` `// module TINASBCommSCommonTypes`
  A name unique to the consumer domain that identifies a stream interface. It is used by the service level SFEP description to indicate the associated SI.
- `t_SIRef` `// module TINAStreamCommonTypes`
  A general interface reference to an SI which may be used to control data flow, if there is DPE support. An SI reference is included in the SFEP and SI descriptions for future DPE support. The SFEP's SI reference belongs to its associated SI.

13. A session relation (of any type) or a group of session relations are not valid participant types for a stream binding.
14. How this is achieved and how SFEPs are created is not part of this reference point. However, standard methods of creating and/or registering SFEPs would help the portability of application code.

- `t_SFEPComDesc // module TINASBCommSCommonTypes`
  A complex data type used to describe SFEPs at the communication level, but included in the service level description. This description includes the local identifier, the "direction" of the SFEP (i.e. sink or source), the administrative state, the media type description, and associated the TCSM interface reference.
- `t_SFEPComDescList // module TINASBCommSCommonTypes`
  A sequence of SFEP communication level descriptions.
- `t_SFEPServDesc // module TINAStreamCommonTypes`
  A complex data type used to describe SFEPs at the service level. This data type contains the communication level descriptor required by communication session level. It also includes the t_SFEPId, the t_SFEPBindName, and the associated SI identifier and interface reference.
- `t_SFEPServDescList // module TINAStreamCommonTypes`
  A sequence of SFEP service level descriptions

5.4.2.6 Success and Recovery Criteria Data Types

Success criteria are used to determine the success of creation and subsequent operations on a stream binding. Recovery criteria (or actions) are used to determine what actions to take on the failure of a stream binding during operation. Criteria may be set for the overall stream binding or for particular stream binding members or media types. Supported criteria types are listed below.

`// module TINAStreamCommonTypes::`

- `t_SBSuccessCriteria`
  The overall stream binding success criteria. This is a enumerated data type that specifies the criteria to be met for the creation of, or subsequent operations on, the stream binding to be judged successful. The value is set during stream binding set up and may be modified later. The following values are supported:
  - SBSuccessDefault: Use the service logic's default success criteria.
  - SBBestEffort: Best effort to connect as many parties to as many media types (or associated SFCs) as possible.
  - SBBEOnParties: Best effort to connect as many parties as possible to all required media types and associated SFCs. If the party can not be connected for all associated SFCs, then it will not be connected for any.
  - SBBEOnFlows: Best effort to connect as many media types and associated SFCs to all parties. An SFC will only be created if all parties can be connected.
  - SBAllOrNone: All parties will be connected for all associated SFCs or the entire stream binding operation will fail.
  - SBPartySpecific: Success criteria are set individually for each stream binding member.
- `t_ParticipantSuccess`
  The stream binding success criteria for a particular stream binding member. The following values are supported:
  - ParticipantDefault: Use default service specific stream binding success criteria for this participant.
  - ParticipantMustAll: The participant must be connected for all associated SFCs, otherwise the stream binding operation will fail.

- ParticipantMustBE: The participant must be connected for at least one associated SFC, otherwise the stream binding operation will fail.
- ParticipantBE: Best effort to connect as many associated SFCs as possible. The stream binding operation will not fail if the participant can not be connected for all or any SFCs.
- ParticipantBEAll: Best effort to connect all associated SFCs to the participant. Do not connect any SFCs if can not connect all SFCs. The stream binding operation will not fail if the participant can not be connected.

- `t_SBRecoveryCriteria`
  The overall stream binding recovery criteria. This is a enumerated data type that specifies the actions to be taken on a failure and the criteria for the recovery to be judged successful. The value is set during stream binding set up and may be modified later. The following values are supported:
  - DefaultRecovery: Use default overall recovery actions determined by service logic.
  - IgnoreFailure: Take no recovery actions on a full or partial failure.
  - ReestablishBE: Try to reestablish, best effort to reestablish previous setup. (i.e. If party or SFC can not be recovered, stream binding does not fail.)
  - ReestablishBEOnParties: Participant must have previous setup or drop it from the stream binding. This does not cause the stream binding overall to fail.
  - RestablishBEOnFlows: Recover SFCs to previous state or drop it from the stream binding.
  - ReestablishAll: Try to reestablish all flows and all participants as before or recovery fails
  - DeleteAll: Pull down rest of stream binding on partial failure
  - PartySpecificRecovery: Use specified criteria for each party.
  - FlowSpecificRecovery: Use specified criteria for a SFC or Media Type (and associated SFCs).
- `t_ParticipantRecovery`
  The recovery actions associated with a particular stream binding member and the associated success criteria. The following values are supported:
  - RecoverDefault: Use default service specific SB recovery actions for this participant.
  - RecoverMustAll: Participant must be recovered for all associated SFCs otherwise the stream binding recovery will fail.
  - RecoverMustBE: Participant must be reconnected for at least one SFC, otherwise SB recovery will fail.
  - RecoverBEAll: Best effort to recover all the participant's associated SFCs: do not reconnect any if not all SFCs recovered. The stream binding will not fail if the participant is not reconnected.
  - RecoverBE: Best effort to recover as many SFCs as possible for the participant. The stream binding will not fail if the participant can not be recovered for any or all SFCs.
  - IgnoreParticipantFailure: Ignore failure of this participant.
  - DeleteOnParticipantFailure: Delete the stream binding if this participant fails.
- `t_SBRecover`
  Overall recovery actions, including the `t_SBRecoveryCriteria` specified above and the number of retries for any recovery action. This is valid only if the boolean flag is true.

Success criteria are used at a number of stages during a stream binding request. These include:

- After the exe operations: check that returns from participants meet success criteria. There is no point continuing with a request if criteria are not met here. Examples include:
  - A stream binding request will fail if a participant that must be connected for any media type throws an exception on an exe request.
  - A stream binding request may fail if a participant must be connected for all media types and does not return SFEPs for all requested media types.
- At the SFC mapping (before communication stage): ensure that success criteria of SFCs match stream binding and participant criteria. (This aids later SB criteria checks.)
- During the communication stage: the communication session should check that the success criteria of each SFC is met. (If mapping SFCs to NFCs, it should also ensure that NFC success criteria are set appropriately as well.)
- After the communication stage: check the success criteria are met when assessing the state of the SFCs supporting the stream binding. For example:
  - The stream binding request may fail if an operation on a supporting SFC fails
  - Some SB criteria may not be met even if all operations on supporting SFCs succeed, requiring a check on the returned state of SFCs.

Recovery action and criteria are used after the failure of the stream binding or any part of it. First, these parameters are used determine what, if anything, should be done to recover communications. Then they are used to determine if the recovery actions have succeeded. If the recovery actions do not succeed, then the entire stream binding may be deleted.

5.4.2.7 Stream Binding Description Data Types

We will only consider the participant oriented stream binding description, which describe stream bindings in terms of overall type, media types, and participants. The advantage of this model is that a requester need not know details such as a potential SB participants' SFEPs which makes the creation or modification of the stream binding simpler. This stream binding data type builds on the types described previously. Other types of stream binding description are possible, for example it would be possible to describe an SB as a group of SFCs.

- `t_PaSBFSDesc // module TINAPaSBTypes`
  A complex data type that describes a stream binding in terms of its overall type (`t_SBType`), media type descriptions (`t_MediaDescList`), administrative state (`t_AdministrativeState`), success and recovery criteria (`t_SBSuccessCriteria` and `t_SBRecoveryCriteria`), and a list of SB members and their participation descriptions (`t_ParticipantDescList`). The media type descriptions identify various media types associated with the stream binding and specify service level requirements such as service quality. The `t_PaSBFSDesc` data type is used to describe a new stream binding to be setup by an `addProviderPaSBReq()` type request.

5.4.2.8 Return Data Types

These are data types that may be returned by stream binding request or information operations.

`// module TINAStreamCommonTypes::`

- `t_RequestId`:
  Identifies a request by the requester's identifier and an integer assigned by the provider. This data type supports asynchronous operations and helps in distributing stream binding state information to SB participants.

- `t_FailureCode`:
  An unsigned short integer to which an error code can be assigned.
- `t_RetElementId`:
  An identifier that maps the local identifier of an object to a session element identifier. It is especially useful for distributing SFEP and SI information to SB participants. It may also be used to return identities of items that have different types of identifier.
- `t_RetElementIdList`: a sequence of `t_RetElementId`.
- `t_FailedElementDesc`:
  Identifies an element and describes why it, or an operation on it, failed.
- `t_FailedElementDescList`:a sequence of `t_FailedElementDesc`.
- `t_SBElementFailure`:
  Describes the failure of a complex element or an operation on it. It identifies the complex element and lists associated failed elements (e.g. a participant and associated SFEPs).
- `t_SBElementFailureList`: a sequence of `t_SBElementFailure`.
- `t_SBElementSuccess`:
  Describes the success of an operation on a complex element in terms of the complex element's identifier and lists of associated elements and their success or failure state.
- `t_SBElementSuccessList`: a sequence of `t_SBElementSuccess`.
- `t_SBBindState`:
  Describes the state of a stream binding after a request. It identifies the stream binding, then lists elements successfully setup or modified by the operation, followed by elements for which the operation failed.
- `t_ProblemType`:
  An enumerated type that specifies the kind of object causing a failure: an element, operation parameter, or complex communication element.
- `t_ReqProblem`:
  A union of an element identifier, a parameter type label, and a complex failure state. It is used as additional information describing failure in a `failurePartyGSInfo`() operation.

5.4.2.9 Error Codes and Exceptions

As well as the general exceptions and error codes associated with e_BasicFSError and e_UserDomainError, PaSB feature set have a number of specific exceptions. Most exceptions indicate errors of some variety, but they are also used to support asynchronous responses.

```
// module TINAProviderPaSBUsage
exception e_NoSynchronousReqResp {
      t_RequestId;
};
```

This exception is thrown when an request is to be processed asynchronously, whether at the parties' request or because the provider does not support synchronous processing. The parameter `t_RequestId` is used to identify the request for later notification of the operation's success or failure.

```
// module TINAProviderPaSBUsage
enum t_PaSBSetupErrors{
      PaSBSetup_InvalidSBId,
```

```
        PaSBSetup_InvalidSBOp,
        PaSBSetup_UnknownSBType,
        PaSBSetup_UnknownMediaType,
        PaSBSetup_IncompatibleParameters,
        PaSBSetup_UnknownParticipantType,
        PaSBSetup_SuspendedParticipant
        PaSBSetup_UnknownParticipantId,
        PaSBSetup_UnknownCriteria,
        PaSBSetup_InvalidCriteria,
        PaSBSetup_UnsupportedCriteria,
        PaSBSetup_CriteriaNotMet,
        PaSBSetup_CommsNotAvailable,
        PaSBSetup_InsufficientBandwidth,
        PaSBSetup_QoSCannotBeMet,
        PaSBSetup_InsufficientResources,
        PaSBSetup_NoPathFound,
        PaSBSetup_UnknownSFEP,
        PaSBSetup_UnknownRFEP
};

exception e_PaSBSetupError {
        t_PaSBSetupErrors errorCode;
        t_ElementId name;
};
```

This exception is returned for errors on setup type requests to the provider. Setup requests include the creation of the stream binding, adding participants, withdrawing or registering SFEPS, and changing media types or their requirements. As well as the error code, the identity of unknown or problem entities is returned, if valid for the error type. The following error codes can be used to specify the problem encountered:

- `PaSBSetup_InvalidSBId`:
  The given stream binding identifier for the operation is not correct.
- `PaSBSetup_InvalidSBOp`:
  This stream binding operation is not valid for this stream binding.
- `PaSBSetup_UnknownSBType`:
  The given stream binding type is not known by either the other participants or the service logic and cannot be set up.
- `PaSBSetup_UnknownMediaType`:
  The given media type is not known to this session or other SB participants and can not be supported.
- `PaSBSetup_IncompatibleParameters`:
  The operation parameters are not compatible. E.g. a media type is not compatible with the stream binding type, or the service quality is not compatible with its media type.
- `PaSBSetup_UnknownParticipantId`:
  The given participant identifier is not known to the session.
- `PaSBSetup_UnknownParticipantType`
  The given participant description is not understood by the session or a participant.
- `PaSBSetup_SuspendedParticipant`:
  The given participant is currently suspended and cannot participate.

- `PaSBSetup_UnknownCriteria`:
  The given success or recovery criteria is not known to this session.
- `PaSBSetup_InvalidCriteria`:
  The given success or recovery criteria is invalid for this stream binding.
- `PaSBSetup_UnsupportedCriteria`:
  The given success or recovery criteria are not supported by this session.
- `PaSBSetup_CriteriaNotMet`:
  This operation did not meet the given (or established) success criteria.
- `PaSBSetup_CommsNotAvailable`:
  Supporting communications were not available (i.e. did not respond or not found).
- `PaSBSetup_InsufficientBandwidth`:
  Insufficient bandwidth was found to setup this stream binding.
- `PaSBSetup_QoSCannotBeMet`:
  Minimum service quality could not be met (due to lack of bandwidth, lack or participant support for the requested quality etc.).
- `PaSBSetup_InsufficientResources`:
  Not enough resources could be found to establish this stream binding (this could include service level resources such as bridges).
- `PaSBSetup_NoPathFound`:
  Paths could not be found to connect the nominated participants for these media types.
- `PaSBSetup_UnknownSFEP`:
  Could not find supporting SFEPs.
- `PaSBSetup_UnknownRFEP`:
  Could not find supporting network endpoints (RFEPs) associated with the SFEPs.

Other retailer domain stream binding exceptions give error codes that are a subset of the error codes described here. These subsets are introduced to ensure that exceptions thrown are relevant to the operation generating the exception. The following additional exceptions and related error codes are supported by stream binding providers:

```
// module TINAProviderPaSBUsage
```

- `e_PaSBOperationError  // module TINAProviderPaSBUsage`
  Used for state change and deletion errors with `t_PaSBOperationErrors` error codes.
- `e_PaSBQueryError  // module TINAProviderPaSBUsage`
  Used for state change and deletion errors with `t_PaSBQueryErrors` error codes.

The consumer domain interfaces also throw exceptions. Two sets have been identified for exe operations which have similar error types to those specified for the `e_PaSBSetupError` exception. there are no exceptions associated with information operations.

- `e_PaSBPartySetupError  // module TINAPartyPaSBUsage`
  Used for state change and deletion errors with `t_PaSBPartySetUpErrors` error codes. This exception is associated with join and modify type exe operations.
- `e_PaSBPartyExeError  // module TINAPartyPaSBUsage`
  Used for state change and deletion errors with `t_PaSBPartyExeErrors` error codes. This exception is associated with leave and state change type exe operations.

5.4.3 Common Communication Session and Stream Binding Data Types

This section describes data types common to the communication session and stream binding feature sets. Common data types include names, attributes, type descriptions, state values, and SFEPs.

5.4.3.1 Naming Data Types

Names are used to identify objects, such as SFEPs, SFCs, resources, and domains. To cope with complex naming schemes, these are implemented using a sequence of name attributes. Type name labels are used to identify data types. These are simply strings and are used for convenience. They need not follow a formal naming scheme.

```
// module TINASBCommSCommonTypes::
```

- `t_TinaName`:
  A sequence of strings that can be used to implement a formal naming scheme. The interpretation of the strings depends on the naming scheme, but generally they form pairs of name attribute types and values.
- `t_TinaNameList`: A sequence of t_TinaName.
- `t_TinaNameAttribute`:
  A string identifying a particular name attribute type in a t_TinaName.
- `t_TinaNameValue`:
  A string giving the value of a particular name attribute in a t_TinaName.
- `t_Identifier`:
  A t_TinaName used to identify resources, including SFEPs, and domains.
- `t_TypeId`:
  A string used as a type name or label (e.g. name attribute types, media types).

5.4.3.2 Attribute Data Types

Attributes are named data types that can take any data type as a value. The type of the value can be determined from the identifier.

```
// module TINASBCommSCommonTypes::
```

- `t_Attrib`:
  An attribute value pair, where the attribute type identifier is given by a `t_TinaName` and the value is given by an `any` data type.
- `t_AttribList`: A sequence of `t_Attrib`.
- `t_AttribIdList`: A sequence of attribute identifiers (i.e. of `t_TinaName`).

5.4.3.3 General Type Descriptions and Media Data Types

General type descriptions are a flexible way of describing types: they may be considered a template listing attributes and their values. Type descriptions may be used to describe complex information types which have variable attributes, such as media types associated with SFEPs and SFCs.

```
// module TINASBCommSCommonTypes::
```

- `t_TypeDesc`:
  A general type descriptor consisting of a type identifier label of type `t_TypeId` and a list of attributes (`t_AttribList`) that describe the type.

- `t_TypeDescList`: A sequence of `t_TypeDesc.`
- `t_TypeChangeDesc`:
  A descriptor that specifies how a type has changed. It consists of a type identifier label (`t_TypeId`), a list of identifying attributes, a list of new attributes that are to replace current attribute values or be added to the description, and a list of attributes that are no longer valid. Identifying and new attributes are listed in a `t_AttribList` while the attributes to be removed are identified by a `t_AttribIdList`.
- `t_TypeChangeDescList`: A sequence of `t_TypeChangeDesc.`
- `t_MediaDesc`:
  A media type descriptor based on the generic type descriptor (`t_TypeDesc`).
- `t_MediaDescList`: A sequence of `t_MediaDesc.`
- `t_MediaChangeDesc`:
  A media type change descriptor, based on the generic type change descriptor (t_TypeChangeDesc).
- `t_MediaChangeDescList`: A sequence of `t_MediaChangeDesc.`

5.4.3.4 State Data Types

These parameters describe the state of a stream binding, its implied SFCs, and other supporting objects. These are also used in the NRA and across the ConS reference point. Only the administrative state is used directly in stream binding operations. The other states in the following groups are ones that could be used in confirmation and notification operations on the information interface.

```
// module TINASBCommSCommonTypes::
```

- `t_AdministrativeState`:
  The administrative state of a resource indicates if a resource is available for use. It does not indicate if the resource is operable. Changing this state can enable or disable an object. Possible values are: `Locked` (inactive); `ShuttingDown` (changing state); and `Unlocked` (active).
- `t_OperationalState`:
  The current operational state of an object or resource that indicates if a resource is operable. Possible values are `Disabled` and `Enabled`.
- `t_UsageState`:
  The usage state of an object or resource indicates how the resource is being used. Possible values are: `Idle`, `Active`, `Busy`, and `Reserved`.
- `t_ManagementState`:
  The current operational, usage, and administrative state of an object.
- `AlarmStatus`:
  ISO alarm state which indicate the priority of an alarm. Possible values are: `UnderRepair`, `Critical`, `Major`, `Minor`, and `AlarmOutstanding.`

Other states are more closely related to resource operation and maintenance. These are not directly used by stream bindings or the communication session and include:

```
// module TINASBCommSCommonTypes::
```

- `ServiceState`: Life-cycle related state of service.
- `StandByStatus`: Indicate if a resource is being used as a standby.

- `ControlStatus`: Testing status.
- `AvailabilityStatus`: A more detailed indication of why a resource is not available.
- `ProceduralStatus`: Indicates the management activities of a resource.

5.4.3.5 Stream Flow Endpoint Communication Description Data Types

Communication level SFEP descriptions are simpler than the service level descriptions described in Section 5.4.2.5. It contains the most basic attributes used at both service and communication levels: a name, the administrative state, the direction, and the associated media type description. Service related attributes such as session identifiers and stream interfaces are not included.

```
// module TINASBCommSCommonTypes::
```

- `t_SFEPName`:
  A `t_TinaName` used to identify a SFEP at a terminal. This name needs to be unique to the consumer domain (or at least the TCSM), but may not be unique to a session.
- `t_SFEPNameList`: A sequence of `t_SFEPName`.
- `t_SFEPDirection`:
  An enumerated type that describes an SFEP's data flow direction.
  - `SFlowSink`: Data flows into the SFEP.
  - `SFlowSource`: Data flows out of the SFEP.
- `t_SFEPComDesc`:
  Specifies the name, direction and administrative state of an SFEP as described above. It also describes the associated media type with a `t_MediaDesc` type media attribute. It includes an interface reference to help with communication level SFC setup. For the TINA communication session, this interface can be cast to the TCSM's terminal flow control interface. (See also Section 5.4.2.5.)

5.4.4 Communication Session Model Information View

The communication session is concerned with the establishment of Stream Flow Connections (SFCs) to support service level stream bindings. To setup an SFC, the communication session needs to establish Terminal Flow Connections (TFCs) for each terminal between SFCs and associated Network Flow Connections (NFCs). The Ret-RP is only concerned with the interactions between the provider and party domains required to configure SFEPs and to establish and control TFCs. This section will introduce the information models supporting this interface.

Figure 5-3 shows the relation between SFCs, NFCs and TFCs. The TFC provides the links between the SFC's SFEPs and the NFC's NFEPs. Point-to-point, point-to-multipoint and bidirectional topologies are allowed. These topologies result in the following options:

- NFEP (sink/bidirectional) to one or more SFEPs (sink);
- SFEP (source) to one or more NFEPs (source/bidirectional);
- SFEP (source) to multiple SFEPs (sinks - internal branches) or NFEPs (source);
- NFEP (bidirectional) to SFEP (sink) and SFEP (source). This option allows unidirectional SFCs map to bidirectional NFCs.

The Ret-RP needs to support functionality to initiate TFCs for the above topologies, and support their modification and deletion. To initiate a TFC, the branches of the TFC need to be defined, either in terms of known SFEPs (known from the SFCs) and NFCs (the network connections to which the CSM

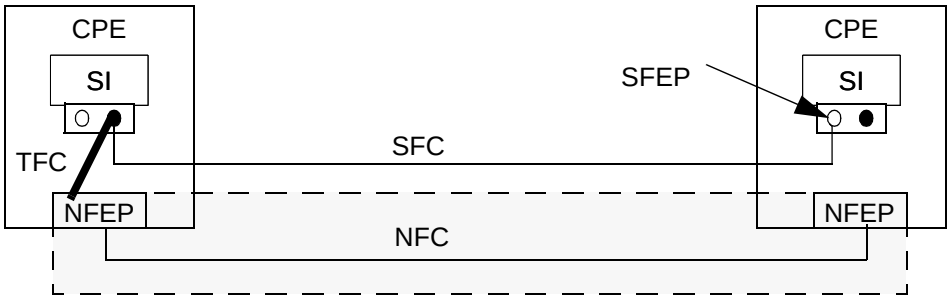


Figure 5-5. Simple relation between SFCs, NFCs and TFCs.l

intends to map the SFCs). The NFEPs are not initially known by the communication session. Instead it knows of ANfeps that represent either a NFEP or a group of NFEPs (or an ability to create NFEPs). This information is used at lower layers to select a suitable NFEP for an NFC.

As well, the Ret-RP needs to support the configuration of SFEPs to support SFC type and quality requirements. At the service level, quality is described in service related terms, e.g. FM or CD quality audio. At the communication level, these requirements are translated into a set of supporting session and terminal capabilities (e.g. codecs). The communication session needs to determine the capabilities available for each SFEP and select which ones will be used

A group of SFEPs is located on a terminal. The terminal supports the SFEPs by a capability set that consists of a list of possible capabilities and lists of simultaneously supported capabilities, see Figure 5-3. This means that using one capability may exclude using another capability for a different SFEP if the two capabilities are not in the set of simultaneously supported capabilities. Capabilities types include terminal, session, and transport related capabilities. A capability may be dependent on a number of other capabilities. Dependencies are described by sets of simultaneous dependencies which are made up of a list of capabilities that may be used as alternatives to each other.

5.4.4.1 Terminology

The following terminology is used to describe communication session functions and requirements. This is used in conjunction with terminology previously described for stream bindings.

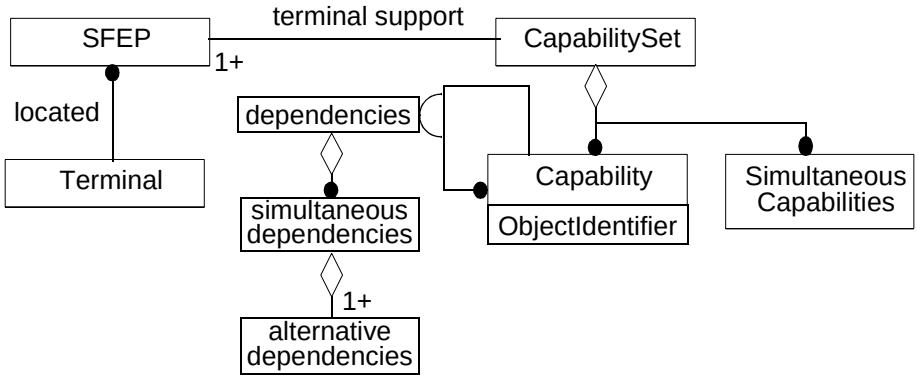


Figure 5-6. SFEPs and Supporting Capabilities.l

- **ANfep**: Represents a potential network termination. It is the super class of the NFEP and NFEPpool. An ANfep description includes layer network technology, direction, and a list of descriptive attributes. Attributes may include the transport quality and associated connectivity provider. They are used to determine which ANfeps to include in a NFC and which connectivity provider to use to setup a NFC.
- **Capability**: Describes an ability to support certain functionality, e.g., an ability to support an audio stream with a particular type of coding from a particular type of codec. Capabilities may be associated with terminal, session or transport requirements. TINA capability descriptions draw on the H.245 standards.
- **Capability Set**: Represents terminal support for SFEPs. It includes a list of capabilities associated with the SFEPs.[15] It also lists the sets of capabilities it can support simultaneously. Use of some capabilities may exclude the use of others.
- **Correlation Identifier**: Is the TFC branch identifier, unique to the terminal. It is used to correlate a NFEP (selected by connectivity layers) with the TFC branch (and hence SFEP) with which it is associated. This allows the completion of the TFC branch setup.
- **Dependencies**: Capabilities are not all independent. One capability may require a number of other types capabilities present to be used. Other capabilities may be used as alternatives to one another.
  - **Simultaneous Dependencies**: A set of capabilities types that are required together to support a capability. Each type is described by an alternative dependencies set.
  - **Alternative Dependencies**: A set of capabilities that may be used as alternatives to each other to support another capability.
- **Initiation**: The communication session may initiate a TFC or a TFC branch. However a TFC branch is not be completely set up until the ANfeps returned by the initiation step is resolved to a NFEP and this NFEP is associated with the TFC branch.
- **Completion**: The setup of a TFC branch by the association with a resolved NFEP.
- **Terminal**: equipment in the party domain terminating a SFC connection.
- **Terminal Flow Connection (TFC)**: A point-to-point or point-to-multipoint connection between a SFEP and NFEPs/SFEPs or a NFEP and SFEPs. TFCs also support bidirectional topologies between a sink and source SFEP and bidirectional NFEP. Each TFC branches represents the terminal part of a SFC branch, usually joining it to a NFC.

5.4.4.2 Communication Session related parameters

The communication session shares a number of common parameters with stream bindings. It also requires the following communication related parameters. Of these, ANfep descriptors and NFC names are also shared with the ConS parameter set.

```
// module TINACommSCommonTypes::
```

- `t_AlternativeCapabilities`: A list of mutually exclusive capabilities. Only one of the set may be used at a time. Capabilities are described by their identifiers.
- `t_AlternativeDependencies`: Describes a set of alternative capabilities on which another capability is dependent. This capability needs to be used in conjunction with one of these capabilities.

15. SFEPs of different types and service quality are associated with different capabilities - e.g. an audio SFEP is only associated with capabilities supporting audio.

`// module TINAConSCommSCommonTypes::`

- `t_ANfep`: Describes an ANfep. It gives the ANfep name, the layer technology, directionality, and the type (NFEP or NFEPpool). It also includes list of descriptive attributes that may be used to describe transport quality and other requirements. If it is an NFEPpool type ANfep, it may include a list of NFEPs or NFEPPool within the NFEPpool.
- `t_ANfepList`: A sequence of ANfep descriptions.
- `t_NFCName`: The name of a NFC to be associated with a TFC branch.

`// module TINACommSCommonTypes::`

- `t_BranchUpdate`: Describes the updates required to a TFC branch after a change of SFEP capabilities. It includes the branch's correlation identifier, the type of NFEP change required (see t_NFEPUpdate), and describes the required NFEP changes or the new connection requirements and ANfeps using a t_ANfep list.
- `t_Capability`: Describes a capability in terms of:
  - Capability description scheme identifier (e.g. ASN.1);
  - Local capability type and instance identifiers;
  - Directionality (i.e. receive, transmit, or receive and transmit associate capability);
  - Simultaneous Dependencies;
  - Descriptive attributes.
- `t_CapabilityDescriptor`: Identifies a capability and the set of simultaneously supported capability types. Each simultaneously supported type is described by a set of mutually exclusive capabilities using a t_AlternativeCapabilities data type.
- `t_CapabilityList`: A sequence of `t_Capability`.
- `t_CapabilitySet`: This data type describes a terminal's capability set. It lists the capabilities the terminal potential supports and the simultaneously available capabilities.
- `t_CorrelationId`: A TFC branch identifier unique to the terminal.
- `t_CorrelationIdList`: A sequence of correlation identifiers.
- `t_NFCCorrelation`: A correlation identifier, a NFC, and a list of ANfeps associated with a TFC branch.
- `t_NFCCorrelationList`: The correlation information returned after initiating a SFEP to multiple NFEP TFC or new branches of such a TFC.

`// module TINACommSCommonTypes`

- `t_NFEPUpdate`: Describes the updates required to TFC branches' supporting NFEP after a change of SFEP capabilities. Possible changes are: no change, modify the existing NFEP, or select a new NFEP.
- `t_SFEPCorrelation`: A correlation identifier and SFEP associated with a TFC branch.
- `t_SFEPCorrelationList`: The correlation information returned after initiating a NFEP to multiple SFEP TFC or new branches of such a TFC.
- `t_SFEPSelect`: A SFEP and it required capability lists. The capability list describes the capabilities to be associated with the SFEP. These need to be reserved for its use.
- `t_SFEPSelectList`: A sequence of `t_SFEPSelect`.

- `t_SimultaneousDependencies`: Describes a set of capability types on which another capability is dependent. This capability needs to be used in conjunction with all of these capability types. The capability types are described by `t_AlternativeDependencies`.
- `t_TFCName`: A TFC identifier, unique in the party domain. It is set by the TCSM on the initiation of a TFC.

5.5 TINA Service Session Model Feature Sets

This section gives a detailed description of the feature sets, currently defined for the TINA Service Session Model.

The first subsection (i_SessionModels interface) does not define a feature set, but describes an interface that is inherited by interfaces in two of the feature sets (BasicFS and BasicExtFS).

5.5.1 i_SessionModels interface

`i_SessionModels` interface provides generic operations for accessing the session models supported by a domain. It is inherited into interfaces on the Basic and BasicExt feature sets.

The following descriptions use the terms client and server domains. this is because this interface is inherited into interfaces supported by both the party and provider domains. When the party domain is the client, then the provider domain is the server, and vice versa.

```
// module TINASessionModel
interface i_SessionModel {};
    void getSessionModels (
          out TINACommonTypes::t_SessionModelList sessionModels
    );
```

`getSessionModels`() allows the client to find out the session models supported by the server domain. `sessionModels` contains a list of all the session models supported by the server domain.

```
    void setSessionModel (
          in TINACommonTypes::t_SessionModel requestedSM,
          out TINACommonTypes::t_SessionModelList supportedSMs
    ) raises (
          TINACommonTypes::e_SessionModelError
    );
```

`setSessionModel`() allows the client to tell the server domain to use a particular session model. The server domain will use only that session model, and not use any of the other session models that it supports. The client can only tell the server to use a model that it supports.

When the session model is set, the server domain assumes that the client domain is also using the same session model. In general, sessions are likely to only support a single session model, and so will not have much choice in the session model that is used. Before this operation is invoked, or if this operation is not called, then the server must support operations on all the session models it supports, (or fail to perform the operations of any of them.)

```
    void registerSessionModel (
          in TINACommonTypes::t_SessionModel clientModel
    ) raises (
          TINACommonTypes::e_SessionModelError
    );
```

`registerSessionModel`() allows the client to inform the server domain of the session models that it supports. It does not ‘force’ the server to use the same session model as `setSessionModel`() does.

5.5.2 Basic Feature Set

The Basic feature set (“BasicFS”) must be supported by all sessions which support the TINA session model.

BasicFS provides sufficient functionality to control a single party session. It allows a client application in the party’s domain to:

- end and suspend the session;
- to discover the interfaces, session models and feature sets supported by the session;
- to retrieve the interfaces supported by the session, (both service specific and those supporting a particular feature set);
- to register the client’s own interfaces and session models with the session.

Table 5-11. BasicFS Interfaces

BasicFS interfaces on:	
Party domain components	(none)
Provider domain components	i_ProviderBasicReq

BasicFS requires that the provider domain components support the following interface:

- **`i_ProviderBasicReq`**: which inherits from the `i_ProviderInterfaces`, and `i_SessionModels` interfaces. It additionally supports operations to end and suspend a session.

```
// module TINAProviderBasicUsage

interface i_ProviderBasicReq
                : i_ProviderInterfaces,
                TINASessionModel::i_SessionModel
{
};
```

The following are the inherited interfaces:

- `i_ProviderInterfaces`: allows a client application in the party domain to discover the interfaces, and interface types, supported by the provider domain components. (This includes the service-specific interfaces supported.) Also allows client to register the client's interfaces, and interface types, with the session. See Section 5.5.3
- `i_SessionModels`: allows client to discover the session models/feature sets supported by the session, and to get the interfaces associated with those feature sets. Also allows the client to register its session models, feature sets and interfaces with the session. See Section 5.5.1

BasicFS supports a client-server paradigm, where the party domain’s components are the clients, and the provider domain components are the server. All requests using BasicFS interfaces are initiated from the client applications, and are serviced by the provider components.

This means that the simplest service that can be implemented using BasicFS is a single party service, that has interfaces on the provider domain components, with no interfaces on the party domain applications.

This does not mean that services using Basic feature set are restricted to single-party services, or only client-server interactions. Using Basic feature set, multiple parties can discover session models and interfaces, and register their own. Also, client applications can have interfaces, and register these with the provider domain components.

5.5.2.1 endSessionReq()

```
void endSessionReq (
        in TINACommonTypes::t_ParticipantSecretId myId
) raises (
        TINAUsageCommonTypes::e_UsageError
);
```

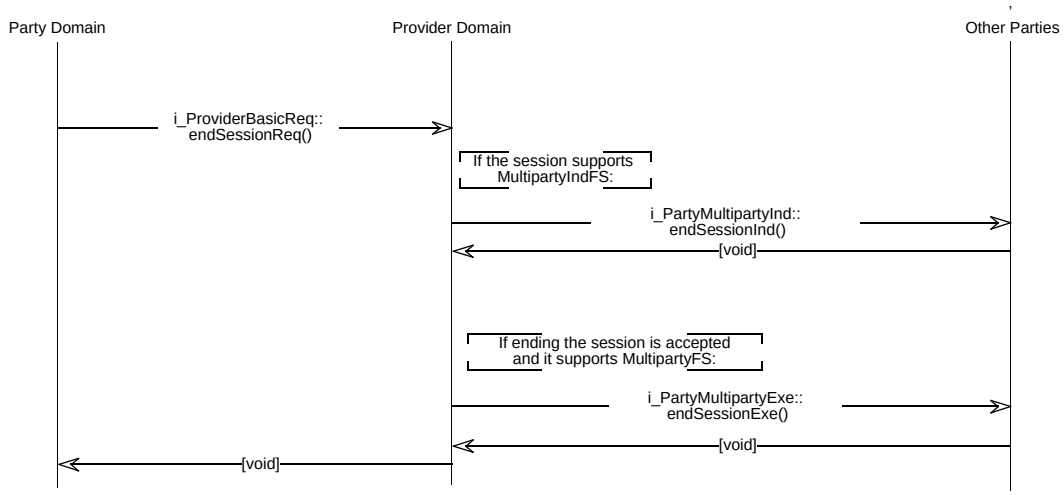


Figure 5-7. End Session event trace.

The `endSessionReq()` allows a participant to request that the session end. The requesting participant's `t_ParticipantSecretId` is sent as the `myId` parameter.

After this request completes successfully, the session will end. The session parties' will no longer be able to make invocations on any of the interfaces belonging to this session. The session will not make any invocations on interfaces of any of the parties. The `t_SessionId` of this session will be released by the access components of the provider. It can no longer be used to refer to this session through the Access part of Ret-RP.

If this is a single party session, then the session will end. (See exceptions below.)

If this is a multiparty session, then the session may send `endSessionInd()` to the `i_PartyMultipartyInd` interface of all the other parties in the session, (if the parties support MultipartyInd feature set), and may wait for votes to be received, (if the session supports Voting feature set). (Precisely who is sent `endSessionInd()` may be determined by ControlSR feature set, or be service specific.)

If session doesn't support Voting feature set, or if the request is agreed (see Voting feature set for details), then endSessionExe() will be sent to the i_PartyMultipartyExe interface of all parties except the requesting party.

If, after the endSessionExe()operations return, the request has been successful, then the endSessionReq() will return successfully, and the session ends. endSessionReq() can only return success, when there is no possiblity that the session will abort the end action.

Exceptions:

When an exception is raised for whatever reason, endSessionReq() has failed and the session will not be ended.

If myId parameter is not recognised, then an e_UsageError exception with the InvalidParticipantSecretId error code will be raised.

If the session decides that this participant is NOT allowed to end the session (e.g. due to ControlSR feature set; voting or a service specific reason), then a e_UsageError exception with the UsageNotAllowed error code will be raised.

If the operation is unsuccessful for any other reason, then a e_UsageError exception with the UnknownUsageError error code will be raised.

This operation should not raise the e_UsageError exception with the UsageOpNotSupported error code, as this operation must always be supported by every session.

5.5.2.2 suspendSessionReq()

```
void suspendSessionReq(
          in TINACommonTypes::t_ParticipantSecretId myId,
          out TINACommonTypes::t_SessionId sessionId
) raises (
          TINAUsageCommonTypes::e_UsageError
);
```

The suspendSessionReq() allows a party to request that the session is suspended. The requesting participant's t_ParticipantSecretId is sent as the myId parameter.

After this request completes successfully, the session will be suspended. The session parties will no longer be able to make invocations on any of the interfaces belonging to this session. The session will not make any invocations on interfaces of any of the parties. The t_SessionId of this session will be continue to refer to this session through the access componets of the provider. It can be used to resume the session through the Access part of Ret RP.

If this is a single party session, then the session will suspend. (See exceptions below.)

If this is a multiparty session, then the session may send suspendSessionInd() to the i_PartyMultipartyInd interface of all the other parties in the session, (if the parties support MultipartyInd feature set), and may wait for votes to be received, (if the session supports Voting feature set). (Precisely who is sent suspendSessionInd() may be determined by ControlSR feature set, or be service specific.)

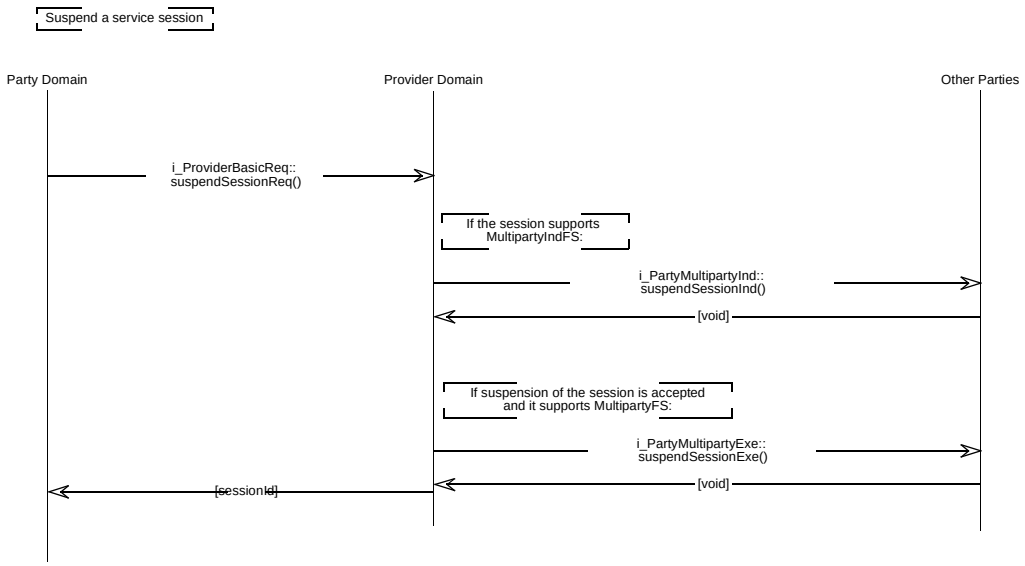


Figure 5-8. Suspend Session event trace.

If session doesn't support Voting feature set, or if the request is agreed (see Voting feature set for details), then `suspendSessionExe()` will be sent to the `i_PartyMultipartyExe` interface of all parties except the requesting party.

If, after the `suspendSessionExe()`operations return, the request has been successful, then the `suspendSessionReq()` will return successfully, and the session will be suspended.

Exceptions:

When an exception is raised for whatever reason, `suspendSessionReq()` has failed and the session will not be suspended.

If `t_ParticipantSecretId myId` parameter is not recognised, then an `e_UsageError` exception with the `InvalidParticipantSecretId` error code may be raised.

If the session decides that this participant is NOT allowed to suspend the session (e.g. due to ControlSR feature set; voting or a service specific reason), then a `e_UsageError` exception with the `UsageNotAllowed` error code will be raised.

If the operation is unsuccessful for any other reason, then a `e_UsageError` exception with the `UnknownUsageError` error code will be raised.

Some services may not support suspending of the session. If this is the case, the session must still provide an implementation of the `suspendSessionReq()` operation, however it must always raise the `e_UsageError` exception with the `UsageOpNotSupported` error code. It must not perform any other action.

5.5.3 i_ProviderInterfaces interface and inherited interfaces

`i_ProviderInterfaces` interface provides generic operations for accessing the interfaces exported by the provider domain, as part of a service session.

Three interfaces are defined.

- `i_ProviderGetInterfaces` - This interface allows the client to get interfaces exported by this domain.
- `i_ProviderRegisterInterfaces` - This interface allows the client to register interfaces exported by the client domain.
- `i_ProviderInterfaces` - This interface inherit from the other two, and so allows the client to get interfaces exported by this domain, and register interfaces exported by the client domain.

None of the interfaces should be exported directly by the provider domains. They can be inherited into another interface which is defined as being exported across Ret RP. For the Basic feature set, the `i_ProviderInterfaces` interface is inherited into the `i_ProviderBasicReq` interface. Other feature sets, or service specific interface can also inherit these interfaces in the same way.

5.5.3.1 i_ProviderGetInterfaces interface

```
// module TINAProviderBasicUsage

interface i_ProviderGetInterfaces
{
};

void getInterfaceTypes (
      in TINACommonTypes::t_ParticipantSecretId myId,
      out TINACommonTypes::t_InterfaceTypeList itfTypeList
) raises (
      TINAUsageCommonTypes::e_UsageError
);
```

`getInterfaceTypes`() allows the party domain to retrieve the interface types supported by the provider domain components. `itfTypeList` includes all the interface types that the provider domain components offer to the party domain, including all the service specific interface types.

```
void getInterface (
      in TINACommonTypes::t_ParticipantSecretId myId,
      in TINACommonTypes::t_InterfaceTypeName type,
      in TINACommonTypes::t_MatchProperties desiredProperties,
      out TINACommonTypes::t_InterfaceStruct itf
) raises (
      TINAUsageCommonTypes::e_UsageError,
      TINACommonTypes::e_InterfacesError,
      TINACommonTypes::e_PropertyError
);
```

`getInterface`() allows the party domain to retrieve an interface of a given type, supported by the provider domain components. Only a single interface reference is returned. If multiple interfaces of the given type are supported, then the `desiredProperties` parameter can be used to select one them to be returned. If all the interfaces match the `desiredProperties`, then an interface is chosen at random. (See `t_MatchProperties` in Section 3.3.1).

```
void getInterfaces (
      in TINACommonTypes::t_ParticipantSecretId myId,
      out TINACommonTypes::t_InterfaceList itfList
) raises (
      TINAUsageCommonTypes::e_UsageError
);
```

`getInterfaces`() allows the party domain to retrieve a list of interfaces supported by the provider domain components.

The `desiredProperties` parameter can be used to scope the list of interfaces. `t_MatchProperties` identifies the properties which the interface must match. It also defines whether an interface must match one, all or none of the properties. (See `t_MatchProperties` in Section 3.3.1). If 'none' properties are to be matched, then all the interfaces supported by the provider domain components are returned. Currently no specific property names and values have been defined, and so its use is service specific.

5.5.3.2 i_ProviderRegisterInterfaces interface

```
// module TINAProviderBasicUsage
interface i_ProviderRegisterInterfaces
{
};
```

This interface allows the party domain to registering interfaces with the provider domain components. Registration allows the party domain to inform the provider domain about interfaces it supports. It allows the interfaces to be registered and later unregistered. The provider domain is only allowed to use the registered notifies reference between these times.

Registration of interface types is slightly different, in that no references to the interface types are given to the provider domain. This allows the provider domain to know about the types of interfaces which the party domain supports, but retrieve interface references through another means, (such as the BasicExt feature set, (Section 5.5.4)).

```
void registerInterface (
      in TINACommonTypes::t_ParticipantSecretId myId,
      in TINACommonTypes::t_InterfaceStruct itf,
      out TINACommonTypes::t_InterfaceIndex itfIndex
) raises (
      TINAUsageCommonTypes::e_UsageError,
      TINACommonTypes::e_InterfacesError
);
```

`registerInterface`() allows the party domain to inform the provider domain of an interface it supports. The party domain should continue to support this interface until it unregisters the interface.

itf describes the interface type, reference and properties being registered.

itfIndex is used to identify the interface, and is used when unregistering the interface, (see unregisterInterface() below.)

```
void registerInterfaces (
      in TINACommonTypes::t_ParticipantSecretId myId,
      inout TINACommonTypes::t_RegisterInterfaceList itfs
) raises (
      TINAUsageCommonTypes::e_UsageError,
      TINACommonTypes::e_InterfacesError,
      TINACommonTypes::e_RegisterError
);
```

registerInterfaces() allows the party domain to register multiple interfaces with the provider domain.

itfs is a list of interfaces, including the type, reference and properties of each interface. It is an inout parameter, as the list also includes an t_InterfaceIndex with each interface. The value for the t_InterfaceIndex is set by the operation, and returned to the party domain. The party domain can then use the t_InterfaceIndex associated with each interface description to unregister the appropriate interface as necessary.

```
void registerInterfaceTypes (
      in TINACommonTypes::t_ParticipantSecretId myId,
      in TINACommonTypes::t_InterfaceTypeList types
) raises (
      TINAUsageCommonTypes::e_UsageError,
      TINACommonTypes::e_InterfacesError
);
```

registerInterfaceTypes() allows the party domain to inform the provider domain of the interface types it supports.

types includes all the interface types that the party domain component wish to offer to the provider domain, including service specific interface types. It does not have to include all the types of interfaces the party domain can offer, just the ones it wishes to inform the provider domain about.

```
void listRegisteredInterfaces (
      in TINACommonTypes::t_ParticipantSecretId myId,
      out TINACommonTypes::t_RegisterInterfaceList registeredItfs
) raises (
      TINAUsageCommonTypes::e_UsageError
);
```

listRegisteredInterfaces() allows the party domain to find out the interfaces which have been registered with the provider domain.

registeredItfs is the list of interfaces which this party domain has registered with the provider domain. Note: it does NOT include interfaces registered by other party domains.

```
    void unregisterInterface (
          in TINACommonTypes::t_ParticipantSecretId myId,
          in TINACommonTypes::t_InterfaceIndex index
    ) raises (
          TINAUsageCommonTypes::e_UsageError,
          TINACommonTypes::e_InterfacesError,
          TINACommonTypes::e_UnregisterError
    );
```

unregisterInterface() allows the party domain to unregister an interface with the provider domain. Once the interface has been unregistered, the provider domain should not longer make use of that interface reference.

index is the t_InterfaceIndex which identifies the interface to be unregistered. The t_InterfaceIndex is returned when the interface is registered, or by listRegisteredInterfaces().

```
    void unregisterInterfaces (
          in TINACommonTypes::t_ParticipantSecretId myId,
          in TINACommonTypes::t_InterfaceIndexList indexes
    ) raises (
          TINAUsageCommonTypes::e_UsageError,
          TINACommonTypes::e_InterfacesError,
          TINACommonTypes::e_UnregisterError
    );
```

unregisterInterfaces() allows the party domain to unregister multiple interfaces at a time from the provider domain. Once the interface has been unregistered, the provider domain should not longer make use of that interface reference.

5.5.3.3 i_ProviderInterfaces interface

```
    // module TINAProviderBasicUsage

interface i_ProviderInterfaces
                  : i_ProviderGetInterfaces,
                  i_ProviderRegisterInterfaces
    {
    };
```

This interface does not define any new operations, it merely inherits all the operations from i_ProviderGetInterfaces and i_ProviderRegisterInterfaces.

5.5.4 BasicExt Feature Set

The Basic Extended feature set (“BasicExtFS”) allows provider domain components to discover interfaces and session models supported by the party domains’ components.

BasicExtFS is an optional feature set, which requires that the session also support BasicFS . .

Table 5-12. BasicExtFS Interfaces

BasicExtFS interfaces on:	
Party domain components	i_PartyBasicExtReq
Provider domain components	(none)

BasicExtFS requires that party domain components support the following interface:

- `i_PartyBasicExtReq`: which inherits from the `i_PartyGetInterfaces`, and `i_SessionModels` interfaces.

The `i_PartyGetInterfaces` interface is described below in Section 5.5.4.1.

The `i_SessionModels` interface is described in Section 5.5.1

BasicExtFS is an extension to the BasicFS . It supports the opposite of the client-server paradigm, in that the provider domain can gain information about the session models and interfaces supported by the party domain components.

It does not support any session control operations, such as ending or suspending the session, from the providers domain.

```
// module TINAPartyBasicExtUsage

interface i_PartyBasicExtReq
                : i_PartyGetInterfaces,
                TINASessionModel::i_SessionModels
{
};
```

5.5.4.1 i_PartyGetInterfaces interface

i_PartyGetInterfaces interface provides generic operations for accessing the interfaces exported by the party domain, as part of a service session.

The following interface is defined.

- `i_PartyGetInterfaces` - This interface allows the provider to get interfaces exported by this domain.

This interfaces should be exported directly by the party domain. It is inherited into the `i_PartyBasicExtReq` interface which is defined as being exported across Ret RP.

```
    // module TINAPartyBasicExtUsage

interface i_PartyGetInterfaces
    {
    };

    void getInterfaceTypes (
          in TINACommonTypes::t_SessionId sessionId,
          out TINACommonTypes::t_InterfaceTypeList itfTypeList
    ) raises (
          TINAUsageCommonTypes::e_UsageError);
```

`getInterfaceTypes`() allows the provider domain to retrieve the interface types supported by the party domain components. `itfTypeList` includes all the interface types that the party domain components offer to the provider domain, including all the service specific interface types.

```
    void getInterface (
          in TINACommonTypes::t_SessionId sessionId,
          in TINACommonTypes::t_InterfaceTypeName type,
          in TINACommonTypes::t_MatchProperties desiredProperties,
          out TINACommonTypes::t_InterfaceStruct itf
    ) raises (
          TINAUsageCommonTypes::e_UsageError,
          TINACommonTypes::e_InterfacesError,
          TINACommonTypes::e_PropertyError);
```

`getInterface`() allows the provider domain to retrieve an interface of a given type, supported by the party domain components. Only a single interface reference is returned. If multiple interfaces of the given type are supported, then the `desiredProperties` parameter can be used to select one them to be returned. If all the interfaces match the `desiredProperties`, then an interface is chosen at random. (See `t_MatchProperties` in Section 3.3.1).

`t_InterfaceTypeName` identifies the type of the interface to be returned.

```
    void getInterfaces (
          in TINACommonTypes::t_SessionId sessionId,
          in TINACommonTypes::t_MatchProperties desiredProperties,
          out TINACommonTypes::t_InterfaceList itfList
    ) raises (
          TINAUsageCommonTypes::e_UsageError
          TINACommonTypes::e_PropertyError);
```

`getInterfaces`() allows the provider domain to retrieve a list of interfaces supported by the party domain components.

The `desiredProperties` parameter can be used to scope the list of interfaces. `t_MatchProperties` identifies the properties which the interface must match. It also defines whether an interface must match one, all or none of the properties. (See `t_MatchProperties` in Section 3.3.1). If ‘none’ properties are to be matched, then all the interfaces supported by the party domain components are returned. Currently no specific property names and values have been defined, and so its use is service specific.

5.5.5 Multiparty Feature Set

The Multyparty feature set (“MultipartyFS”) allows the session to support multiparty services. MultipartyFS is an optional feature set, which requires that the session also support BasicFS. It supports the party domain components making requests for generic multiparty control actions, such as suspending a party’s participation in the session. It also supports the session providing information on events that have happened to other participants, eg. another party has suspended; and the session asking the party domain components to execute an action, (eg. the user of a UAP is being suspended, and the UAP needs to perform some action before they are suspended.)

Table 5-13. MultipartyFS Interfaces

MultipartyFS interfaces on:	
Party domain components	i_PartyMultipartyExe i_PartyMultipartyInfo (optional)
Provider domain components	i_ProviderMultipartyReq

MultipartyFS requires that the provider domain components support the following interface:

- `i_ProviderMultipartyReq`: supports operations to:
  - request details on other participants in the session;
  - request the end of a party’s participation in the session;
  - request the suspension of a party’s participation in the session;
  - request that a user is invited to join the session;
  - request that the session is announced, in some manner.

MultipartyFS requires that party domain components support the following interface:

- `i_PartyMultipartyExe`:
  supports the execution of generic session control actions, (such as the operations on `i_ProviderMultipartyReq`, and `i_ProviderBasicReq`. Operations ask the party domain component to execute an action for this participant, due to a change in the session state, such as:
  - changing this participant’s party type;
  - ending the session; or this participant’s participation in the session;
  - suspending the session; or this participant’s participation in the session.
- `i_PartyMultipartyInfo`:
  supports the provider domain informing the party domain components of changes in the session state. Party domain components do not have to support this interface. If they do support this interface, and register it with the provider domain, then it will call operations on this interface to inform the party domain of changes in the session, such as:
  - changes to another party’s type;
  - another party having ended/suspended/resumed their participation in the session;
  - another party having joined the session;
  - a user having been invited to join the session;
  - the session having been announced, in some manner.

MultipartyFS provides operation to request and execute generic multiparty control actions, such as suspending a party's participation in the session. It does not define whether a particular party is allowed to perform the action. The operations are defined with exceptions that can be raised, if the session determines that a request for an action it not allowed. It is up to the session, and possibly parties to decide if an action should be allowed, or should not be performed.

The session may use entirely service specific mechanisms to decide if an action should be performed. Alternatively, ControlSR feature set may be used to associate owners to session entities, which determine if an action is allowed. Also MultipartyInd and Voting feature sets allow the session to indicate to party domain components that an action has been requested, and to vote on whether an action is performed. All three may also be used together to determine which parties are indicated about the action, and which can vote.

Once an action has been performed and Exe messages sent, party domain components that have passed a reference of their `i_PartyMultipartyInfo` to the provider domain will receive an Info message. All party domains that have registered their interface will receive the Info messages, unless the ControlSR feature set defines who should receive these messages.

```
// module TINAProviderMultipartyUsage
```

interface i_ProviderMultipartyReq

```
{
};
```

5.5.5.1 listParties()

```
void listParties (
      in TINACommonTypes::t_ParticipantSecretId myId,
      out TINACommonTypes::t_PartyIdList partyIdList
) raises (
      TINAUsageCommonTypes::e_UsageError
);
```

The `listParties()` allows a participant to request the id's of the parties in the session. The requesting participant's `t_ParticipantSecretId` is sent as the `myId` parameter.

The request returns a list of the party ids' of all parties currently in the session.

Exceptions: See `e_UsageError` exception description.

5.5.5.2 listPartiesWithDetails()

```
void listPartiesWithDetails (
      in TINACommonTypes::t_ParticipantSecretId myId,
      out TINAUsageCommonTypes::t_PartyDetailsList partyDetailsList
) raises (
      TINAUsageCommonTypes::e_UsageError
);
```

The `listPartiesWithDetails()` allows a participant to request the party details of the parties in the session. The party's `t_ParticipantSecretId` is sent as the `myId` parameter.

The request returns a list of the party details' of all parties currently in the session.

Exceptions: See `e_UsageError` exception description.

5.5.5.3 getPartyDetails()

```
void getPartyDetails (
      in TINACommonTypes::t_ParticipantSecretId myId,
      in TINACommonTypes::t_PartyId partyId,
      out TINAUsageCommonTypes::t_PartyDetailsList partyDetailsList
) raises (
      TINAUsageCommonTypes::e_UsageError,
      TINAUsageCommonTypes::e_PartyError
);
```

The `getPartyDetails()` allows a participant to request the party details of the specified party in the session. The requesting participant's `t_ParticipantSecretId` is sent as the `myId` parameter. The `t_PartyId`, of the party's details to be retrieved, is sent as the `partyId`.

The request returns the party detail's of the party identified by `partyId`.

Exceptions:

If `partyId` is not a valid id of any party in the session, then an `e_PartyError` exception with the `InvalidPartyId` error code will be raised.

See `e_UsageError` exception description for other reasons for raising exceptions and the error codes to use.

5.5.5.4 getMyPartyDetails()

```
void getMyPartyDetails (
      in TINACommonTypes::t_ParticipantSecretId myId,
      out TINAUsageCommonTypes::t_PartyDetails myDetails
) raises (
      TINAUsageCommonTypes::e_UsageError
);
```

The `getMyPartyDetails()` allows a participant to request their own party details. The requesting participant's `t_ParticipantSecretId` is sent as the `myId` parameter.

The request returns the party detail's of the requesting participant. (This is useful in case the participant 'looses' their party id, and so can retrieve their own party id based on their `t_ParticipantSecretId myId`).

Exceptions:

If the requesting participant is not a party (i.e. they are a resource, or something else) in the session, then an `e_PartyError` exception with the `InvalidPartyId` error code will be raised.

See `e_UsageError` exception description for other reasons for raising exceptions and the error codes to use.

5.5.5.5 modifyPartyTypeReq()

```
void modifyPartyTypeReq (
      in TINACommonTypes::t_ParticipantSecretId myId,
      in TINACommonTypes::t_PartyId partyId,
      in TINAUsageCommonTypes::t_PartyType newType
) raises (
      TINAUsageCommonTypes::e_UsageError,
      TINAUsageCommonTypes::e_PartyError
);
```

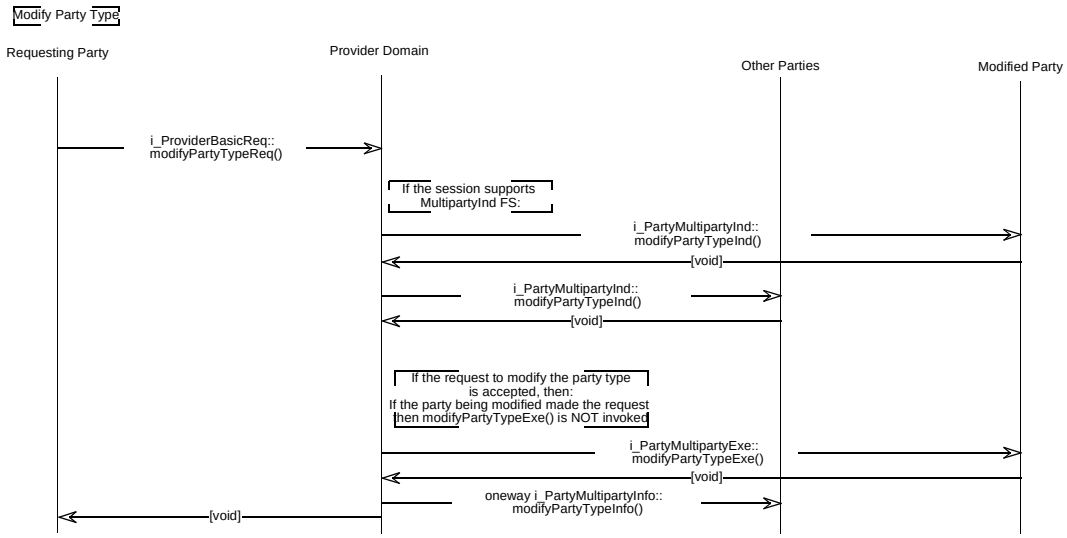


Figure 5-9. Modify Party Type event trace.

The `modifyPartyTypeReq()` allows a participant to request that a party's type is modified. The requesting participant's `t_ParticipantSecretId` is sent as the `myId` parameter. The `t_PartyId`, of the party's type to be modified, is sent as the `partyId`. The `newType` parameter contains the new `t_PartyType` to be assigned to the party.

After this request completes successfully, the party, identified by `partyId`, will have their party type as `newType`.

The session may send `modifyPartyTypeInd()` to the `i_PartyMultipartyInd` interface of the party domains representing all the other parties in the session, (if they support MultipartyInd feature set), and may wait for votes to be received, (if the session supports the Voting feature set). (Precisely who is sent `endSessionInd()` may be determined by ControlSR feature set, or be service specific.)

If the requesting party is the party who's type is to be modified, then they will not be sent a `endSessionInd()`. However other parties may still receive `endSessionInd()`, and may be able to vote, (as above).

If session doesn't support VotingFS, or if the request is agreed (see Section 5.5.7 for details), then `modifyPartyTypeExe()` will be sent to the `i_PartyMultipartyExe` interface of the party identified by `partyId`. (No `modifyPartyTypeExe()` will be sent if the requesting party is the identified party.)

If the modifyPartyTypeExe() operation returns successfully, then the session will send modifyPartyTypeInfo() to the i_PartyMultipartyInfo interface of all the party domain components that have registered this interface with the session.

If at this stage the request has been successful, then the modifyPartyTypeReq() will return successfully, and the party's type will have been changed.

Exceptions:

When an exception is raised for whatever reason, modifyPartyTypeReq() has failed and the party's type will NOT be modified.

If partyId is not a valid id of any party in the session, then an e_PartyError exception with the InvalidPartyId error code will be raised.

If newType is not a valid party type, then an e_PartyError exception with the InvalidPartyType error code will be raised.

See e_UsageError exception description for other reasons for raising exceptions and the error codes to use.

5.5.5.6 endMyParticipationReq()

```
void endMyParticipationReq (
      in TINACommonTypes::t_ParticipantSecretId myId
) raises (
      TINAUsageCommonTypes::e_UsageError
);
```

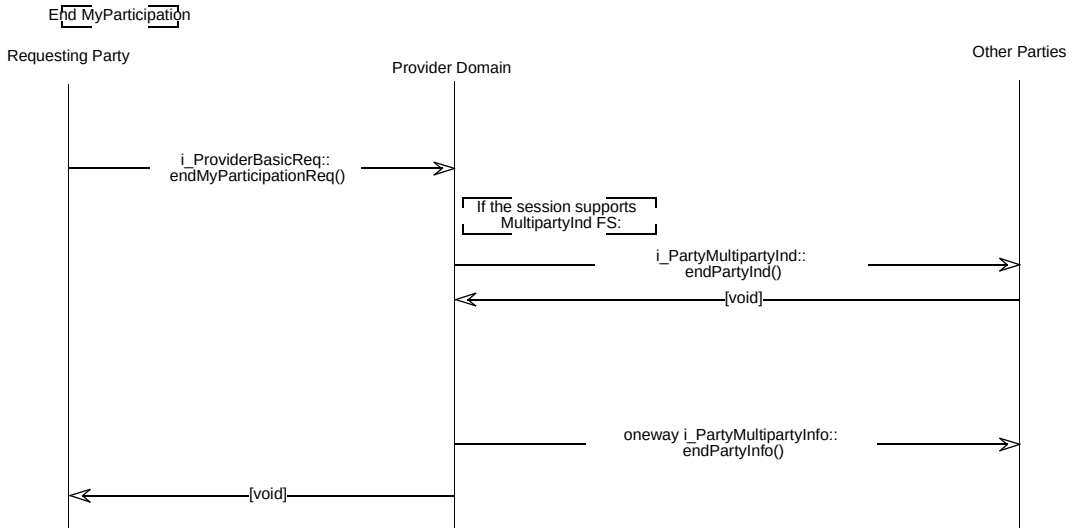


Figure 5-10. End My Participation event trace.

The endMyParticipationReq() allows a participant to request that their participation in the session end. The requesting participant's t_ParticipantSecretId is sent as the myId parameter.

After this request completes successfully, the participant's participation in the session will have ended. The participant will no longer be able to make invocations on any of the interfaces belonging to this session. The session will not make any invocations on interfaces of the participant's domain components.

The session may send `endPartyInd()` to the `i_PartyMultipartyInd` interface of the components representing all the other parties in the session, (if the parties support MultipartyIndFS), and may wait for votes to be received, (if the session supports VotingFS). (Precisely who is sent `endPartyInd()` may be determined by ControlSR feature set, or be service specific.)

If session doesn't support VotingFS, or if the request is agreed (see Section 5.5.7 for details), then `endParticipationInfo`() will be sent to the `i_PartyMultipartyInfo` interface of all the components that have registered this interface with the session.

If at this stage the request has been successful, then the `endParticipationReq()` will return successfully, and the participant's participation in the session will have ended. This operation can return success when the action of ending the party's participation cannot be aborted. This effectively means that the request can return successfully after all the `endPartyInd()` operations have returned, and any voting, if necessary, is complete.

Exceptions:

When an exception is raised for whatever reason, `endMyParticipationReq()` has failed and the participant's participation in the session will not be ended.

See `e_UsageError` exception description for reasons for raising exceptions and the error codes to use.

5.5.5.7 endPartyReq()

```
void endPartyReq (
      in TINACommonTypes::t_ParticipantSecretId myId,
      in TINACommonTypes::t_PartyId endPartyId
) raises (
      TINAUsageCommonTypes::e_UsageError,
      TINAUsageCommonTypes::e_PartyError
);
```

The `endPartyReq()` allows a party to request that another party's participation in the session is ended. The requesting participant's `t_ParticipantSecretId` is sent as the `myId` parameter. The `t_PartyId` of the party which is to have their participation ended, is sent as the `endPartyId` parameter.

After this request completes successfully, the `endPartyId`'s participation in the session will have ended. The ended party will not be able to make invocations on any of the interfaces belonging to this session. The session will not make any invocations on interfaces of the ended party domain's usage components.

The session may send `endPartyInd()` to the `i_PartyMultipartyInd` interface of the components representing some or all the other parties in the session, (if the parties support the MultipartyInd feature set), and may wait for votes to be received, (if the session supports VotingFS). (Precisely who is sent `endPartyInd()` may be determined by ControlSR feature set, or be service specific, though in general the party to be ended would receive the indication).

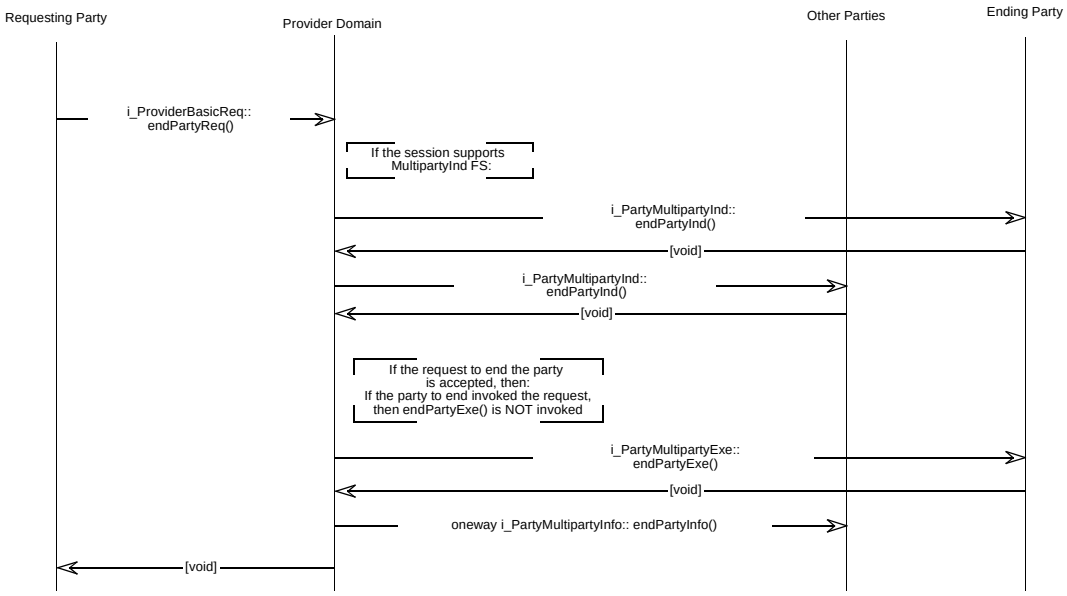


Figure 5-11. End Party event trace.

If session doesn't support VotingFS, or if the request is agreed (see Section 5.5.7 for details), two things happen. Firstly, `endPartyExe()` is sent to the `i_PartyMultipartyExe` interface of the ended party. That party's participation in the session will end. Secondly, `endPartyInfo()` will be sent to the `i_PartyMultipartyInfo` interface of all the components that have registered this interface with the session, (except the requesting and ending parties).

If at this stage the request has been successful, then the `endPartyReq()` will return successfully, and the ended party's participation in the session will have ended. This operation can return success when the action of ending the party's participation cannot be aborted. This effectively means that the request can only return successfully after `endPartyExe()` has returned successfully. It can however return before all the `endPartyInfo()` operations are sent.

Exceptions:

When an exception is raised for whatever reason, `endPartyReq()` has failed and the ended party's participation in the session will not be ended.

See `e_PartyError` exception description for reasons associated with the `endPartyId` for raising exceptions and the error codes to use.

See `e_UsageError` exception description for other reasons for raising exceptions and the error codes to use.

5.5.5.8 suspendMyParticipationReq()

```
void suspendMyParticipationReq (
      in TINACommonTypes::t_ParticipantSecretId myId
) raises (
      TINAUsageCommonTypes::e_UsageError
);
```

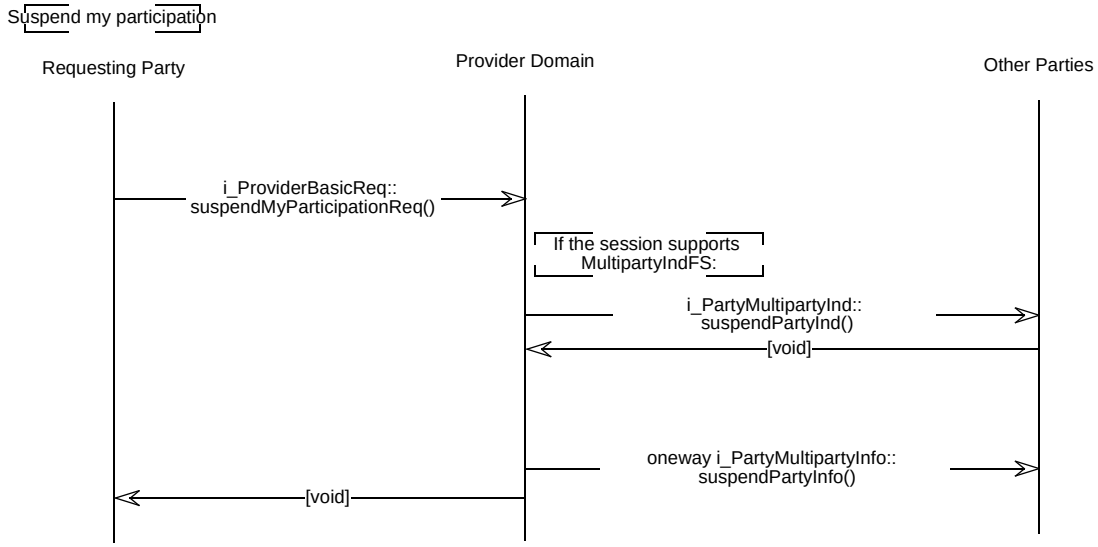


Figure 5-12. Suspend My Participation event trace.

`suspendMyParticipationReq()` allows a participant to request that their participation in the session is suspended. The requesting participant's `t_ParticipantSecretId` is sent as the `myId` parameter.

It follows the same scenario as `endMyParticipationReq()`, except that the requesting party's participation in the session is suspended, not ended, when the request completes successfully.

(Also, `suspendPartyInd()` is sent instead of `endPartyInd()` to the `i_PartyMultipartyInd` interface, and `endSessionInfo()` is sent instead of `endPartyInfo()` to the `i_PartyMultipartyInfo` interface of the other parties in the session.)

5.5.5.9 suspendPartyReq()

```
void suspendPartyReq (
      in TINACommonTypes::t_ParticipantSecretId myId,
      in TINACommonTypes::t_PartyId suspendPartyId
) raises (
      TINAUsageCommonTypes::e_UsageError,
      TINAUsageCommonTypes::e_PartyError
);
```

The `suspendPartyReq()` allows a party to request that another party's participation in the session is suspended. The requesting participant's `t_ParticipantSecretId` is sent as the `myId` parameter. The `t_PartyId` of the party which is to have their participation suspended, is sent as the `suspendPartyId` parameter.

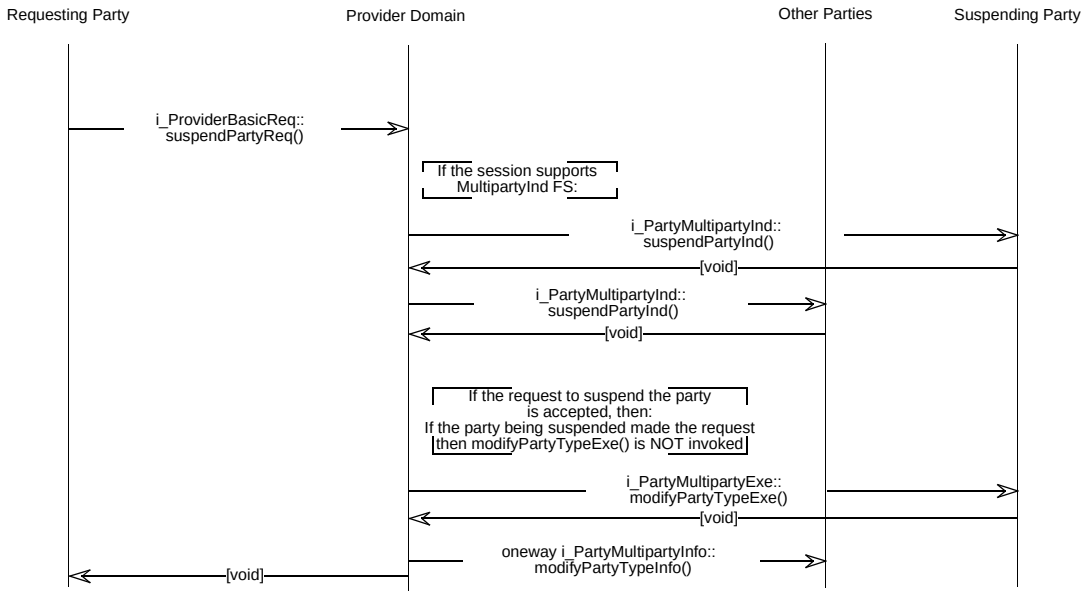


Figure 5-13. Suspend Party event trace.

It follows the same scenario as `endPartyReq`(), except that the requesting party's participation in the session is suspended, not ended, when the request completes successfully.

(Also, `suspendPartyInd()` is sent instead of `endPartyInd()` to the `i_PartyMultipartyInd` interface; `suspendPartyExe()` is sent instead of `endPartyExe()` to the `i_PartyMultipartyExe` interface and `endSessionInfo`() is sent instead of `endPartyInfo`() to the `i_PartyMultipartyInfo` interface of the other parties in the session.)

5.5.5.10 inviteUserReq()

```
void inviteUserReq (
      in TINACommonTypes::t_ParticipantSecretId myId,
      in TINACommonTypes::t_UserDetails invitedUser,
      out TINAUsageCommonTypes::t_InvitationId invitationId,
      out TINACommonTypes::t_InvitationReply reply
) raises (
      TINAUsageCommonTypes::e_UsageError,
      TINACommonTypes::e_UserDetailsError
);
```

The `inviteUserReq()` allows a party to request that another user is invited to join the session. It is used to invite a single specific user to join the session. More information on invitations can be found in Section 3.3.5, "Invitations and Announcements". Information regarding the stability of the invitation specification can also be found in Section 6.1.4, "Invitations". It should not be used to 'announce' the session to many users, (see Section 5.5.5.11 for this). The requesting participant's `t_ParticipantSecretId` is sent as the `myId` parameter.

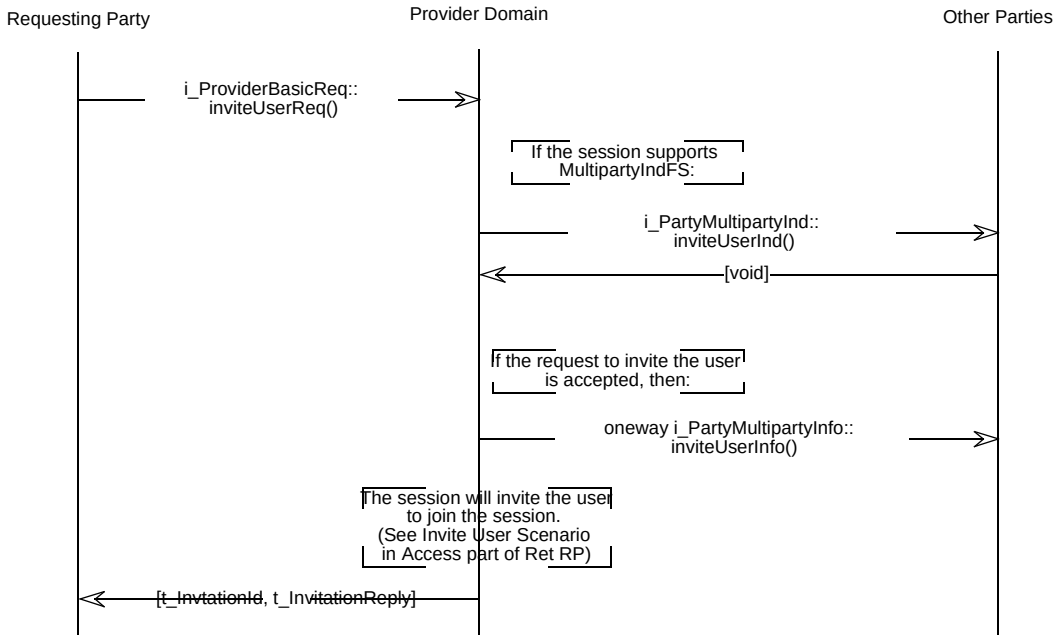


Figure 5-14. Invite User event trace.

The `t_UserDetails` of the user which is to be invited is sent as the `invitedUser` parameter. The `t_UserDetails` is a structure containing the `t_UserId` of the user, and a list of `t_UserProperty`'s. The `t_UserPropertys` may allow the provider domain to locate the user, e.g. by including a reference to the retailer to contact for the user, (although the `t_UserId` should be sufficient to locate the user). No `t_UserProperty` names or values have been defined, and so its use is currently provider specific.

The `t_InvitationId` identifies the invitation request. The party may wish to cancel the request, and can use the `t_InvitationId` to identify the invitation to be cancelled. It can also be used to identify the invitation that an `inviteReplyInfo()` refers to. The `inviteReplyInfo()` is sent when a user replies to an invitation, and the provider domain session wished to inform the parties of the user's intentions. (See Section 3.3.5 for more details on invitations.)

The `t_InvitationReply` is the user's initial reply to the invitation. (See Section 3.3.5 for more details.)

'The session may send `inviteUserInd()` to the `i_PartyMultipartyInd` interface of the components representing some or all the other parties in the session, (if the parties support the MultipartyInd feature set), and may wait for votes to be received, (if the session supports VotingFS). (Precisely who is sent `inviteUserInd()` may be determined by ControlSR feature set, or be service specific).

If session doesn't support VotingFS, or if the request is agreed (see Section 5.5.7 for details), then the invitation will be sent to the user. This process is not described here, nor in this document. It is described in [5], and the part that occurs across the access part of Ret-RP is described in Section 4.4.1.3, "i_ConsumerInvite Interface".

When the invitation has been successfully delivered an `inviteUserInfo`() will be sent to the `i_PartyMultipartyInfo` interface of all the components that have registered this interface with the session, (except the requesting party.)

5.5.5.11 announceSessionReq()

```
void announceSessionReq (
      in TINACommonTypes::t_ParticipantSecretId myId,
      in TINACommonTypes::t_AnnouncementProperties announcement
) raises (
      TINAUsageCommonTypes::e_UsageError,
      TINAUsageCommonTypes::e_AnnouncementError
);
```

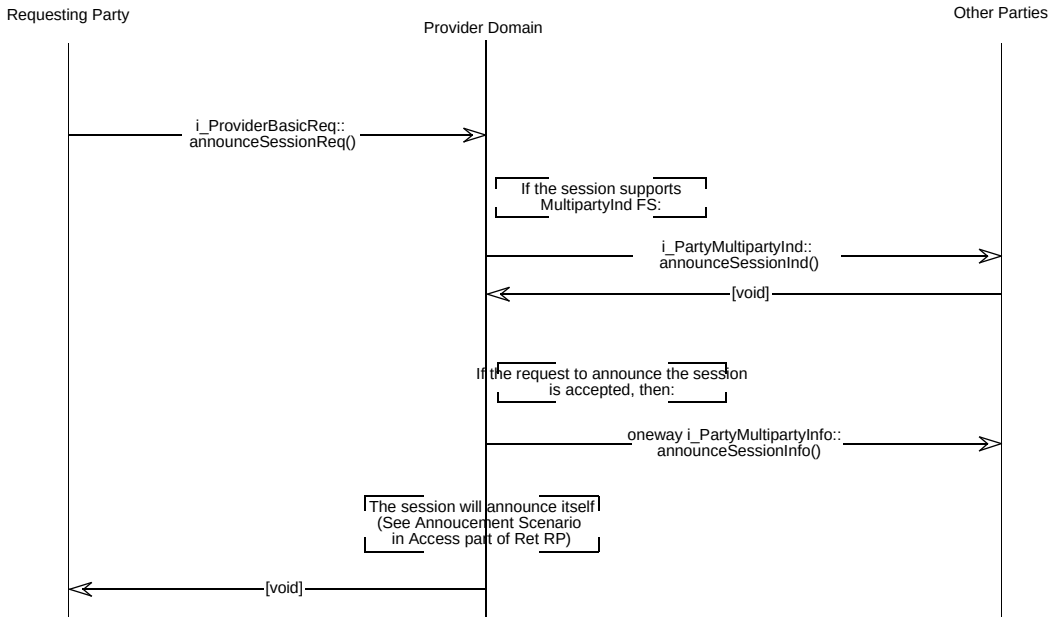


Figure 5-15. Announce Session event trace.

The `announceSessionReq()` allows a party to request that the session be announced. This announcement is intended to reach many users. The complete list of users that will be reached by the announcement may or may not be known by the session. It can however be scoped by the announcement properties. All users that receive the announcement will have the opportunity to attempt to join the session (see Section 4.4.2.4.24, "joinSessionWithAnnouncement()" in the access part of Ret-RP), but may be refused to join at that time. More information on announcements can be found in Section 3.3.5, "Invitations and Announcements". Information regarding the stability of the announcement specification can also be found in Section 6.1.5, "Announcements".

The requesting participant's `t_ParticipantSecretId` is sent as the `myId` parameter.

The `t_AnnouncementProperties` informs the session of the type of announcement to send, and may scope the range of users which the announcement reaches. Currently, no specific announcement property names and values have been defined, and so its use is service and provider specific.

5.5.5.12 i_PartyMultipartyExe interface

This interface is supported by the party domain for sessions that support the Multiparty feature set. The operations on this interface are Exe operations as described in Section 5.2.2, "Types of Operations and Interfaces.". Each operation corresponds to a Req operation described above.

```
    // module TINAPartyMultipartyUsage

interface i_PartyMultipartyExe
    {

    void modifyPartyTypeExe (
          in TINACommonTypes::t_SessionId sessionId,
          in TINAUsageCommonTypes::t_PartyType newType
    ) raises (
          TINAUsageCommonTypes::e_PartyDomainError,
          TINAUsageCommonTypes::e_PartyError
    );

    void endSessionExe (
          in TINACommonTypes::t_SessionId sessionId
    ) raises (
          TINAUsageCommonTypes::e_PartyDomainError
    );

    void endPartyExe (
          in TINACommonTypes::t_SessionId sessionId
    ) raises (
          TINAUsageCommonTypes::e_PartyDomainError
    );

    void suspendSessionExe (
          in TINACommonTypes::t_SessionId sessionId
    ) raises (
          TINAUsageCommonTypes::e_PartyDomainError
    );

    void suspendPartyExe (
          in TINACommonTypes::t_SessionId sessionId
    ) raises (
          TINAUsageCommonTypes::e_PartyDomainError
    );

    };
```

5.5.5.13 i_PartyMultipartyInfo interface

This interface is supported by the party domain for sessions that support the Multiparty feature set. The operations on this interface are Info operations as described in Section 5.2.2, "Types of Operations and Interfaces.". Each operation corresponds to a Req operation described above.

```
    // module TINAPartyMultipartyUsage

interface i_PartyMultipartyInfo
    oneway void modifyPartyTypeInfo (
          in TINACommonTypes::t_SessionId sessionId,
          in TINAUsageCommonTypes::t_PartyDetails partyDetails);

    oneway void endPartyInfo (
          in TINACommonTypes::t_SessionId sessionId,
          in TINACommonTypes::t_PartyId partyId);

    oneway void suspendPartyInfo (
          in TINACommonTypes::t_SessionId sessionId,
          in TINACommonTypes::t_PartyId partyId);

    oneway void resumePartyInfo (
          in TINACommonTypes::t_SessionId sessionId,
          in TINACommonTypes::t_PartyId partyId);

    oneway void joinSessionInfo (
          in TINACommonTypes::t_SessionId sessionId,
          in TINAUsageCommonTypes::t_PartyDetails partyDetails);

    oneway void inviteUserInfo (
          in TINACommonTypes::t_SessionId sessionId,
          in TINACommonTypes::t_UserDetails userDetails,
          in TINAUsageCommonTypes::t_InvitationId invitationId);

    oneway void announceSessionInfo (
          in TINACommonTypes::t_SessionId sessionId,
          in TINACommonTypes::t_AnnouncementProperties announcement);

    oneway void inviteReplyInfo (
          in TINACommonTypes::t_SessionId sessionId,
          in TINAUsageCommonTypes::t_InvitationId invitationId,
          in TINACommonTypes::t_InvitationReply reply);
```

The `inviteReplyInfo()` is different to the other Info operations defined on this interface because it does not correspond to a Req operation. It is sent to all the parties in the session, when a reply to an invitation is received from an invited user. The invited user can reply to an invitation using the `replyToInvitation()` operation on the `i_RetailerNamedAccess` interface, (see Section 4.4.2.3 in the Access part of Ret-RP).

The `replyToInvitation`() can be invoked by the invited user multiple times before joining or declining to join the session. So `inviteReplyInfo`() may be invoked several times on the party domains. The latest invocation gives the users current reply to the invitation. `t_InvitationReply` is described in Section 3.3.5).

5.5.6 Multiparty Ind Feature Set

This feature set is optional. It is supported by the interface `i_PartyMultipartyInd`.

Table 5-14. MultipartyIndFS Interfaces

MultipartyIndFS interfaces on:	
Party domain components	i_PartyMultipartyInd
Provider domain components	(none)

Pre-conditions:

This feature set can only be used if the following pre-conditions are fulfiled:

- The “Multiparty” feature set is used on the same instance of Ret-RP typed interface.

Indications are generated in response to a request operation being received by a session. See Section 5.2.2, "Types of Operations and Interfaces." for details of request, inidcations, execution and info operations.

In general, indications are send out to the ‘owner’ of the party that is the subject of the request. E.g. a session receives a `endPartyReq`(). The operation specifies a party, and an indication would be sent to that party. Ret-RP defines an information model called the Service Session Graph (SSG), which defines ownership roles for the parties, and other resources in a multiparty session. The SSG defines the conditions for sending indications upon receiving a request. In order to fully understand the usage of this feature set the reader is advised to read the “ownership” concept used in the SSG, see Section 5.4.1, "TINA Service Session Model related Information". However, Ret-RP does not mandate that a session implement a model of the SSG in each of the domains. It only mandates that domains that are conformant to this feature set implements the interfaces and behaviour as defined in this feature set.

This feature set can be used in conjunction with the Control Session Relationships Feature Set (ControlSRFS). ControlSRFS allows parties to get and set ownership relationships for parties, (and for other session elements, and other relationships.) This feature set is descibed in Section 5.5.8.

However, this feature set can be used independently from ControlSRFS. Sessions can use default ‘owners’ for parties, or use service-specific interfaces to to modify ‘ownerships’. Session can determine who to send indications to independent of the party affected. Indications merely inform a party domain that a request has been received to perform a session control action, and that the action has not yet been taken. They do not define what action a party domain should take in response to the indication, (although voting, or use of some service-specific operations may be appropriate.)

5.5.6.1 Usage

This feature set is used when “third party confirmations” are required.

5.5.6.2 Components and roles

The `i_PartyMultipartyInd` interface is supported by the party domain and required by the provider domain.

5.5.6.3 IDL Definition and usage scenarios

Note:

- `i_PartyMultipartyInd` re-uses common definitions from Basic and Multiparty Feature Sets
- the scenarios for all the operations on `i_PartyMultipartyInd` are as described in Section 5.5.5, "Multiparty Feature Set".

```
// module TINAPartyMultipartyIndUsage
```

```
interface i_PartyMultipartyInd
    {
    };
```

5.5.6.4 operationCanelled()

```
void operationCancelled (
      in TINACommonTypes::t_SessionId sessionId,
      in TINAUsageCommonTypes::t_IndId indId
) raises (
      TINAUsageCommonTypes::e_PartyDomainError,
      TINAUsageCommonTypes::e_IndError
);
```

5.5.6.5 modifyPartyTypeInd()

```
void modifyPartyTypeInd(
      in TINACommonTypes::t_SessionId sessionId,
      in TINAUsageCommonTypes::t_IndId indId,
      in TINACommonTypes::t_PartyId reqPartyId,
      in TINAUsageCommonTypes::t_PartyDetails partyDetails
) raises (
      TINAUsageCommonTypes::e_PartyDomainError,
      TINAUsageCommonTypes::e_PartyError
);
```

This operation is invoked on all the parties that are defined as Owners of the SSG Party object related to the party whose Type modification is requested, when a `modifyPartyTypeReq`() has been issued by a non-owning party who wants to modify the Type of another party in the service session. If successful, the Type of the Party object related to the modified Party will be modified accordingly.

execution: a `modifyPartyTypeExe()` is invoked on the party who is to be modified.

information: a `modifyPartyTypeInfo()` is invoked on all parties that have at least ReadPermission over the SSG Party object related to the party whose modification has been performed.

5.5.6.6 endSessionInd()

```
void endSessionInd (
      in TINACommonTypes::t_SessionId sessionId,
      in TINAUsageCommonTypes::t_IndId indId,
      in TINACommonTypes::t_PartyId reqPartyId
```

```
) raises (
      TINAUsageCommonTypes::e_PartyDomainError,
      TINAUsageCommonTypes::e_PartyError
);
```

This operation is invoked on all the parties that are defined as Owners of the SSG Session object, if an `endSessionReq()` has been issued by a non-owning party who wants to end an existing service session. If successful, the session will end.

execution: an `endSessionExe()` is invoked on all the parties that are participating in the service session.

information: empty

5.5.6.7 endPartyInd()

```
void endPartyInd (
      in TINACommonTypes::t_SessionId sessionId,
      in TINAUsageCommonTypes::t_IndId indId,
      in TINACommonTypes::t_PartyId reqPartyId,
      in TINACommonTypes::t_PartyId partyId
) raises (
      TINAUsageCommonTypes::e_PartyDomainError,
      TINAUsageCommonTypes::e_PartyError
);
```

This operation is invoked on all the parties that are defined as Owners of the SSG Party object related to the party whose deletion is requested, when an `endPartyReq()` has been issued by a non-owning party who wants to delete another party from the service session. If successful, the Party object related to the ended Party will be removed from the SSG.

execution: a `endSessionExe()` is invoked on the party who is to be deleted.

information: an `endPartyInfo()` is invoked on all parties that have at least ReadPermission over the SSG Party object related to the party whose deletion has been performed.

5.5.6.8 suspendSessionInd()

```
void suspendSessionInd (
      in TINACommonTypes::t_SessionId sessionId,
      in TINAUsageCommonTypes::t_IndId indId,
      in TINACommonTypes::t_PartyId reqPartyId
) raises (
      TINAUsageCommonTypes::e_PartyDomainError,
      TINAUsageCommonTypes::e_PartyError
);
```

This operation is invoked on all the parties that are defined as Owners of the SSG Session object, if a suspendSessionReq has been issued by a non-owning party who wants to suspend an existing service session. If successful, the related USM and SSM service sessions will be suspended.

execution: a `suspendSessionExe()` is invoked on all the parties that are involved in the service session.

information: empty

5.5.6.9 resumeSessionInd()

```
void resumeSessionInd(
      in TINACommonTypes::t_SessionId sessionId,
      in TINAUsageCommonTypes::t_IndId indId,
      in TINACommonTypes::t_PartyId reqPartyId
) raises (
      TINAUsageCommonTypes::e_PartyDomainError,
      TINAUsageCommonTypes::e_PartyError
);
```

This operation is invoked on all the parties that are defined as Owners of the SSG Session object, if a request to resume the suspended session has been issued by a non-owning party via its Access Session. The resume request is issued via its access session and then forwarded to the SSM/GSC. If successful, the related USM and SSM service sessions will be resumed.

execution: a `resumeSessionExe()` is invoked on all the parties that are involved in the service session.

information: empty

5.5.6.10 suspendPartyInd()

```
void suspendPartyInd (
      in TINACommonTypes::t_SessionId sessionId,
      in TINAUsageCommonTypes::t_IndId indId,
      in TINACommonTypes::t_PartyId reqPartyId,
      in TINACommonTypes::t_PartyId partyId
) raises (
      TINAUsageCommonTypes::e_PartyDomainError,
      TINAUsageCommonTypes::e_PartyError
);
```

This operation is invoked on all the parties that are defined as Owners of the SSG Party object related to the party whose suspension is requested, when a suspendPartyReq has been issued by a non-owning party who wants to suspend another party in the service session.

execution: a `suspendPartyExe()` is invoked on the party who is to be suspended.

information: a `suspendPartyInfo()` is invoked on all parties that have at least ReadPermission over the SSG Party object related to the party whose suspension has been performed.

5.5.6.11 resumePartyInd()

```
void resumePartyInd (
      in TINACommonTypes::t_SessionId sessionId,
      in TINAUsageCommonTypes::t_IndId indId,
      in TINACommonTypes::t_PartyId partyId
) raises (
      TINAUsageCommonTypes::e_PartyDomainError,
      TINAUsageCommonTypes::e_PartyError
);
```

This operation is invoked on all the parties that are defined as Owners of the SSG Party object related to the party whose resumption is requested, when the suspended party requests a resume via its Access Session.

execution: empty

information: a resumePartyInfo is invoked on all parties that have at least ReadPermission over the SSG Party object related to the party whose resumption has been performed.

5.5.6.12 joinSessionInd()

```
void joinSessionInd (
      in TINACommonTypes::t_SessionId sessionId,
      in TINAUsageCommonTypes::t_IndId indId,
      in TINACommonTypes::t_UserDetails userDetails
) raises (
      TINAUsageCommonTypes::e_PartyDomainError,
      TINACommonTypes::e_UserDetailsError
);
```

This operation is invoked on all the parties that are defined as Owners of the SSG Session object when a `joinSessionReq()` has been issued by a user who wants to join an existing service session. If successfull, a new Party object is added to the SSG to model this user (who becomes a party) in the service session.

execution: empty

information: a `joinSessionInfo()` is invoked on all parties that have at least ReadPermission over the SSG Session object.

5.5.6.13 inviteUserInd()

```
void inviteUserInd (
      in TINACommonTypes::t_SessionId sessionId,
      in TINAUsageCommonTypes::t_IndId indId,
      in TINACommonTypes::t_PartyId reqPartyId,
      in TINACommonTypes::t_UserDetails userDetails
) raises (
      TINAUsageCommonTypes::e_PartyDomainError,
      TINAUsageCommonTypes::e_PartyError,
      TINACommonTypes::e_UserDetailsError
);
```

This operation is invoked on all parties that are defined as Owners of the SSG Session object when an `inviteUserReq()` has been issued by a non-owning party who wants to invite another user to join an existing service session. If successfull, a new Party object is added to the SSG to model this new party in the service session.

information: an `inviteUserInfo()` is invoked on all parties that have at least ReadPermission over the SSG Session object.

5.5.6.14 announceSessionInd()

```
void announceSessionInd (
      in TINACommonTypes::t_SessionId sessionId,
```

```
        in TINAUsageCommonTypes::t_IndId indId,
        in TINACommonTypes::t_PartyId reqPartyId, // Requesting Party
        in TINACommonTypes::t_AnnouncementProperties announcement
    ) raises (
        TINAUsageCommonTypes::e_PartyDomainError,
        TINAUsageCommonTypes::e_PartyError,
        TINAUsageCommonTypes::e_AnnouncementError
    );
```

This operation is invoked on all parties that are defined as Owners of the SSG Session object when an **`announceSessionInd`**`()` has been issued by a non-owning party who wants to announce the session, to allow users to join the session.

5.5.7 Voting Feature Set

VotingFS allows the parties to vote on whether an action should occur. (The party domain finds out that the action is going to occur by receiving an indication through the i_PartyMultipartyInd interface of the MultipartyInd feature set, or indiction interfaces of other feature sets.

VotingFS is an optional feature set, which requires that the session also support MultipartyInd feature set.

Table 5-15. VotingFS Interfaces

VotingFS interfaces on:	
Party domain components	i_PartyVotingInfo
Provider domain components	i_ProviderVotingReq

5.5.7.1 i_ProviderVotingReq Interface

This interface allows the party domain to register a vote, in response to an indication. It is assumed that any party that receives an indication can send a vote, if the session supports the VotingFS. (No other mechanism is defined in Ret-RP for asking parties to vote, or for asking them not to vote when receiving an indication. It is supported on the provider domain components.

```
// module TINAProviderVotingUsage

interface i_ProviderVotingReq
{
      void voteReq(
               in TINACommonTypes::t_ParticipantSecretId myId,
               in TINAUsageCommonTypes::t_IndId indId,
               in t_Vote vote
      ) raises (
               TINAUsageCommonTypes::e_UsageError,
               TINAUsageCommonTypes::e_IndError,
               e_VoteError
      );
};
```

`voteReq()` allows a participant to vote after an indication has been received. All indications carry an indication identifier `t_IndId`. This allows a part to send a response to receiving the indication. The `voteReq()` operation is a response to an indication that allows the party domain to vote on whether they think the action indicated should occur.

`indId` identifies the indiction to which this is a response.

`vote` is a `t_Vote` structure containing `response`, and `value`. The `response` is an enumerated type, with the values: `NoVote` (the party is not registering a vote when sending this operation), `Agree` (the party agrees to allow the action to occur), `Disagree` (the party disagrees with the action occuring), `Abstain` (the party abstains from voting). The `value` is a short integer which can be used in any service-specific way desired. It is provided to allow the session to support a ‘grey-scale’ of voting. The `value` may be used in addition to the response to indicate the ‘level’ of agreement or

disagreement, or may be used with `NoVote` to indicate that response is to be ignored, and only `value` to be used. The exact semantics for the `value` are service-specific, and it can be ignored if required.

5.5.7.2 i_PartyVotingInfo Interface

This interface allows the provider domain to inform the parties that voted of the result of the voting. It is supported on the party domain.

```
// module TINAPartyVotingUsage

interface i_PartyVotingInfo
{
oneway voteInfo(
               in TINACommonTypes::t_SessionId sessionId,
               in TINAUsageCommonTypes::t_IndId indId,
               in t_VoteResult result
     );
};
```

`voteInfo()` allows the provider domain to inform the parties that send `voteReq`() in response to an indication, the result of the voting. It is a oneway operation. It is sent to all the parties that received the original indication, whether they sent a `voteReq`() or not.

`indId` identifies the indiction, and so the `voteReq`() response.

`result` is the result of the voting. It is an enumerated type, with the following values:

- `UnknownVoteResult`: The result of the voting is unknown. The voting has finished. This operation does not inform the party as to the outcome of the voting. (i.e. if the action which generated the indication will occur or not.). However the voting has finished so there's no point in tring to send a `voteReq`() to register your vote. (It may be used just to stop more voting, and another `voteInfo`() could be sent subsequently, or the party may just have to wait to receive the Info operation generate by the Req().)
- `VoteAgreed`: The vote has agree that the action will take place, (expect Info or Exe operations soon.)
- `VoteNotAgreed`: The vote has not agreed to the action taking place. The action will no occur, and the Req that generated the indications will raise an `e_UsageError` exception, with an error code of `UsageNotAccepted`.

5.5.8 Control Session Relationship feature set

This section will be divided in two subsections: the first one will describe the information modelling concepts which are necessary to fully understand the Control Session Relationship, the second one will describe the Control Session Relationship Feature Set itself.

5.5.8.1 Control Session Relationship information model

The Control Session Relationship feature set (“ControlSR”) expresses the session relationship between a Party -the Controller- and a controlled SSG Object. The complete description of the session relationship concept can be found in [5].

The possible controlled Session Graph objects are Session, Party and StreamBinding. The possible levels of control are NoControl (lowest), ReadPermission, WritePermission and Ownership (highest). These levels of control express:

- the capability of a Party to control changes to the Session Graph and potentially deny them (i.e. Ownership)
- the capability of a Party to introduce changes to the Session Graph (i.e. WritePermission)
- the visibility of Session Graph Objects as perceived by a Party (i.e. ReadPermission).

A higher level of control always includes the capabilities implied by a lower level of control (e.g. Ownership implicitly implies ReadPermission). Going from a lower level to a higher level is referred to as upgrading, the reverse is referred to as downgrading.

Each object that can be controlled is attributed by a defaultControl setting. In addition, the StreamBinding information object is also attributed by a minimumParticipantControl setting. These attributes will be used to determine the level of control between a Party and the controlled information object for those parties that haven’t explicitly declared a session relationship between themselves and the information object.

This is explained in more detail below. The minimumParticipantControl setting of a StreamBinding has to be at least as high as its defaultControl setting and must be at least ReadPermission. The default control associated with an object has to be set at creation time by the creating party, and cannot be changed later.

5.5.8.1.1. ControlSR expressed on the Ret-RP interfaces

It is important to distinguish between the actual realisation of the Session Graph as a graph consisting of different information objects: as managed within the domains at each side of the Ret-RP; and the perception at Ret-RP interface level of information objects which are identified by means of object identifiers. In order to simplify the operations on Ret-RP, to increase the clarity and consistency of the usage of ControlSR, and to suppress the possibility to create Session Graph objects in the session for which no default control would be defined, no explicit Session Graph object is introduced at Ret-RP interface level to describe a ControlSR. Instead, the t_ControlInfo and t_ControlDescription IDL structure types have been introduced. The t_ControlInfo struct identifies both the controller Party and controlled Object and it identifies the control level by means of a t_ControlDescription value. The t_ControlDescription contains a boolean useDefaultControl field. If the useDefaultControl field of the t_ControlDescription equals to FALSE, the ControlSR is assumed to exist and its control level is expressed by the controlType field of t_ControlDescription. The controlType field equals to one of the values: NoControl, ReadPermission, WritePermission or Ownership. If the useDefaultControl equals to TRUE, this is equivalent to the non-existence of a session relationship between the controller and the controlledObject (instead the default control of the controlled Object will be assumed).

The creator of a Session, Party or StreamBinding object must indicate within the same IDL invocation which results in the creation of the Session Graph object:

- the kind of control he would like to have over that object when it is created: this value must be at least ReadPermission
- the defaultControl of the object (+ the minimumParticipantControl in the case of the StreamBinding)

5.5.8.1.2. How to determine the Control of a Party over a Session Graph Object

If there exists a session relationship between the controller Party and the controlled Session Graph Object, the level of control exercised by the controller Party is expressed by that session relationship control level. If there is no such session relationship defined, we need to distinguish between the different types of controlled objects:

Controlled Session Graph Object = Session Object

The level of control is equal to the defaultControl setting of the Session Object.

Controlled Session Graph Object = StreamBinding Object

Again two cases need to be distinguished:

- the Party is not a Participant in the StreamBinding: If there exists a session relationship between the controller Party and the Session Object, the level of control is expressed by that session relationship control level. If there is no such session relationship, the level of control is equal to the defaultControl setting of the controlled StreamBinding Object
- the Party is a Participant in the StreamBinding: the control level is equal to the maximum of (the minimumParticipantLevel of the controlled StreamBinding, the control level determined as if the Party would not be a Participant in the StreamBinding).

Controlled Session Graph Object = Party Object

If there exists a session relationship between the controller Party and the Session Object, the level of control is expressed by that session relationship control level. If there is no such session relationship, the level of control is equal to the defaultControl setting of the controlled Party Object.

In addition, one more rule needs to be observed in the case the controller and the controlled Party are one and the same: the control level can never be any lower than ReadPermission. This means, if the method explained above leads to a NoControl result, this is automatically upgraded to ReadPermission.

If the level of control between a controller Party and a controlled Object is determined to be equal to Ownership we refer to the controller Party as an Owner of the controlled Object. It is important to determine within the Provider domain the complete list of owners of an Session Graph object when an operation is applied to it, because these owners will be invoked to authorize the operation. To achieve this, the Provider domain invokes the appropriate operation on the `i_PartyMultipartyInd` interface of the party domain, and the party answers by invoking the appropriate operation on the `i_ProviderVotingReq` interface of the Provider domain. See also Section 5.2.2, "Types of Operations and Interfaces."; Section 5.5.6, "Multiparty Ind Feature Set", and Section 5.5.7, "Voting Feature Set".

5.5.8.1.3. The Semantics of the Different Levels of Control

This part elaborates the impact of the control settings on the processing of the different Session Graph modification operations.

First of all, we need to distinguish between general “Add” and “Delete” type of operations.

“Add” operations include:

- addition of a Party to a Session
- addition of a StreamBinding to a Session

“Delete” operations include:

- delete a Session
- delete a Party from a Session
- delete a StreamBinding from a Session

In addition to these operations, we also need to consider the addition/deletion of Participants to a StreamBinding (a Participant is a Party which Participates to a StreamBinding). To determine the rules applicable to addition/deletion of Participants, the following principles are applied to the general add/ delete mechanisms:

- semantically the control level of a controller Party over a Participant is equal to the control level of that controller Party over the StreamBinding to which the Participant belongs
- the relation of a Participant to the StreamBinding is considered to be analogous to the relation of a Party to the Session.

The control level of a Party over an Session Graph Object can be determined at any time, using the relevant defaultControl settings. This can be interpreted as if the session relationship always exists, either explicitly or implicitly. Therefore, we talk about modification to a session relationship, rather than addition or deletion. To determine the rules applicable to the modification of session relationships the following principles are applied to the general add/delete mechanisms:

- semantically the control level of a Party over a session relationship is equal to ReadPermission, WritePermission or Ownership if that Party has ReadPermission, WritePermission or Ownership over both the ControlSR-controlled object and the ControlSR-controller Party

5.5.8.1.4. Ownership

The Ownerships are one of the means to determine which Parties have to acknowledge the processing of certain operations on the SSG

The following rules apply to the processing of the general add/delete operations:

- Add Operations: If the operation is initiated by an Owner of the Session, no additional Parties need to acknowledge the operation on the basis of session relationships. otherwise, the operation needs to be acknowledged by each of the Parties whose control level on the object that is to be created is determined to be equal to Ownership. This level of control needs to be determined as if the object that is to be created has already been added to the SSG : i.e. if there exists a session relationship between the Party and the Session Object, the level of control is equal to that session relationship control level; otherwise, the level of control is equal to the defaultControl setting of the Object that is to be created, as expressed in the request operation.

- Delete Operations: If the operation is initiated by an Owner of the object that is to be deleted, no additional Parties need to acknowledge the operation. otherwise, all Owners of the object that is to be deleted need to acknowledge the operation before its processing can proceed.

The following rules apply to the addition/deletion of Participants to/from a StreamBinding:

If the operation is initiated by an Owner of the StreamBinding, no additional Parties need to acknowledge the operation on the basis of session relationships otherwise, the operation needs to be acknowledged by all Owners of the StreamBinding.

The following rules apply to the modification of session relationships:

If the operation is initiated by an Owner of the session relationship, no additional Parties need to acknowledge the operation on the basis of session relationships. Otherwise, the operation needs to be acknowledged by all Owners of the session relationship.

5.5.8.1.5. WritePermission

The WritePermissions determine which Parties have the possibility to request modifications on the SSG. If the Party only has ReadPermission the request is automatically refused. If the Party has WritePermission the request involves confirmation by the Owners of the concerned SSG object, and is accepted or refused according to the Owners voting. If the Party is an owner then the request is accepted and does not involve any voting by other owners.

Rules and applicability are already described in the paragraph above on Ownership.

5.5.8.1.6. ReadPermission

The ReadPermissions determine the visibility of Objects by the Parties. Visibility is defined for the following Objects (Notice that Participant and session relationships are considered as an Object in this context):

A Party or StreamBinding is only visible to -controller- Parties which have -at least- ReadPermission control level on the -controlled- Party/StreamBinding.

A Participant in a StreamBinding is only visible to -controller- Parties which have -at least- ReadPermission control level on both the StreamBinding and the Party (i.e. the Party object behind the Participant).

A session relationship is only visible to -controller- Parties which have -at least- ReadPermission control level on both the ControlSR controller Party and the ControlSR controlled Object.

Note: Sessions are assumed to be visible to all of its Parties. The difference between having NoControl or ReadPermission on the Session only affects the determination of the control-level a Party has on either a Party or a StreamBinding when there exists no explicit session relationship between that Party and the controlled Party or StreamBinding.

Visibility is checked when:

- a party explicitly retrieves SSG information
- a party is added or deleted, (including a User/Participant, StreamBinding, Participant, session relationship):

Parties (i.e. the end-user associated with the Party) to which the object is visible are automatically informed. To determine the visibility of the added/deleted object towards a Party, control levels have to be determined as explained above.

Note for the add operations: the defaultControl as specified in the operation which resulted in the addition may affect the visibility of the added object.

Note for the delete operations: of course, the visibility of the deleted object is determined as if the object hasn't yet been removed from the SSG (i.e. all relevant session relationships and defaultControl settings still apply)

A Party will never receive any information about Objects which are not visible to that Party.

5.5.8.2 Control Session Relationship feature set:

This feature set is optional. .

It supports the party domain components making requests to modify the session relationship between a Party (the Controller) and a controlled SSG Object. The complete description of the session relationship concept can be found in Section 5.5.8.1, "Control Session Relationship information model".

Table 5-16. ControlSRFS Interfaces

ControlSRFS interfaces on:	
Party domain components	i_PartyControlSRInd i_PartyControlSRInfo
Provider domain components	i_ProviderControlSRReq

Pre-conditions:

This feature set can only be used if the following pre-conditions are fulfiled:

- The "Multiparty" feature set is used on the same instance of Ret-RP typed interface.

Usage:

This feature set is used when Session Relationships need to be modified (as described in the previous section Session Relationships are by default always supposed to exist).

Components and roles:

The `i_ProviderControlSRReq` interface is supported by the provider domain and required by the party domain.

The `i_PartyControlSRInd` and `i_PartyControlSRInfo` interfaces are supported by the party domain and required by the provider domain.

Specific types for the Control Session Relationship Feature Set:

```
enum t_ControlType {
      NoControl,
      ReadPermission,
```

```
      WritePermission,
      Ownership
};

struct t_ControlDescription {
      boolean useDefaultControl;
      t_ControlType controlType;
                // only relevant if useDefaultControl==FALSE
};

struct t_ControlInfo {
      t_ControlDescription controlDescription;
      TINAUsageCommonTypes::t_PartyDetails controllerInfo;
      TINACommonTypes::t_ElementId controlledObject;
};

typedef sequence<t_ControlInfo> t_ControlInfoList;

struct t_DefaultControlInfo {
      t_ControlType controlType;
      TINACommonTypes::t_ElementId controlledObject;
};

typedef sequence<t_DefaultControlInfo> t_DefaultControlInfoList;
```

5.5.8.2.1. IDL Definition and usage scenarios

Note: the scenarios only shows messages on the usage part of Ret-RP.

Request Phase:

The `setControlReq` operation is invoked by a participant who wants to modify a session relationship between a Party -the controller- and a controlled object - SSG Session, Party or StreamBinding object.

```
void setControlReq(
      in TINACommonTypes::t_ParticipantSecretId reqPartySecretId,
      in TINACommonTypes::t_PartyId controllerPartyId,
      in TINACommonTypes::t_ElementId controlledId,
      in TINAControlSRTypes::t_ControlDescription control
) raises (
      TINAUsageCommonTypes::e_UsageError,
      TINAUsageCommonTypes::e_PartyError
);
```

Indication Phase:

The setControlInd operation is invoked on:

- all parties implied by the “modify session relationship” Ownership semantics

- if the controllerPartyId is different from the requester, the party related to that controllerPartyId.

```
void setControlInd (
      in TINACommonTypes::t_SessionId sessionId,
      in TINAUsageCommonTypes::t_IndId indId,
      in TINAControlSRTypes::t_ControlInfo controlInfo
) raises (
      TINAUsageCommonTypes::e_PartyDomainError,
      TINAUsageCommonTypes::e_PartyError
);
```

Execution Phase: empty.

Information Phase:

The setControlInfo operation is invoked on:

- all parties implied by the “session relationship” ReadPermission semantics

```
oneway void setControlInfo (
      in TINACommonTypes::t_SessionId sessionId,
      in TINAControlSRTypes::t_ControlInfo controlInfo
);
```

The scenario is as follows::

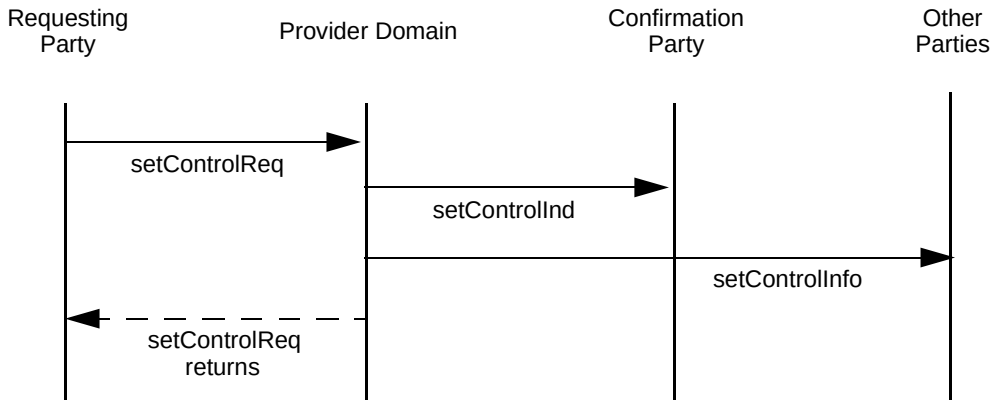


Figure 5-16. ControlFS Scenario

5.5.9 Participant Oriented Stream Binding Feature Set

The Participant Oriented Stream Binding feature set (PaSBFS) is based on the high level stream binding model presented in Section 5.4.1.3, which supports multiparty-to-multiparty (i.e. multipoint-to-multipoint), multimedia connections. Section 5.4.2 introduced the terminology and parameters which will be used to describe this feature set. In summary, stream bindings are described in terms of participants, type, and associated media type descriptions. Media types map to the SFCs implicitly defined by the stream binding description. Though media type requirements are specified for the overall stream binding, they may be modified for particular participants.

Stream binding operation requests are specified in terms of this model, i.e. participants, stream binding types, and media types. A session member need not know details of other session members to request the creation of a stream binding or the addition of subsequent participants. Instead, after a request has been made, the provider makes exe requests to the nominated session members, which respond with the necessary binding information in terms of SFEPs. Modification, activation, deactivation, and deletion request operations are also specified in these terms.

Any party associated with a stream binding is called a Stream Binding (SB) member. Parties which submit SFEPs for binding are termed SB participants. A party can participate in a number of stream bindings. Not all SB members are SB participants, they may have control relationships instead, e.g. the party that initiates a stream binding need not be bound. Such parties are termed SB controllers. Any SB member that makes a request to initiate, modify, or delete a stream binding (or any part of one) is termed a requester.

The PaSB feature set is not sufficient to support stream binding alone: it only allows the initiation and configuration of stream bindings. It needs to be supported by lower level communication functionality. A TINA communication session, or equivalent functionality, is needed to set up a stream binding. A stream binding needs to be mapped to the communication session (or equivalent's) information model. An algorithm is required for such a mapping. Section 5.4.2.2 details the basis of such an algorithm and how it relates to the TINA communication session model.

5.5.9.1 Interfaces

The following interfaces form the Participant SB feature set.

Table 5-17. Participant SB Feature Set Interfaces

Participant SB interfaces on:	
Party domain components	i_PartyPaSBExe i_PartyPaSBInfo
Provider domain components	i_ProviderPaSBReq

i_ProviderPaSBReq: Allows a party to request the establishment, modification, and termination of a stream binding. Stream bindings are specified in terms of participants, type and quality of service. SFCs are implicit to the stream binding. Participants may always include the requester.

- Add stream binding: create a stream binding, nominating the type and initial participants.
- Add participants to a stream binding.
- Delete participants from a stream binding.
- Delete a stream binding.

- Activate participants within a stream binding or an entire stream binding.
- Deactivate participants within a stream binding or an entire stream binding.
- Modify the stream binding: change media type requirements for the entire stream binding or for given participants.
- List stream bindings: return list of stream binding identifiers for the session.
- Get stream binding info: request detailed information on a given stream binding.

i_PartyPaSBExe: Allows a provider session to request the establishment, modification, and termination of participation in a stream binding. The participant is given type and media type requirements of the stream bindings, and information specific to their participation. When a join or modify request is made, the participant needs to returns SFEPs that support the requested media types[1] and implied SFCs.

- Join a stream binding: Sent to participants either when the stream binding being setup or after an add participant request is made.
- Leave a stream binding: Sent to participants either when a stream binding is being deleted or to specified participants after a delete participant request is made.
- Modify participation in a stream binding: Modify the participants required media types.

i_PartyPaSBInfo: notifies a requester or participant in a stream binding of the changes in the stream binding's state, or failure or successful completion of stream binding requests.

- Confirm the successful completion of a stream binding request and gives the new state of the stream binding (i.e. which participants bound for which SFEPs).
- Notify the failure of a stream binding request and give the reason for the failure in terms of failed elements (i.e. participants or SFEPs that were not bound).
- Distribute SI and SFEP information between SB Members.
- Notify the withdrawal of stream binding elements (e.g. SIs, SFEPs, SB members).
- Notify change of status of the stream binding, results of operations, etc.
- Update or cancel notifications: indicates a change in a previous notification's status.

5.5.9.2 Asynchronous and synchronous responses

The participant oriented stream binding feature set supports asynchronous as well as synchronous interactions, as it may not always be possible to support communication requests in a synchronous manner. There are two ways of initiating an asynchronous response:

- Requester driven: A requester may explicitly ask for asynchronous set up. This is done by means of a "wait" flag in the operation's parameter list to false.
- Provider driven: The provider initiates an asynchronous response.

However it is initiated, asynchronous requests are supported by the provider making an `e_NoSynchronousReqResp` exception after it receives the request, which gives the requester a request identifier. When the operation is successfully completed, a confirm operation is made on the requester's `i_PartyPaSBInfo` interface. If there is an error or the communications fail, a failure notification operation is made to the requester. These operations return the new state of the stream

1. It may not always be possible to meet all requirements. In this case, the SFEPs returned should indicate which requirements can be met. If this is not suitable, the provider logic may not proceed with binding that participant.

binding. Figure 5-17 shows an example asynchronous response. Note that to complete a request, operations on a number of other interfaces may be required. All these operations are completed before the provider sends a confirm to the requester.

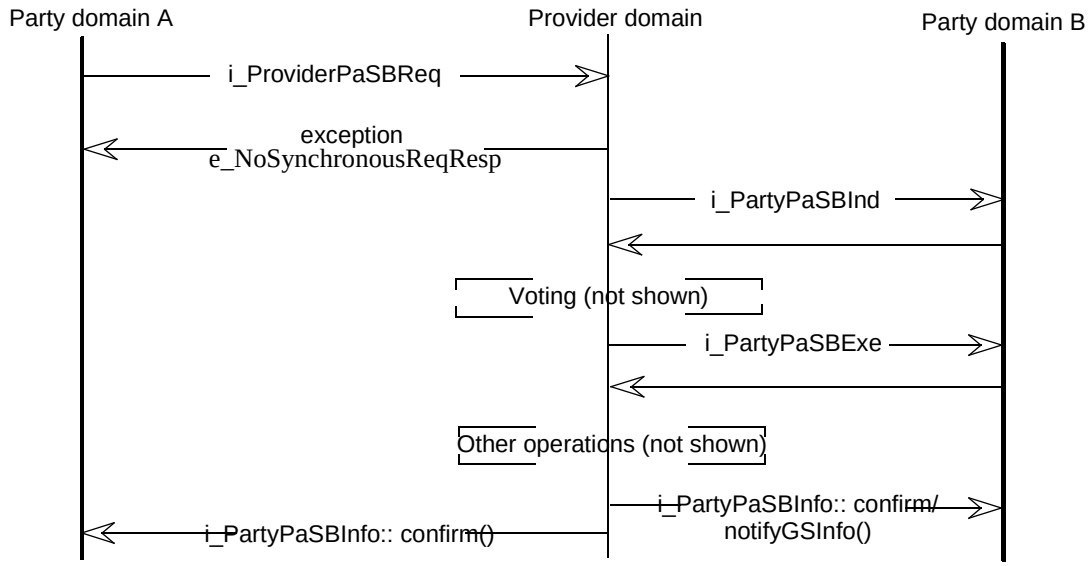


Figure 5-17. Example asynchronous response event trace

For a synchronous response, the “wait” flag must be set true. Then, if the provider supports synchronous responses, the request will not return until it has completed successfully. Once a request is completed, the new state of the stream binding is returned to the requester. If there is an error or the request fails, then an exception will be thrown. Figure 5-18 shows an example successful synchronous request event trace. As before, many operations may be required to support the request, all of which must be completed before the original request returns.

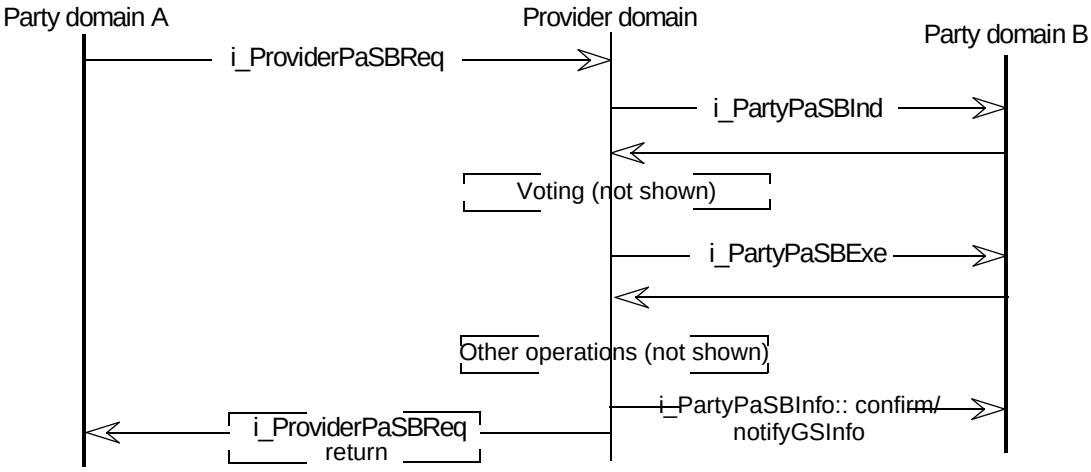


Figure 5-18. Example synchronous response event trace

Other participants (both SB Members and controllers) may also be returned the new state of the stream binding using confirm or notify operations on their `i_PartyPaSBInfo` interfaces. Both figures indicate these operations.

5.5.9.3 Indications and voting

If the PaSBFS is used in conjunction with the PaSBInd feature set, indications of various stream binding requests may be issued to SB participants. Indications are issued before any exe operations, see Figures 5-17 and 5-18. Which parties receive indications may be governed by control relations (see the TINA ControlSRFS) or be service session specific.

If the TINA Voting feature set (or similar) is supported, parties may vote on whether an operation can proceed. The operation will not proceed unless the vote is successful according to the agreed voting method. The PaSBInd feature set describes indication operations for stream bindings. Also see the MultiPartyInd feature set and the Voting feature set.

5.5.9.4 Scenario

The following sections provide event traces which describe how the PaSB operations and interfaces relations and use. This section gives a overview of the underlying scenario, see Figure 5-19. Party A and Party B are existing parties ina session established between their respective party domains and Provider R. Once the stream binding is created, both A and B become SB participants. Provider R acts as the stream binding provider. Party A initiates the stream binding requests, i.e. acts as the requester. For multiple parties, actions on B's interfaces would be repeated for other participants.

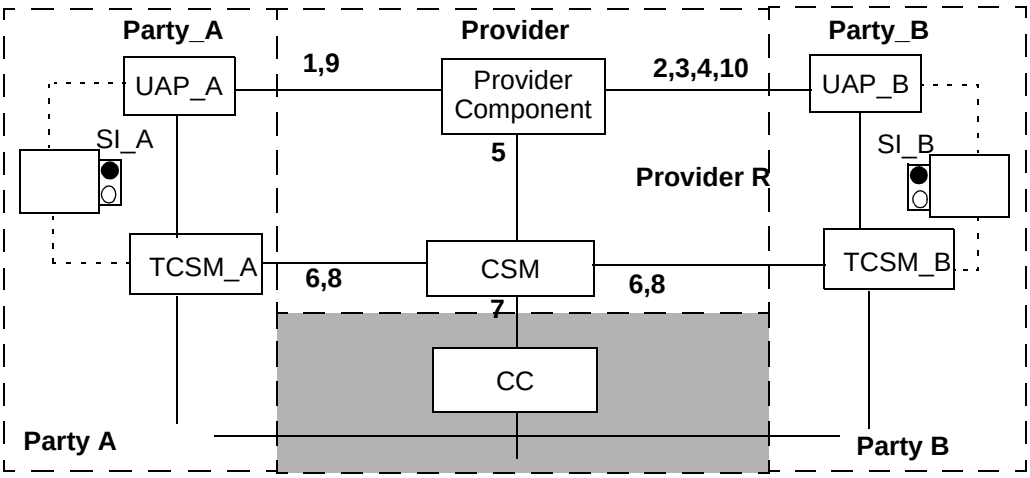


Figure 5-19. Add stream binding scenario

Each party has an associated User Application (UAP) that supports the party PaSB interfaces and has one or more associated SIs or groups of SFEPs. These SFEPs either terminate connections to the application (like sockets) or devices under the applications control, such as speakers, microphones, cameras etc. These SFEPs must be known by the Terminal Communication Session Manger (TCSM) before they can be used. The creation of SFEPs and their registration with the TCSM is the responsibility of the terminal software and is not specified by this reference point. The provider components are in the Provider's domain and support the PaSB provider interfaces.

This scenario assumes the existence of a separate communication session with underlying connectivity components. This is shown by the TCSM and Communication Session Manager (CSM) components which support communication functionality in the party and provider domains respectively, and the Connection Coordinator (CC) which supports network connectivity. This is not mandatory and the components could be subsumed into the overall UAPs and service session components or non-TINA compliant functionality could be supported.

They are included to show the relationship between service session and communication functions. Section 5.6.2. provides details of TINA communication session operations. Communication session level operations are initiated as the result of service level interactions on the stream binding. The TINA communication session functionality, or an equivalent, is needed to support the PaSB feature set.

5.5.9.5 i_ProviderPaSBReq Interface

Operations will be discussed in the context of scenarios. These scenarios may involve operations on other interfaces (including interfaces of other feature sets). When this occurs, this section will briefly describe why the operation is required. All request operations identify the requester by a `t_ParticipantSecretId` type `myId` parameter, see Section 3.3.4.2. Once a stream binding is created, it is identified by a t_SBId type sbId parameter, see Section 5.4.2, for all subsequent operations.

```
// module TINAProviderPaSBUsage

interface i_ProviderPaSBReq
{
};
```

5.5.9.5.1. Add stream binding request

```
void addProviderPaSBReq(
      in TINACommonTypes::t_ParticipantSecretId myId,
      in TINAStreamCommonTypes::t_SBType reqType,
      in TINASBComSCommonTypes::t_MediaDescList media,
      in TINAPaSBTypes::t_ParticipantDescList reqMembers,
      in TINAStreamCommonTypes::t_SFEPServDescList requesterSIs,
      in TINAStreamCommonTypes::t_SBSuccessCriteria criteria,
      in TINAStreamCommonTypes::t_SBRecover recActions,
      in boolean wait,
      out TINAStreamCommonTypes::t_SBBindState status
) raises (
      TINAUsageCommonTypes::e_UsageError,
      e_PaSBSetupError,
      e_NoSynchronousReqResp
);
```

The `addProviderPaSBReq()` allows a party to request the creation of a new stream binding, which is described by the `reqType` parameter that indicates the overall stream binding type; a `media` parameter that describes particular media type requirements; a list of stream binding participants (including the requester); and success and recovery criteria. The requester is identified by the `myId` parameter, and may optionally specify their own SFEPs for binding.

The overall type may be service specific. It is interpreted with the media type by the participants' components to determine which SFEPs should be returned for binding by the associated exe operations.The success criteria indicate which participants and media types need to be bound to successfully setup the stream binding and to complete subsequent operations. Recovery criteria specify actions to be taken on the failure of the stream binding (or part of the stream binding) and specifies how to determine if the recovery has been successful.

As explained in Section 5.5.9.2, the request may be processed synchronously or asynchronously. A number of steps to indicate the request to parties, initiate actions by and acquire information from stream binding participants, and to setup the underlying communications are required to complete a request. An error at any stage will cause an exception or failure notification. Once the operation is complete, information may be distributed to SB members and the initial state of the stream binding is returned to the requester. Figure 5-19 gives an overview of the scenario. The steps are detailed below and Figure 5-20 shows the event trace diagram.

Pre-conditions:

The requested stream binding members are already members of the session. Stream binding participants have (or can create) SFEPs which are known to the TCSM and are ready to use.

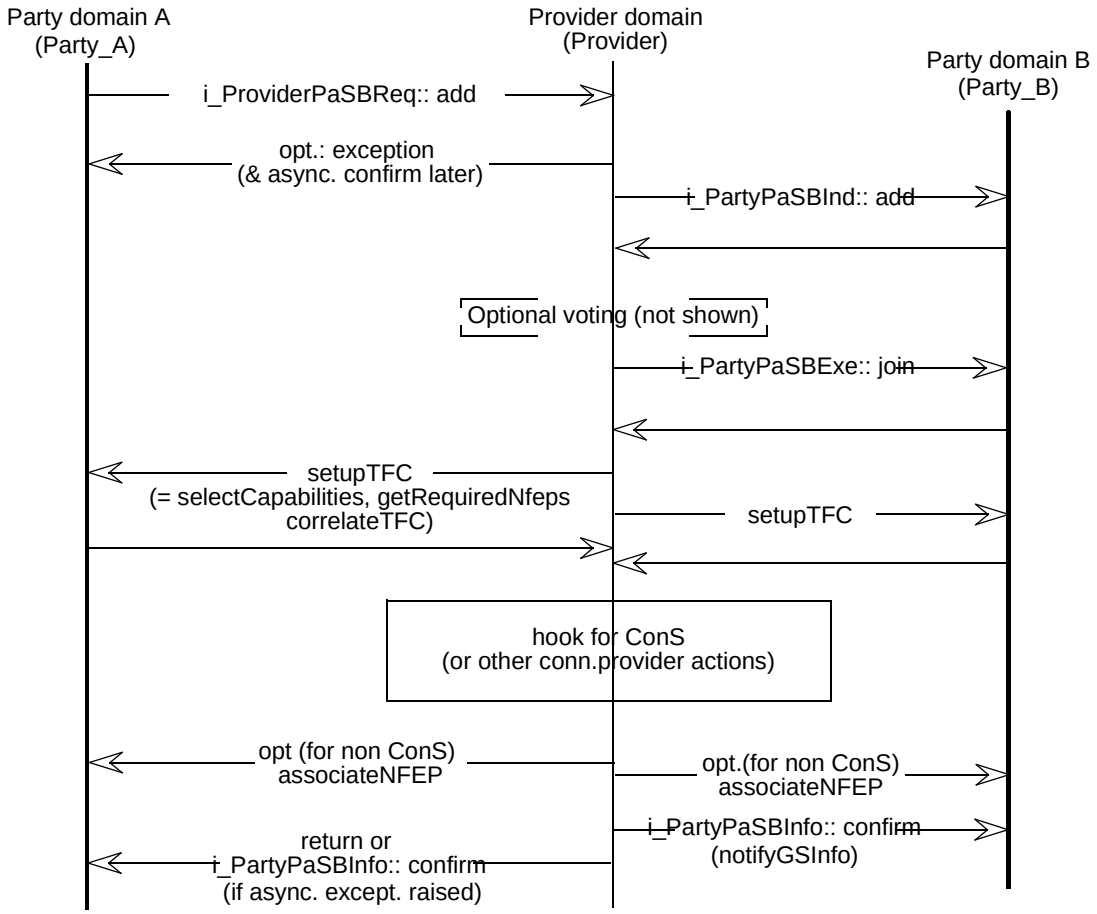


Figure 5-20. Add a stream binding request event trace

Steps:

1. Party A makes an `addProviderPaSBReq()` operation on the provider's `i_ProviderPaSBReq` interface. The operation describes the stream binding to be setup, as detailed above. It may be responded to asynchronously or synchronously.
2. The provider can optionally make `addPartyPaSBInd()` operations on the `i_PartyPaSBInd` interface of the other parties to notify them of the request to create the stream binding, if the parties support the PaSBInd FS (see Section 5.5.10).
3. The provider may wait for votes to be received, if the session supports the Voting FS. If no voting is supported or the request is agreed, then the request proceeds.
4. A `joinPartyPaSBExe()`, see Section 5.5.9.6.1, will be sent to the `i_PartyPaSBExe` interface of all parties identified as SB participants, unless the participant is the requesting party. The `joinPartyPaSBExe()` operation returns a `participantDesc` parameter which describes the SFEPS a participant wants to bind. See Section 5.2.2 and Section 5.5.9.2 for details of Exe operations.
5. Once each participant has returned its SFEPs, the provider determines if the stream binding can proceed (based on success criteria) and runs a stream binding algorithm to determine which SFEPs need to be bound. This is expressed by a set of SFCs. The service session then needs communication support to setup connections between SFEPs. In TINA, this is modelled by the communication session which is considered a separate entity from the service session. We use this model here[2] and assume some operation requests the creation of the SFCs.
6. The communication session, see Section A-X.x.x, is responsible for matching SFEP capabilities, mapping SFCs to NFCs, and coordinating terminal flow connections (TFCs) between SFEPs and NFEPs. Operations to set the capability and correlation identifier of each SFEP and acquire SFEPs that support its required capabilities are needed. Operations on the `i_TerminalFlowControl` interface: `selectCapabilities()`, `getRequiredNfeps()`, and `correlateTFC()` (grouped as `setupTFC()`); support these needs. Queries may be made to establish terminal capabilities using the `queryCapabilities()` operation.
7. The communication session then needs to set up the network communications. In Figure 5-20, this is shown as communication across the ConS to setup NFCs. However, this is not mandatory and other options, including connectionless communications, could be used.
8. The communication session may optionally use an `associateNFEP()` operation to associate the resolved NFEP that is used by the NFC (or equivalent) with the correct TFC. This behavior is supported by the ConS/TCon reference points, but is included here to allow other options.
9. The provider must notify the SB members that stream binding they agreed to join has been setup (using `confirmPartyGSInfo()` or `notifyGSInfo()` operations on the `i_PartyPaSBInfo` interface, see Section 5.5.9.7).
10. Finally, the provider notifies the requester, either synchronously on the return or asynchronously using a `confirmPartyGSInfo()` operation on the `i_PartyPaSBInfo` interface.

2. This separation is not mandatory for the Ret RP. However, a set of communication session support interfaces are specified. These interfaces allow the communication session to establish common capabilities and session protocols necessary to set up connections, correlate SFCs and TFCs, acquire SFEPs, find transport quality and protocol requirements and hence determine NFCs needed to support the SFCs and help locate a connectivity provider. These interfaces, or some equivalent functional support, is necessary to set up stream bindings.

Post-conditions:

A new stream binding has been created for the given participants in the requested state, supporting connections have been setup, and the success criteria have been met. A stream binding identifier is issued which remains valid to the deletion of the stream binding. The success criteria and recovery criteria are stored for later use.

5.5.9.5.2. Add participants to a stream binding request

```
void addParticipantsProviderPaSBReq(
      in TINACommonTypes::t_ParticipantSecretId myId,
      in TINAStreamCommonTypes::t_SBId sbId,
      in TINAPaSBTypes::t_ParticipantDescList reqMembers,
      in TINAStreamCommonTypes::t_SFEPServDescList requesterSIs,
      in boolean wait,
      out TINAStreamCommonTypes::t_SBBindState status
) raises (
      TINAUsageCommonTypes::e_UsageError,
      e_PaSBSetupError,
      e_NoSynchronousReqResp
);
```

The addParticipantsProviderPaSBReq() allows a party to request that it or other session members be added to a stream binding identified by the sbId parameter. The prospective participants and their requirements are given by the reqMembers parameter. If the requester wishes to be added to the binding, they should list their own SFEPs in the requesterSIs parameter.

As before, the request may be processed synchronously or asynchronously. A number of steps to indicate the request to parties, initiate actions by and acquire information from participants, and to setup the underlying communications are required to complete a request. An error at any stage causes an exception or failure notification. Once the request is complete, information may be distributed to SB members and the new state of the stream binding is returned to the requester.

This request operation has the same pre-conditions takes the same steps outlined by Figure 5-19 and detailed in Figure 5-20, except that the addParticipantsProviderPaSBReq and appropriate indication operation (if any) are made.

Post-conditions: The new participants have been added to the stream binding, additional supporting connections setup, and the success criteria (as previously set) have been fulfilled.

5.5.9.5.3. Delete participants from a stream binding request

```
void deleteParticipantsProviderPaSBReq(
      in TINACommonTypes::t_ParticipantSecretId myId,
      in TINAStreamCommonTypes::t_SBId sbId,
      in boolean all,
      in TINAPaSBTypes::t_ParticipantIdList reqMembers,
      in boolean wait,
      out TINAStreamCommonTypes::t_SBBindState status
) raises (
      TINAUsageCommonTypes::e_UsageError,
      e_PaSBOperationError,
      e_NoSynchronousReqResp
);
```

The `deleteParticipantsProviderPaSBReq()` allows a party to request the deletion of itself or other participants from a stream binding. The `reqMembers` parameter lists the identities of the participants to be deleted. All participants may be deleted by setting the `all` flag true. Deleting a participant removes its SFEPs from the binding and any supporting communications. It does not necessarily change control relationships with the stream binding.

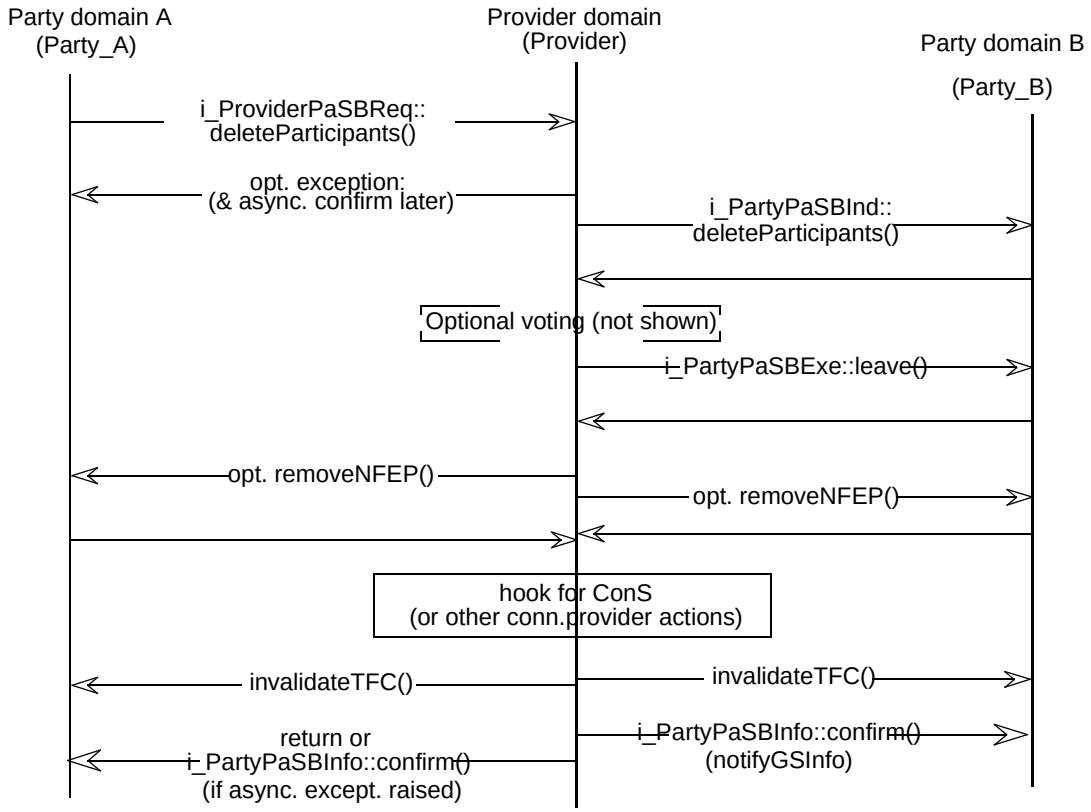


Figure 5-21. Delete participants from a stream binding request event trace

As before, the request may be processed synchronously or asynchronously. A number of steps to indicate the request to parties, initiate actions by participants, and to remove the underlying communications are required to complete a request. An error at any stage will cause an exception or failure notification. Once the request is complete, information may be distributed to SB members and the new state of the stream binding is returned to the requester. The event trace diagram of Figure 5-21 shows the steps, which are also explained below.

Pre-conditions:

The stream binding exists and the session members are existing participants in the stream binding.

Steps

1. Party A makes a `deleteParticipantsProviderPaSBReq()` request, listing the SB participants to be removed, on the provider's `i_ProviderPaSBReq` interface.

2. The provider can optionally make `deleteParticipantsPartyPaSBInd`() operations on the `i_PartyPaSBInd` interface of the other parties to notify them of the request to delete participants from the stream binding if the parties support the PaSBInd FS.
3. The provider may wait for votes to be received, if the session supports the Voting FS. If the vote succeeds or the Voting FS is not supported, the requests proceeds.
4. The provider makes `leavePartyPaSBExe()`, see Section 5.5.9.6.2, calls on the `i_PartyPaSBExe` interface of all the participants to be deleted, unless the participant is the requesting party.
5. Once each participant has agreed to leave, the provider starts deleting SFC branches and SFCs as appropriate. We will assume it uses a communication session to do this.
6. The CSM optionally makes `removeNFEP`() calls on the `i_TerminalFlowControl` interface of each SFEP of each SFC branch to be deleted. This removes the NFEP from the TFC.
7. The communication session then needs to modify the network communications: for ConS RP this would involve deleting the associated NFCs or NFC branches.
8. The communication session then uses the `invalidateTFC()` operation to remove the TFCs supporting each of the SFC branches.
9. The provider confirms the request to deleted participants (using `confirmPartyGSInfo`() or `notifyGSInfo`() operations) on the `i_PartyPaSBInfo` interface. Other SB members may also be notified.
10. Finally, the provider notifies the requester either synchronously on the return or asynchronously using a `confirmPartyGSInfo`() operation on the `i_PartyPaSBInfo` interface.

Post-conditions:

The requested stream binding participants and associated connections have been removed within the previously set success criteria. The stream binding still exists.

5.5.9.5.4. Delete a stream binding request

```
void deleteProviderPaSBReq(
      in TINACommonTypes::t_ParticipantSecretId myId,
      in TINAStreamCommonTypes::t_SBId sbId,
      in boolean wait,
      out TINAStreamCommonTypes::t_SBBindState status
) raises (
      TINAUsageCommonTypes::e_UsageError,
      e_PaSBOperationError,
      e_NoSynchronousReqResp
);
```

The `deleteProviderPaSBReq`() operation allows a party to request the deletion of a given stream binding. This operation deletes all participants and ensures the removal of the stream binding. It has the same preconditions and follows the same steps as the previous operation. Once the stream binding is deleted, the request is complete. SB members are notified of its removal and the request's completion is confirmed to the requester.

Post-conditions: The stream binding has been removed and its identifier is no longer valid.

5.5.9.5.5. Activate participants in a stream binding request

```
void activateParticipantsProviderPaSBReq(
      in TINACommonTypes::t_ParticipantSecretId myId,
      in TINAStreamCommonTypes::t_SBId sbId,
      in boolean all,
      in TINAPaSBTypes::t_ParticipantIdList reqMembers,
      in boolean allFlows,
      in TINASBComSCommonTypes::t_MediaDescList reqFlows,
      in boolean wait,
      out TINAStreamCommonTypes::t_SBBindState status
) raises (
      TINAUsageCommonTypes::e_UsageError,
      e_PaSBOperationError,
      e_NoSynchronousReqResp
);
```

The `activateParticipantsProviderPaSBReq()` allows a party to request the activation of itself or other SB participants. The SB participants to be activated are listed by the `reqMembers` parameter. All SB participants may be activated by setting the all flag to true. When a SB participant is activated, each branch of each SFC terminated by its SFEPs is activated.

Activation can also be targeted at the SFCs associated with particular media types specified by the `reqFlows` parameter. If a media type is activated, then all implicit SFCs associated with that media type are activated. All media types and associated SFCs will be activated if the `allFlows` parameter is set true. If a media type is activated for a particular SB participant, then each branch of the media type's associated SFCs terminated by the SB participant's SFEPs is activated.

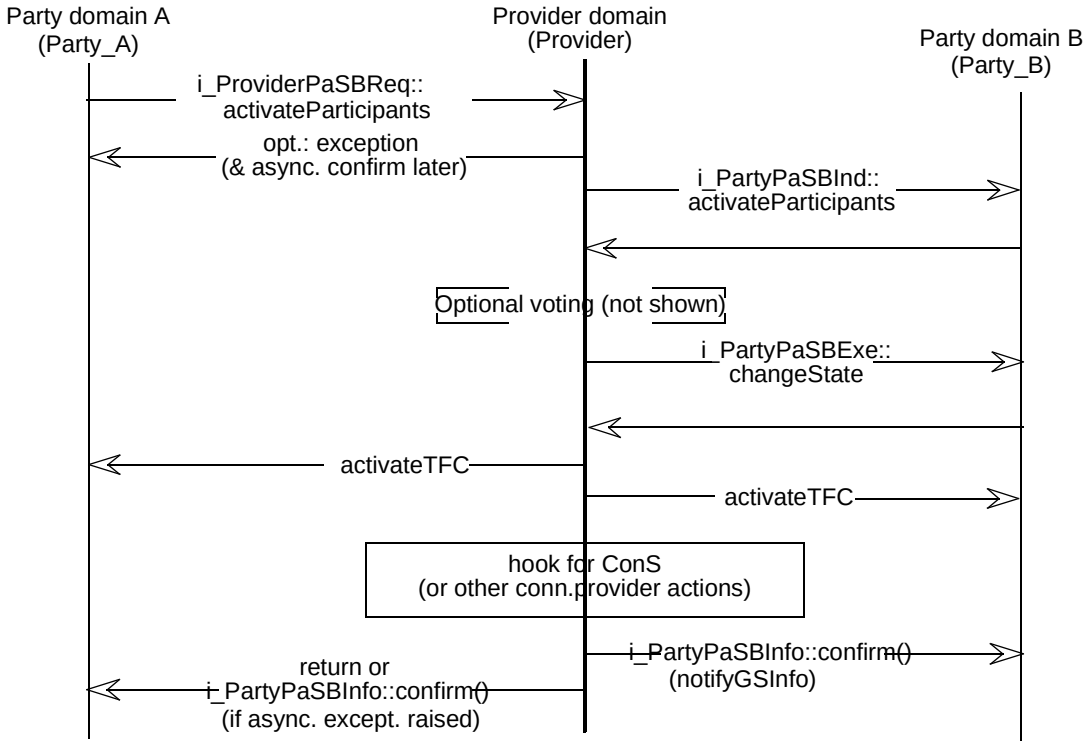


Figure 5-22. Activate stream binding request event trace

This request follows similar steps to the `deleteParticipantsProviderPaSBReq` scenario. It includes optional indication and voting steps, exe operations on SB participants, a communication stage, and information distribution. Once the process is complete, success is confirmed to the requester with the new state of the stream binding. Figure 5-22 shows the steps explained below.

Pre-conditions:

The stream binding exists and the members are already participants in the stream binding.

Steps

1. Party A makes an `activateParticipantsProviderPaSBReq`() operation on the provider's `i_ProviderPaSBReq` interface as described above.
2. The provider can optionally make `activateParticipantsPartyPaSBInd`() calls on the `i_PartyPaSBInd` interface of the other parties to notify them of the request to activate the stream binding if the parties support the PaSBInd FS.
3. The provider may wait for votes to be received, if the session supports the Voting FS. Once the operation is agreed, or if the Voting FS is not supported, the request proceeds.
4. The provider makes `changeStatePartyPaSBExe()`calls on the `i_PartyPaSBExe` interface of all the SB participants to be activated, unless the participant is the requesting party. The administrative state is set to `unlocked` (active).
5. The provider starts activating SFC branches and SFCs as appropriate. As before, we will assume it makes use of a communication session.
6. The CSM makes an `activateTFC`() operation on the `i_TerminalFlowControl` interface of each SFEP of each SFC branch to be activated. This may be optional.
7. The communication session then needs to activate the network communications: for ConS RP this involves activating the associated NFCs or NFC branches.
8. Once the SFCs or SFC branches have been activated, the provider notifies the requester either synchronously on the return or asynchronously using a `confirmPartyGSInfo`() operation on the party's `i_PartyPaSBInfo` interface.
9. The provider confirms the request to SB participants and other parties with `confirmPartyGSInfo`() or `notifyGSInfo`() calls to their `i_PartyPaSBInfo` interfaces
10. Finally, the provider notifies the requester either synchronously on the return or asynchronously using a `confirmPartyGSInfo`() operation on the `i_PartyPaSBInfo` interface and returns the activated state of the stream binding.

Post-conditions: Requested SB participants and media types (and associated SFCs) are active.

5.5.9.5.6. Deactivate participants in a stream binding request

```
void deactivateParticipantsProviderPaSBReq(
      in TINACommonTypes::t_ParticipantSecretId myId,
      in TINAStreamCommonTypes::t_SBId sbId,
      in boolean all,
      in TINAPaSBTypes::t_ParticipantIdList reqMembers,
      in boolean allFlows,
      in TINASBComSCommonTypes::t_MediaDescList reqFlows,
      in boolean wait,
      out TINAStreamCommonTypes::t_SBBindState status
) raises (
      TINAUsageCommonTypes::e_UsageError,
      e_PaSBOperationError,
      e_NoSynchronousReqResp
);
```

The `deactivateParticipantsProviderPaSBReq()` operation allows a party to request the deactivation of itself or other SB participants. The `reqMembers` parameter lists SB participants to be deactivated. All SB participants may be deactivated by setting the `all` flag true. Deactivation can also be targeted at SFCs associated with media types specified by the `reqFlows` parameter. If the `allFlows` parameter is set true, then all media types and associated SFCs will be deactivated.

This request follows the same steps and has the same preconditions as the previous activation scenario, except that deactivate rather than activate operations are used and the administrative state is set to `locked` (inactive) rather than `unlocked`. Once the process is complete, success is confirmed to the requester with the new state of the stream binding.

Post-conditions: Requested SB members and media types (and associated SFCs) are inactive.

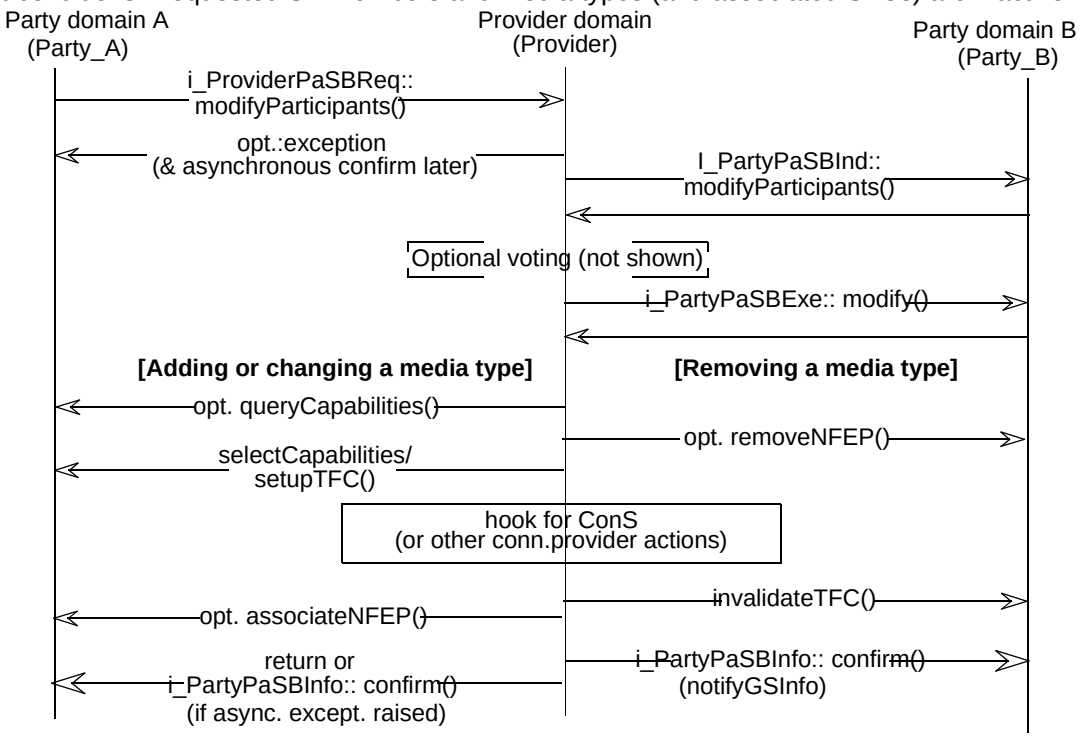


Figure 5-23. Modify stream binding request event trace

5.5.9.5.7. Modify participation in a stream binding request

```
void modifyParticipantsProviderPaSBReq(
      in TINACommonTypes::t_ParticipantSecretId myId,
      in TINAStreamCommonTypes::t_SBId sbId,
      in boolean all,
      in TINAPaSBTypes::t_ParticipantIdList reqMembers,
      in TINASBComSCommonTypes::t_MediaDescList newTypes,
      in TINASBComSCommonTypes::t_MediaDescList oldTypes,
      in TINASBComSCommonTypes::t_MediaChangeDescList modTypes,
      in TINAStreamCommonTypes::t_SFEPServDescList requesterSIs,
      in boolean wait,
      out TINAStreamCommonTypes::t_SBBindState status
) raises (
      TINAUsageCommonTypes::e_UsageError,
      e_PaSBSetupError,
      e_NoSynchronousReqResp
);
```

The `modifyParticipantsProviderPaSBReq()` operation allows a party modify the stream binding by changing the media types supported the by overall stream binding or the given participants (optionally including the requester). The `reqMembers` parameter lists SB participants to be modified. All SB participants may be modified by setting the `all` flag true. New media types (i.e those to be added), modified media types (i.e existing media types to be changed) and old media types (those to be removed) are specified by the `newTypes`, `modTypes`, and `oldTypes` parameters respectively. The requester may specify new SFEPs or modified SFEPs of its own to bind in the `requesterSIs` parameter.

This request follows similar steps to the `addParticipantsProviderPaSBReq` request scenario. It includes optional indication and voting steps, exe operations on SB members, a communication stage, and information distribution. Completion is then confirmed to the requester who receives the new state of the stream binding. Figure 5-23 shows the supporting steps, also described below.

Pre-conditions:

The stream binding exists and the given session members are already participating in the stream binding. SB participants can modify SFEPs or create new ones to meet modification requests.

Steps

1. Party A makes a `modifyParticipantsProviderPaSBReq()` operation on the provider's i_ProviderPaSBReq interface. The operation describes the modifications as detailed above.
2. The provider can optionally make `modifyParticipantsPartyPaSBInd()` operation on the `i_PartyPaSBInd` interface of the other parties to notify them of the request to modify the stream binding, if the parties support the PaSBInd FS.
3. The provider may wait for votes to be received if the session supports the Voting FS. The request proceeds once the request is approved or if the Voting FS is not supported.
4. The provider makes `modifyPartyPaSBExe()` calls to the `i_PartyPaSBExe` interface of all the participants to be modified. The `modifyPartyPaSBExe()` operation returns their new or modified SFEPs via the `participantSIs` parameter.

5. Once each participant has returned their new SFEPs, the provider determines how to modify the stream binding and its associated SFCs for the changed media types. As before, we will assume it uses a communication session to add, delete, or modify SFCs.
6. The communication session's actions depend on what is required. To add or delete a SFC it will make `setupTFC`() or `removeNFEP( )` calls respectively. To modify a SFC it may need to query and reset the capabilities and modify the NFC, which could change the associated NFEP.
7. The communication session then needs to modify the network communications. If using the ConS RP, it can make requests to add, delete or modify NFCs or their branches.
8. The communication session may then optionally make `associateNFEP`() or `invalidateTFC`() calls to the associated terminals for each SFC and NFC as appropriate.
9. The provider may notify SB members that the stream binding has been modified (by `confirmPartyGSInfo`() or `notifyGSInfo`() calls on the `i_PartyPaSBInfo` interface).
10. Finally, the provider notifies the requester either synchronously on the return or asynchronously using a `confirmPartyGSInfo`() operation on the `i_PartyPaSBInfo` interface.

Post-conditions: Media types are modified for the entire stream binding or designated SB participants.

5.5.9.5.8. Modify criteria for a stream binding request

```
void modifyCriteriaProviderPaSBReq(
      in TINACommonTypes::t_ParticipantSecretId myId,
      in TINAStreamCommonTypes::t_SBId sbId,
      in TINAStreamCommonTypes::t_SBSuccessCriteria criteria,
      in TINAStreamCommonTypes::t_SBRecover recActions,
      in TINAPaSBTypes::t_PCriteriaList newPCriteria)
) raises (
      TINAUsageCommonTypes::e_UsageError,
      e_PaSBSetupError
);
```

The `modifyCriteriaProviderProviderPaSBReq`() allows a party to request the modification of success and recovery criteria of the stream binding. Overall success and recovery criteria are described by the `criteria` and `recActions` parameters respectively. It is also possible to modify the criteria of particular participants. The `newPCriteria` parameter lists participants to be changed with their new success and recovery criteria.

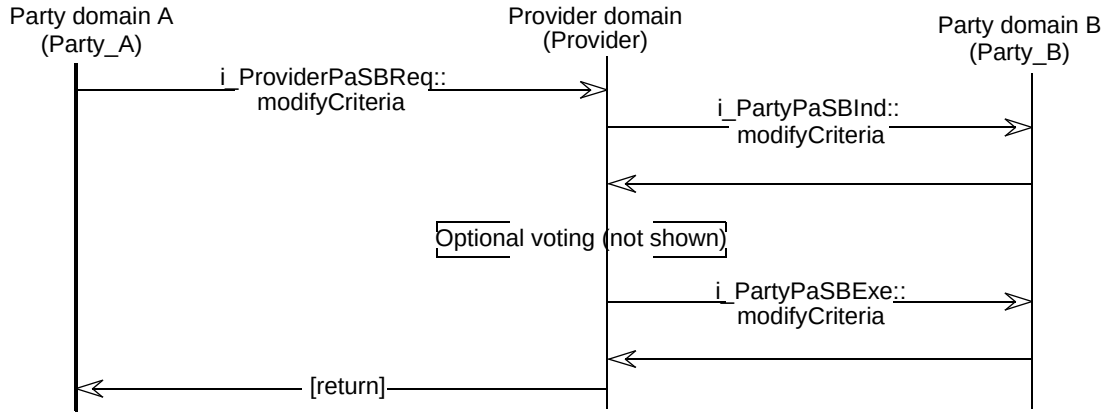


Figure 5-24. Modify a stream binding's success criteria request event trace

The operation returns once the criteria have been agreed, using indications and voting if appropriate, and modified. Exe operations are required to change participant criteria. Figure 5-24 shows the necessary steps which are detailed below.

Pre-conditions:

The stream binding exists and the given session members are existing SB participants.

Steps

1. Party A makes an `modifyCriteriaProviderPaSBReq()` operation on the provider's `i_ProviderPaSBReq` interface. The operation describes the modifications as detailed above.
2. The provider can optionally make `modifyCriteriaPartyPaSBInd()` operations on the i_PartyPaSBInd interface of the other parties to notify them of the request to modify the stream binding's success and recovery criteria, if they support the PaSBInd FS.
3. The provider may wait for votes to be received, if the session supports the Voting FS. Once the change is agreed, or if voting is not supported, the request proceeds. Otherwise an exception is thrown.
4. If any participant's success criteria (other than the requester's) is modified, the provider makes a `modifyCriteriaPartyPaSBExe()` on the parties' `i_PartyPaSBExe` interface.
5. The operation returns: no state information is required.

Post-conditions: Success and recovery criteria of the stream binding or nominated participants has been updated for subsequent operations on that stream binding.

5.5.9.5.9. Notification of sudden change

```
void notifyProviderPaSBReq(
      in TINACommonTypes::t_ParticipantSecretId myId,
      in TINAStreamCommonTypes::t_SBId sbId,
      in TINAStreamCommonTypes::t_SFEPServDescList myStatus
) raises (
      TINAUsageCommonTypes::e_UsageError,
      e_PaSBQueryError
);
```

The `notifyProviderPaSBReq()` operation allows a SB member to inform the provider and other SB members of a (sudden) change in their stream binding participation. The requester is identified by the `myId` parameter. The participants' current participation is described by the `myStatus` parameter. It lists any remaining SFEPs and their currently supported quality and state. The provider may inform other SB members by a `notifyGSInfo()` operation on the `i_PartyPaSBInfo` interface. Figure 5-25 shows the event trace.

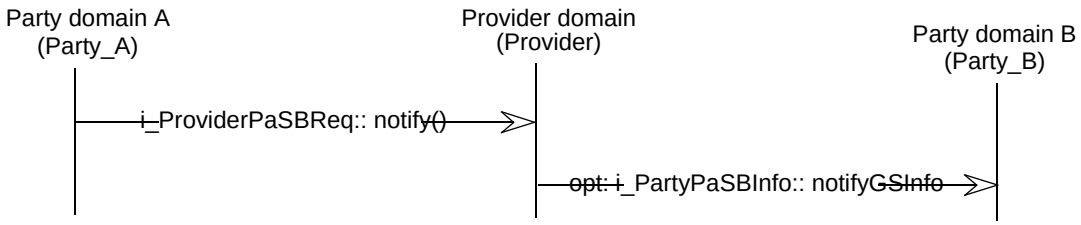


Figure 5-25. Notify change in stream binding participation request event trace

5.5.9.5.10. Register SFEPs

```
void registerSFEPsProviderPaSBReq(
      in TINACommonTypes::t_ParticipantSecretId myId,
      in TINAStreamCommonTypes::t_SBId sbId,
      in TINAStreamCommonTypes::t_SFEPServDescList fepList,
      out TINAStreamCommonTypes::t_SBBindState status
) raises (
      TINAUsageCommonTypes::e_UsageError,
      e_PaSBSetupError,
      e_NoSynchronousReqResp
);
```

The `registerSFEPsProviderPaSBReq()` operation allows a SB participant to register additional SFEPs for use in a particular stream binding. The requester is identified by the `myId` parameter. The requester passes a `fepList` parameter which describe the SFEPs to be registered.

The provider can optionally make `registerSFEPsPartyPaSBInd()` calls on other parties' `i_PartyPaSBInd` interfaces to notify them of the request. If the session supports a voting feature set, the provider may then wait for votes to be received. Once the request is approved, or if no voting feature set is supported, the request may proceed.

The provider then registers the SFEPs and checks if the stream binding algorithm needs to be run. If it does, then the provider needs to modify the stream binding, following steps 5 to 8 of the `addProviderPaSBReq` scenario, see Section 5.5.9.5.1. Once the communication changes are complete, the provider can distribute the registered SFEPs with the `SIDistribPartyGSInfo()` calls on participants `i_PartyPaSBInfo` interface. It may also notify SB members of other stream binding changes. Finally it returns to the requester or confirms the completion of the registration.

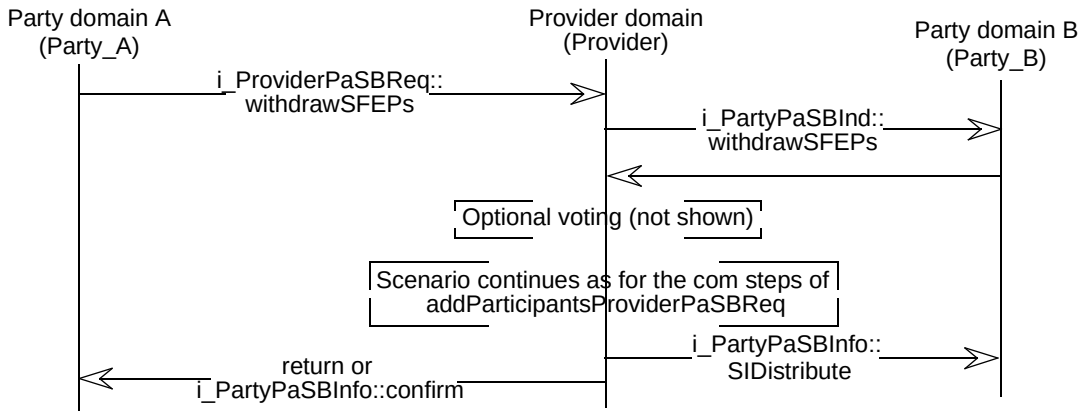


Figure 5-26. Register SFEPs from stream binding request event trace

5.5.9.5.11. Withdraw SFEPs

```
void withdrawSFEPsProviderPaSBReq(
      in TINACommonTypes::t_ParticipantSecretId myId,
      in TINAStreamCommonTypes::t_SBId sbId,
      in TINASBComSCommonTypes::t_SFEPNameList fepList,
      out TINAStreamCommonTypes::t_SBBindState status
) raises (
      TINAUsageCommonTypes::e_UsageError,
      e_PaSBSetupError,
      e_NoSynchronousReqResp
);
```

The withdrawSFEPsProviderPaSBReq() operation allows a SB participant to request the withdrawal of its SIs and SFEPs from a stream binding. The requester is identified by the myId parameter and also passes a fepList parameter which identifies the SFEPs or SIs to be withdrawn.

The operation proceeds in a similar manner to the `registerSFEPsPartyPaSBReq` request. After the optional indication and voting steps, the provider checks if the binding algorithm needs to be rerun. If it does, it proceeds to the communication stage, this time repeating steps 5 to 8 of the `deleteParticipantsProviderPaSBReq` scenario. It notifies participants of the withdrawn SIs and SFEPs by `notifyWithdrawnElementsPartyGSInfo`() calls to their `i_PartyPaSBInfo` interface. Finally it returns to the requester or sends a confirmation of the elements' withdrawal.

5.5.9.5.12. Rebind stream binding request

```
void rebindProviderPaSBReq(
      in TINACommonTypes::t_ParticipantSecretId myId,
      in TINAStreamCommonTypes::t_SBId sbId,
      out TINAStreamCommonTypes::t_SBBindState status
) raises (
      TINAUsageCommonTypes::e_UsageError,
      e_PaSBSetupError,
      e_NoSynchronousReqResp
);
```

The `rebindProviderPaSBReq`() operation allows a SB member to explicitly request the rebinding of a given stream binding: i.e. the rerunning of the bind algorithm. This request is used in conjunction with operations external to this feature set that may affect which parties may be bound to each other.

The operation then proceeds in a similar manner to the `withdrawSFEPsProviderPaSBReq` request. After the optional indication and voting steps, the provider checks if the binding algorithm needs to be rerun. If it does, it proceeds to the communication stage, repeating steps 5 to 8 of the `deleteParticipantsProviderPaSBReq` scenario. It may then notify participants of the stream binding's new state by `notifyGSInfo`() or `confirmPartyGSInfo`() operations on their `i_PartyPaSBInfo` interface. Finally, it returns to the requester or sends a confirmation.

5.5.9.5.13. List stream bindings request

```
void listProviderPaSBReq(
      in TINACommonTypes::t_ParticipantSecretId myId,
      in boolean all,
      in TINACommonTypes::t_ElementIdList participants,
      out TINAStreamCommonTypes::t_SBIdList sbList
) raises (
      TINAUsageCommonTypes::e_UsageError,
      e_PaSBQueryError
);
```

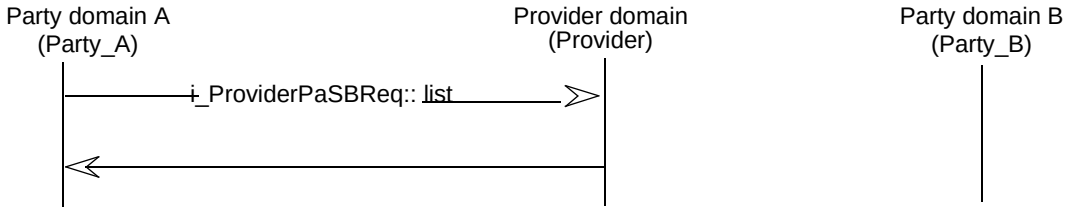


Figure 5-27. List stream bindings request event trace

The `listProviderPaSBReq()` operation allows a party to request a list of stream bindings the session currently supports. The requester is identified by the `myId` parameter. The party may request information on all stream bindings by setting the `all` flag true or they may restrict the list by requesting stream bindings involving particular session members listed by the `participants` parameter. If successful, the operation returns a list of stream binding identifiers in the sbList parameter. Otherwise an exception is thrown.

5.5.9.5.14. Get stream binding information request

```
void getInfoProviderPaSBReq(
      in TINACommonTypes::t_ParticipantSecretId myId,
      in TINAStreamCommonTypes::t_SBId sbId,
      out TINAPaSBTypes::t_SBDesc thisSB
) raises (
      TINAUsageCommonTypes::e_UsageError,
      e_PaSBQueryError
);
```

The `getInfoProviderPaSBReq()` operation allows a party to request a information about a stream binding identified by the `sbId` parameter. If successful, the operation returns a `thisSB` parameter which describes a stream binding in terms of participants and which of their SFEPs are connected. Otherwise, an exception is raised.

5.5.9.6 i_PartyPaSBExe Interface

The `i_PartyPaSBExe` interface consists of a number of exe operations that support requests made on the i_ProviderPaSBReq interface. They allow the provider session to ask stream binding participants for SFEPs and to modify their participation status with in the stream binding. For all exe operations, the session is identified by a `t_SessionId` type `sessionId` parameter, and the stream binding by a `t_SBId` type `sbId` parameter. The original request may be identified by a `t_RequestId` type `reqId` parameter. These parameters allow participants to identify the request and stream binding when related information is received from the `i_PartyPaSBInfo` interface.

```
// module TINAPartyPaSBUsage
```

interface i_PartyPaSBExe

```
{
};
```

5.5.9.6.1. Join a stream binding exe request

```
void joinPartyPaSBExe(
      in TINACommonTypes::t_SessionId sessionId,
      in TINAStreamCommonTypes::t_SBId sbId,
      in TINAStreamCommonTypes::t_SBType reqType,
      in TINASBComSCommonTypes::t_MediaDescList media,
      in TINAPaSBTypes::t_ParticipantIdList others,
      in TINAPaSBTypes::t_ParticipantDesc reqParticipation,
      in TINAStreamCommonTypes::t_RequestId reqId,
      out TINAStreamCommonTypes::t_SFEPServDescList participantSIs
) raises (
      TINAUsageCommonTypes::e_PartyDomainError,
      e_PaSBPartySetupError
);
```

A session uses the `joinPartyPaSBExe`() operation to request parties to join a stream binding.This operation is issued in response to addParticipantsProviderPaSBReq() or addProviderPaSBReq() operations on a `i_ProviderPaSBReq` interface, see Sections 5.5.9.5.1 and 5.5.9.5.2. The session making the exe request is identified by the sessionId parameter, the stream binding to be joined is identified by the sbId parameter, and the originating request is identified by the reqId parameter. These parameters allow participants to identify the operation and stream binding when related information is passed later via the `i_PartyPaSBInfo` interface, e.g. confirming the success or failure of an operation or changes in the stream binding status.

The stream binding is described by a stream binding type, a set of associated media types, and information specific to the party. The party's domain interprets the stream binding type and any media types to determine how many and what kind of SFEPs it needs to offer. If necessary, it acts to create SFEPs to support its requirements. If the party's terminal can support the required SFEPs and it is allows to join the binding, it returns the SFEP descriptions (see Section 5.4.2.5) in the `participantSIs` parameter. Otherwise an exception is thrown.

5.5.9.6.2. Leave a stream binding exe request

```
void leavePartyPaSBExe(
      in TINACommonTypes::t_SessionId sessionId,
      in TINAStreamCommonTypes::t_SBId sbId,
      in TINAStreamCommonTypes::t_RequestId reqId
) raises (
      TINAUsageCommonTypes::e_PartyDomainError,
      e_PaSBPartyExeError
);
```

A session uses the leavePartyPaSBExe() operation to request SB participants leave a stream binding. This operation is invoked in response to deleteProviderPaSBReq() or deleteParticipantsProviderPaSBReq() operations on a i_ProviderPaSBReq interface, see Sections 5.5.9.5.4 and 5.5.9.5.3.

When a SB participant receives a leavePartyPaSBExe operation, it should prepare to be removed from the stream binding. No return is required but an exception may be raised if the request is incorrect, e.g. the stream binding is unknown or the request is not properly approved and authorized.

5.5.9.6.3. Modify stream binding participation exe request

```
void modifyPartyPaSBExe(
      in TINACommonTypes::t_SessionId sessionId,
      in TINAStreamCommonTypes::t_SBId sbId,
      in TINASBComSCommonTypes::t_MediaDescList newTypes,
      in TINASBComSCommonTypes::t_MediaDescList oldTypes,
      in TINASBComSCommonTypes::t_MediaChangeDescList modTypes,
      in TINAStreamCommonTypes::t_RequestId reqId,
      out TINAStreamCommonTypes::t_SFEPServDescList participantSIs
) raises (
      TINAUsageCommonTypes::e_PartyDomainError,
      e_PaSBPartySetupError
);
```

A session uses the modifyPartyPaSBExe() operation to request SB participants to change the media types they support in a stream binding. A modifyParticipantsProviderPaSBReq() operation on a i_ProviderPaSBReq interface, see Section 5.5.9.5.7, triggers this operation. The newTypes, modTypes, and oldTypes parameters describe media types to be added, modified and deleted respectively. Media types help determine which kinds of SFEPs the SB participant should offer. So changing the media types require a participant to change the SFEPs offered.

When a SB participant receives a modifyPartyPaSBExe() operation, it needs to create or modify SFEPs to support new or modified media types. The TCSM must know of new SFEPs and changes to existing ones. The SB participant returns a list of SFEP descriptions of new or modified SFEPs for the stream binding in the participantSIs parameter. SFEPs that are no longer required may be released once the modification request is complete (i.e. a confirmation of the request is received via the i_PartyPaSBInfo interface) and the SFEPs have been withdrawn. To ensure the SFEPs are withdrawn, either the provider should issue a notifyWithdrawnElementsPartyPaSBInfo() notification or the SB participant should make a withdrawSFEPsProviderPaSBReq() request.

5.5.9.6.4. Change stream binding criteria exe request

```
void modifyCriteriaPartyPaSBExe(
      in TINACommonTypes::t_SessionId sessionId,
      in TINAStreamCommonTypes::t_SBId sbId,
      in TINAPaSBTypes::t_ParticipantCriteria newPCriteria
) raises (
      TINAUsageCommonTypes::e_PartyDomainError,
      e_PaSBPartySetupError
);
```

The session uses the `modifyCriteriaPartyPaSBExe()` operation to modify a SB participant's success and recovery criteria. A `modifyCriteriaProviderPaSBReq()` operation on a `i_ProviderPaSBReq` interface (see Section 5.5.9.5.8) may trigger this operation if it modifies the criteria of any SB participants. The `newPCriteria` parameter describe the new success and recovery criteria to be used. These criteria are specific to the participant only. No information is returned by this operation, but an exception is raised if there is an error, e.g. an unknown stream binding or unsupported success criteria.

5.5.9.6.5. Change stream binding state exe request

```
void changeStatePartyPaSBExe(
      in TINACommonTypes::t_SessionId sessionId,
      in TINAStreamCommonTypes::t_SBId sbId,
      in TINASBComSCommonTypes::t_AdministrativeState state,
      in boolean allFlows,
      in TINASBComSCommonTypes::t_MediaDescList reqFlows,
      in TINAStreamCommonTypes::t_RequestId reqId
) raises (
      TINAUsageCommonTypes::e_PartyDomainError,
      e_PaSBPartyExeError
);
```

The session uses the `changeStatePartyPaSBExe()` operation to change a SB participant's administrative state for a given stream binding. The `state` parameter may be set to `unlocked` (active) or `locked` (inactive). By changing this state, the SB member's participation is activated or deactivated. This operation results from an `activateParticipantsProviderPaSBReq()` or a `deactivateParticipantsProviderPaSBReq` operation on a `i_ProviderPaSBReq` interface by another party, see Section 5.5.9.5.5. The `reqFlows` parameter indicates the media types and associated SFEPs that are affected. All SFEPs are changed if the `allFlows` flag is set true. No information is returned by this operation, but an exception is raised if there is an error.

5.5.9.7 i_PartyPaSBInfo Interface

This interface is used to support general stream binding information distribution to parties. It supports asynchronous responses to requests, confirmation of operations to other parties, distribution of SFEP and SI information, and general notifications. Generally, these operations identify the session by a `t_SessionId` type `sessionId` parameter. Stream bindings are identified by a `t_SBId` type `sbId` parameter.

```
// module TINAPartyPaSBUsage

interface i_PartyPaSBInfo
                : i_PartyGeneralStreamInfo
{
};
```

All of the operations `i_PartyPaSBInfo` are inherited from the `i_PartyGeneralStreamInfo` interface. This is a general stream information interface that allows the session to make status reports on a synchronous operations; the distribution of Stream Interfaces (in a very simple way); on the withdrawals of SIs (and other elements) and the notification of communication errors. `i_PartyGeneralStreamInfo`interface inherits some of this operations from `i_GeneralStreamInfo.`

Currently, `i_PartyPaSBInfo` does not provide any additional operations over those defined for `i_PartyGeneralStreamInfo`.

5.5.9.7.1. Confirm request information operation

```
oneway void confirmPartyGSInfo(
      in TINACommonTypes::t_SessionId sessionId,
      in TINAStreamCommonTypes::t_RequestId reqId,
      in TINAStreamCommonTypes::t_RequestType reqType,
      in TINAStreamCommonTypes::t_SBBindState info
);
```

This operation confirms a previous request has succeeded and optionally give the current state of the stream binding. The request is identified by `reqId` and `reqType` parameters. The stream binding state is described by the `info` parameter in terms of participants and SFEPs bound. This operation may result from any asynchronously handled operation on the `i_ProviderPaSBReq` interface. It may also be used to pass information to SB members involved by exe or indication operations on their `i_PartyPaSBExe` or `i_PartyPaSBInd` interfaces.

5.5.9.7.2. Request failure information operation

```
oneway void failurePartyGSInfo(
      in TINACommonTypes::t_SessionId sessionId,
      in TINAStreamCommonTypes::t_RequestId reqId,
      in TINAStreamCommonTypes::t_RequestType reqType,
      in TINAStreamCommonTypes::t_FailureCode error,
      in boolean additionalInfo,
      in TINAStreamCommonTypes::t_ReqProblem reqProblem
);
```

This operation notifies recipients of the failure of a previous request and the reasons for the failure.As before, the request is identified by `reqId` and `reqType` parameters. The reason for the failure is given by an appropriate error code for the request type and an optional parameter: req`Problem`. For problems caused by invalid request parameters, usually the problemEl (element associated with a problem) or problemParam (non-element parameter causing a problem) is returned. Other problems may result in lists of failed elements (e.g. criteria not met errors). This operation may result from any asynchronously handled operation on the `i_ProviderPaSBReq`. It may also be used to pass information to SB members involved by exe or indication operation on their `i_PartyPaSBExe` or `i_PartyPaSBInd` interfaces.

5.5.9.7.3. SI distribution operation

```
oneway void SIDistribPartyGSInfo(
      in TINACommonTypes::t_SessionId sessionId,
      in TINAStreamCommonTypes::t_SBId sbId,
      in TINAStreamCommonTypes::t_SIDescList newSIs
);
```

This operation distributes stream interface information among SB members.This operation may be used to distribute SI and SFEP information after creating the stream binding, adding or modifying participants, or registering SFEPs. The SFEP information is grouped into SI descriptions, which include the owner's identity as well as the SI identity and reference.

5.5.9.7.4. SFEP distribution operation

```
oneway void SFEPDistribPartyGSInfo(
      in TINACommonTypes::t_SessionId sessionId,
      in TINAStreamCommonTypes::t_SBId sbId,
      in TINAStreamCommonTypes::t_SFEPServDescList newSFEPs
);
```

This operation distributes SFEP information among SB members.This operation may be used to distribute SFEP information after creating the stream binding, adding or modifying participants, or registering SFEPs. However, since SFEPs do not indicate their owners, this operation is best suited to distributing additional SFEPs associated with known SIs after a stream binding modification or SFEP registration request.

5.5.9.7.5. Notify withdrawal of elements operation

```
oneway void notifyWithdrawnElementsPartyGSInfo(
      in TINACommonTypes::t_SessionId sessionId,
      in TINAStreamCommonTypes::t_SBId sbId,
      in TINACommonTypes::t_ElementIdList gone
);
```

This operation informs SB members of the withdrawal of certain stream binding elements, in particular SIs and SFEPs.This operation may be used after deleting or modifying SB participants, a notification of sudden participation change, or the withdrawal of SFEPs or SIs.

5.5.9.7.6. General notification operation

```
oneway void notifyGSInfo(
      in TINASBComSCommonTypes::t_Notification event
);
```

This operation notifies SB members and other parties of:

- The addition of a stream binding;
- The modification of a stream binding by adding or deleting participants; changing the state of a stream binding or any part of it; registering or withdrawing SFEPs; and adding, deleting or modifying media types (and hence their associated SFCs);
- The deletion of a stream binding;
- Errors during the execution of operations on a stream binding;
- An error resulting in the full, partial, or temporary loss of a stream binding.

Stream binding related notifications are defined in terms of the event type (i.e. add stream binding), a stream binding identifier, and a description of the stream bindings state (i.e. participants and SFEPs bound). The session identifier (if needed) may be included in the notification parameters.

5.5.9.7.7. Update on error notification operation

```
oneway void notifyUpdateGSInfo(
      in TINASBComSCommonTypes::t_NotifyIdentifier changedEvent,
      in TINASBComSCommonTypes::t_StatusInfo eventChange
);
```

This operation notifies SB members and other parties of changes relating to a previous notification due to an error. It relates the most recent change of stream binding status or other information (such as when it might be recovered).

5.5.9.7.8. Cancel error notification operation

```
oneway void notifyCancelGSInfo(
      in TINASBComSCommonTypes::t_NotifyIdentifier changedEvent
);
```

This operation notifies participants and other parties of cancelation of a previous notification due to an error. It means that the stream binding has been recovered (fully or partially) or has been permanently lost. No further actions related to the notification can be taken.

5.5.10 Participant Oriented Stream Binding Indications (PaSBInd) Feature Set

The PaSBInd feature set allows a session to indicate that an action will be taken shortly (e.g. a stream binding is going to be added). Parties may be able to vote on whether they wish this action to be taken if the session also supports the Voting feature set or equivalent. The PaSBInd feature set is dependant on the session supporting the PaSB feature set.

5.5.10.1 Interfaces

The following interfaces form the Participant SB Ind feature set.

Table 5-18. Participant SB Ind Feature Set Interfaces

ParticipantSBInd interfaces on:	
Party domain components	i_PartyPaSBInd
Provider domain components	(none)

i_PartyPaSBInd: Allows the service to inform a party that actions to create a stream binding or modify an existing one will be taken shortly. Stream bindings and operations on them are specified in terms of participants, overall stream binding type, and media types. When a member is informed of an action, it may be able to vote on it, if the session supports voting.

- Add stream binding: indicate the type, media and initial participants of a proposed stream binding.
- Delete stream binding: indicate proposal to entirely delete a stream binding.
- Add participants to a stream binding: indicate the proposed participants to be added to an existing stream binding and their particular requirements.
- Delete participants from the stream binding: indicate participants to be deleted from the stream binding.
- Modify the stream binding: indicate proposed changes in the supported media types of the entire stream binding or of the given participants. Media types may be added, deleted, or modified.
- Activate participants within a stream binding or an entire stream binding: indicate the participants and media types to be activated in the stream binding.
- Deactivate participants within a stream binding or an entire stream binding: indicate the participants and media types to be deactivated in the stream binding.

5.5.10.2 i_PartyPaSBInd Interface

This interface is used to allow indications of stream binding operations. Its operations are determined by the operations of the `i_ProviderPaSBReq` interface, see Section 5.5.9.5. In general, the interface has operations in response to all requests, except the information requests (`listProviderPaSBReq` and `getInfoProviderPaSBReq`) or status change notifications (`notifyProviderPaSBReq`). These operations are to notify parties of impending changes. If they support the voting feature set, then they may allow or disallow operations.

```
// module TINAPartyPaSBIndUsage

interface i_PartyPaSBInd
{
};
```

The indication operations take the same name as the corresponding request operations except that "Req" has been replaced by "Ind", and "Provider" by "Party". The indication operations parameters are based on the corresponding `i_ProviderPaSBReq` request operations with the following general alterations:

- The `myId` parameter is replaced by a `t_SessionId` type `sessionId` parameter, a `t_RequestId` type `reqId` parameter, and a `t_IndId` type `indId` parameter.
- The requester's SI or SFEP information parameters, any asynchronous request related parameters or output parameters are omitted.
- The following exceptions may be returned for all indications (regardless of the request operation and its exceptions):
  - `TINAUsageCommonTypes::e_PartyDomainError`
  - `TINAIndCommonTypes::e_IndError`
  - `e_PaSBIndError`

For example the corresponding indication operation to the `deleteProviderPaSBReq` operation is:

```
void deleteParticipantsPartyPaSBInd(
      in TINACommonTypes::t_SessionId sessionId,
      in TINAUsageCommonTypes::t_IndId indId,
      in TINAStreamCommonTypes::t_RequestId reqId,
      in TINAStreamCommonTypes::t_SBId sbId,
      in boolean all,
      in TINAPaSBTypes::t_ParticipantIdList reqMembers
) raises (
      TINAUsageCommonTypes::e_PartyDomainError,
      TINAUsageCommonTypes::e_IndError,
      e_PaSBIndError
);
```

The `indId` parameter is used in to identify the indication in later voting, if any is supported. The `reqId` parameter is used to identify subsequent confirmations or notifications associated with request. Other parameters and parameter types are the same as those used in the PaSB feature set. Section 5.5.9.5.3 shows the corresponding request operation.

There are some exceptions to these basic guidelines. The register and withdraw SFEP indication operations (`withdrawSFEPsPartyPaSBInd`() and `registerSFEPsPartyPaSBInd`()) do include the requester's SFEP information.

As well, simplifications have been made to the parameters used in `addPartyPaSBInd`() and `addParticipantPartyPaSBInd`() operations which send participant identifiers rather than the full participant descriptions used in the corresponding requests. This was done to simplify the processing required for voting. As an example, the `addParticipantPartyPaSBInd`() operation is shown below. Section 5.5.9.5.2 shows the corresponding request operation.

```
void addParticipantsPartyPaSBInd(
      in TINACommonTypes::t_SessionId sessionId,
      in TINAUsageCommonTypes::t_IndId indId,
      in TINAStreamCommonTypes::t_RequestId reqId,
      in TINAStreamCommonTypes::t_SBId sbId, // stream binding id
      in TINAPaSBTypes::t_ParticipantIdList reqMembers
) raises (
      TINAUsageCommonTypes::e_PartyDomainError,
      TINAUsageCommonTypes::e_IndError,
      e_PaSBIndError
);
```

Additional operations are defined on the `i_PartyPaSBInd` interface

5.6 TINA Communication Session Model

The TINA Communication Session Model describes the interfaces required to support communication session level interactions across the Ret-RP. The communication session does not (currently) support requests for connections from party domains: these types of requests are handled by the stream binding feature sets of the TINA service session model. Instead, the communication session supports lower level requests required to set up Stream Flow Connections that support stream bindings initiated by service sessions (or their members).

The TINA communication session model is identified by the string “TINACommSessionModel”, in the access and usage parts of Ret-RP.

The communication session interface specifications here are not mandatory. The interface described here only supports basic functionality. Other interfaces may be used, but should at least support the basic functionality specified here. Extra abilities and communications from the party to the provider may be desired. A list of desirable extra functionality can be found at the end of this section.

Currently the communication session model does not define its interfaces in terms of feature sets. This is because the functionality defined is equal to a basic feature set for the communication session. When interfaces to support the extra functionality are defined, the session model will be structured according to feature sets. The interfaces defined here will become the basic feature set for the TINA communication session model.

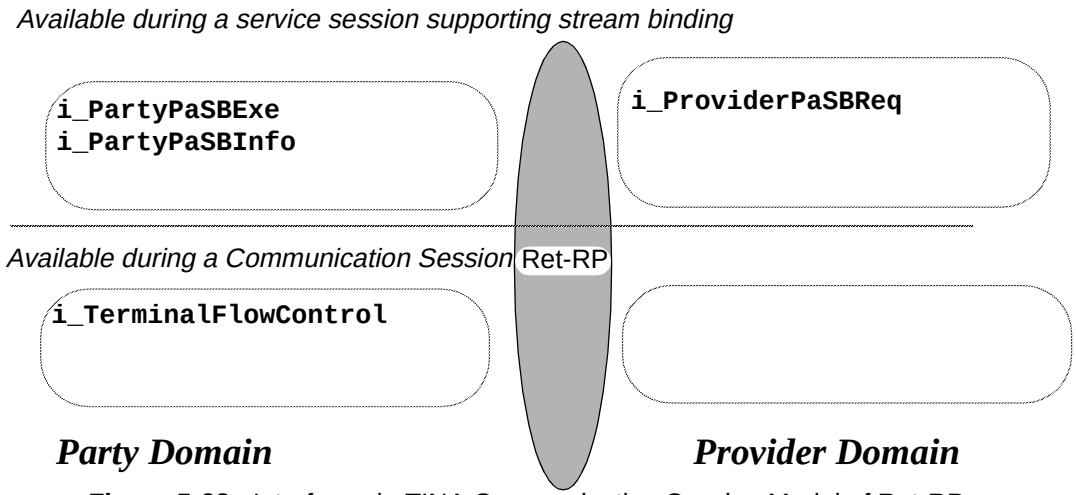


Figure 5-28. Interfaces in TINA Communication Session Model of Ret-RP.

5.6.1 Communication Session Model Information View

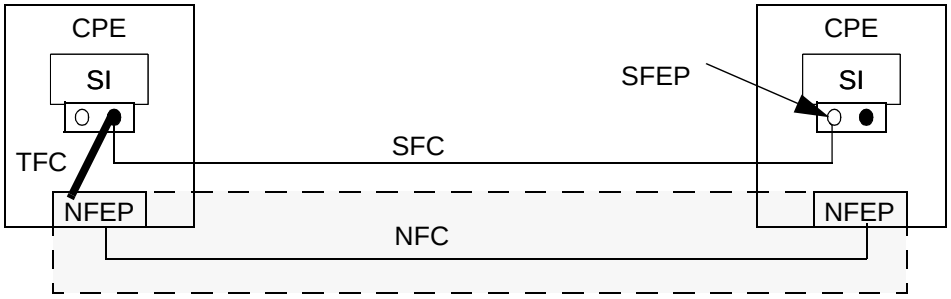


Figure 5-29. Simple relation between SFCs, NFCs and TFCs.l

The communication session is concerned with the establishment of Stream Flow Connections (SFCs) to support service level stream bindings. To setup an SFC, the communication session needs to establish Terminal Flow Connections (TFCs) for each terminal between SFCs and associated Network Flow Connections (NFCs). The Ret-RP is only concerned with the interactions between the provider and party domains required to configure SFEPs and to establish and control TFCs. This section will introduce the information models supporting this interface.

Figure 5-30 shows the relation between SFCs, NFCs and TFCs. The TFC provides the links between the SFC's SFEPs and the NFC's NFEPs. Point-to-point, point-to-multipoint and bidirectional topologies are allowed. These topologies result in the following options:

- NFEP (sink/bidirectional) to one or more SFEPs (sink);
- SFEP (source) to one or more NFEPs (source/bidirectional);
- SFEP (source) to multiple SFEPs (sinks - internal branches) or NFEPs (source);
- NFEP (bidirectional) to SFEP (sink) and SFEP (source). This option allows unidirectional SFCs map to bidirectional NFCs.

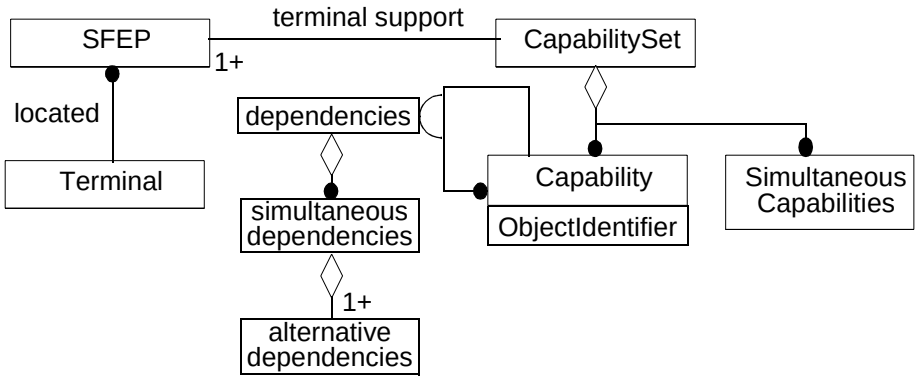


Figure 5-30. SFEPs and Supporting Capabilities.l

The Ret-RP needs to support functionality to initiate TFCs for the above topologies, and support their modification and deletion. To initiate a TFC, the branches of the TFC need to be defined, either in terms of known SFEPs (known from the SFCs) and NFCs (the network connections to which the CSM intends to map the SFCs). The NFEPs are not initially known by the communication session. Instead it knows of ANfeps that represent either a NFEP or a group of NFEPs (or an ability to create NFEPs). This information is used at lower layers to select a suitable NFEP for an NFC.

As well, the Ret-RP needs to support the configuration of SFEPs to support SFC type and quality requirements. At the service level, quality is described in service related terms, e.g. FM or CD quality audio. At the communication level, these requirements are translated into a set of supporting session and terminal capabilities (e.g. codecs). The communication session needs to determine the capabilities available for each SFEP and select which ones will be used

A group of SFEPs is located on a terminal. The terminal supports the SFEPs by a capability set that consists of a list of possible capabilities and lists of simultaneously supported capabilities, see Figure 5-30. This means that using one capability may exclude using another capability for a different SFEP if the two capabilities are not in the set of simultaneously supported capabilities. Capabilities types include terminal, session, and transport related capabilities. A capability may be dependent on a number of other capabilities. Dependencies are described by sets of simultaneous dependencies which are made up of a list of capabilities that may be used as alternatives to each other.

5.6.1.1 Terminology

The following terminology is used to describe communication session functions and requirements. This is used in conjunction with terminology previously described for stream bindings.

- **ANfep**: Represents a potential network termination. It is the super class of the NFEP and NFEPpool. An ANfep description includes layer network technology, direction, and a list of descriptive attributes. Attributes may include the transport quality and associated connectivity provider. They are used to determine which ANfeps to include in a NFC and which connectivity provider to use to setup a NFC.
- **Capability**: Describes an ability to support certain functionality, e.g., an ability to support an audio stream with a particular type of coding from a particular type of codec. Capabilities may be associated with terminal, session or transport requirements. TINA capability descriptions draw on the H.245 standards.
- **Capability Set**: Represents terminal support for SFEPs. It includes a list of capabilities associated with the SFEPs.[1] It also lists the sets of capabilities it can support simultaneously. Use of some capabilities may exclude the use of others.
- **Correlation Identifier**: Is the TFC branch identifier, unique to the terminal. It is used to correlate a NFEP (selected by connectivity layers) with the TFC branch (and hence SFEP) with which it is associated. This allows the completion of the TFC branch setup.
- **Dependencies**: Capabilities are not all independent. One capability may require a number of other types capabilities present to be used. Other capabilities may be used as alternatives to one another.
  - **Simultaneous Dependencies**: A set of capabilities types that are required together to support a capability. Each type is described by an alternative dependencies set.
  - **Alternative Dependencies**: A set of capabilities that may be used as alternatives to each other to support another capability.
- **Initiation**: The communication session may initiate a TFC or a TFC branch. However a TFC branch is not be completely set up until the ANfeps returned by the initiation step is resolved to a NFEP and this NFEP is associated with the TFC branch.
- **Completion**: The setup of a TFC branch by the association with a resolved NFEP.
- **Terminal**: equipment in the party domain terminating a SFC connection.

1. SFEPs of different types and service quality are associated with different capabilities - e.g. an audio SFEP is only associated with capabilities supporting audio.

- **Terminal Flow Connection (TFC)**: A point-to-point or point-to-multipoint connection between a SFEP and NFEPs/SFEPs or a NFEP and SFEPs. TFCs also support bidirectional topologies between a sink and source SFEP and bidirectional NFEP. Each TFC branches represents the terminal part of a SFC branch, usually joining it to a NFC.

5.6.1.2 Communication Session related parameters

The communication session shares a number of common parameters with stream bindings. It also requires the following communication related parameters. Of these, ANfep descriptors and NFC names are also shared with the ConS parameter set.

- `t_AlternativeCapabilities`: A list of mutually exclusive capabilities. Only one of the set may be used at a time. Capabilities are described by their identifiers.
- `t_AlternativeDependencies`: Describes a set of alternative capabilities on which another capability is dependent. This capability needs to be used in conjunction with one of these capabilities.
- `t_ANfep`: Describes an ANFep. It gives the ANfep name, the layer technology, directionality, and the type (NFEP or NFEPpool). It also includes list of descriptive attributes that may be used to describe transport quality and other requirements. If it is an NFEPpool type ANfep, it may include a list of NFEPs or NFEPPool within the NFEPpool.
- `t_ANfepList`: A sequence of ANfep descriptions.
- `t_BranchUpdate`: Describes the updates required to a TFC branch after a change of SFEP capabilities. It includes the branch's correlation identifier, the type of NFEP change required (see t_NFEPUpdate), and describes the required NFEP changes or the new connection requirements and ANfeps using a t_ANfep list.
- `t_Capability`: Describes a capability in terms of:
  - Capability description scheme identifier (e.g. ASN.1);
  - Local capability type and instance identifiers;
  - Directionality (i.e. receive, transmit, or receive and transmit associate capability);
  - Simultaneous Dependencies;
  - Descriptive attributes.
- `t_CapabilityDescriptor`: Identifies a capability and the set of simultaneously supported capability types. Each simultaneously supported type is described by a set of mutually exclusive capabilities using a t_AlternativeCapabilities data type.
- `t_CapabilityList`: A sequence of `t_Capability`.
- `t_CapabilitySet`: This data type describes a terminal's capability set. It lists the capabilities the terminal potential supports and the simultaneously available capabilities.
- `t_CorrelationId`: A TFC branch identifier unique to the terminal.
- `t_CorrelationIdList`: A sequence of correlation identifiers.
- `t_NFCCorrelation`: A correlation identifier, a NFC, and a list of ANfeps associated with a TFC branch.
- `t_NFCCorrelationList`: The correlation information returned after initiating a SFEP to multiple NFEP TFC or new branches of such a TFC.
- `t_NFCName`: The name of a NFC to be associated with a TFC branch.

- `t_NFEPUpdate`: Describes the updates required to TFC branches' supporting NFEP after a change of SFEP capabilities. Possible changes are: no change, modify the existing NFEP, or select a new NFEP.
- `t_SFEPCorrelation`: A correlation identifier and SFEP associated with a TFC branch.
- `t_SFEPCorrelationList`: The correlation information returned after initiating a NFEP to multiple SFEP TFC or new branches of such a TFC.
- `t_SFEPSelect`: A SFEP and it required capability lists. The capability list describes the capabilities to be associated with the SFEP. These need to be reserved for its use.
- `t_SFEPSelectList`: A sequence of `t_SFEPSelect`.
- `t_SimultaneousDependencies`: Describes a set of capability types on which another capability is dependent. This capability needs to be used in conjunction with all of these capability types. The capability types are described by `t_AlternativeDependencies`.
- `t_TFCName`: A TFC identifier, unique in the party domain. It is set by the TCSM on the initiation of a TFC.

5.6.2 Communication Session Model Interfaces

The communication session is separate from the service session. However, as it is part of Ret-RP usage part, it also needs to be considered by the Ret-RP. In regards to Ret-RP, the provider part of the communication session coordinates with terminals to establish TFCs. These TFCs form the terminal part of a SFC and need to be associated SFCs and NFCs. This functionality is supported by set of interfaces on the party domains' and provider domain's components. These components are referred to as the Communication Session Manager (CSM) and the Terminal Communication Session Manager (TCSM). In the TINA service architecture, these components are separate from the service components. However, this only recommended - it is not mandatory.

The interfaces assume service level stream binding support is used to specify the SFCs it supports. It assumes that SFEPs that it is passed are already in existence and are known to the TCSMs. It also assumes that the TFC is not yet fully established. The provider (via the CSM) uses the TCSM to initiate TFCs or TFC branches and associate them with SFC and NFCs. A TFC may be point to point (i.e. SFEP to NFEP), or point to multipoint (NFEP to multiple SFEPs, SFEP to multiple NFEPs/ SFEPs). Each branch is identified by a correlation identifier.

When a NFC associated with a TFC branch is established, the correlation identifier is used by the TCSM to complete the TFC. If the communication session is supported by the ConS and TCon reference points (ConS implies TCon), the correlation identifier is passed over the TCon reference point. The Terminal Layer Adapter, the party component supporting the TCon reference point, uses this identifier plus the selected NFEP to request the TCSM to complete the TFC branch. As the communication session needs to cope with non-ConS reference points, the functionality for completing a TFC branch is also supported between the TCSM and CSM.

5.6.2.1 Interfaces

The following interfaces are required to support the communication session over Ret-RP.

Table 5-19. TINA Communication Session Model Interfaces

Interfaces on:	
Usage party domain components	i_TerminalFlowControl
Usage provider domain components	

5.6.2.1.1. i_TerminalFlowControl

i_TerminalFlowControl: This interface supports the setup of the nodal (TFC) parts of a SFC and its coordination with the physical (NFC) parts of the SFC. It supports the following functionality.

- Query Capabilities: Determine the terminal capabilities available for a given set of SFEPs.
- Select Capabilities: Set the of capabilities (e.g. codecs, session protocols) of a SFEP[2].
- Initiate a TFC:
  - Initiate a TFC for a number of branches described by SFEPs or NFCs. Return the correlation identifier and ANFEPs that are associated with each branch. The TFC setup is not completed until the ANfeps are resolved to a NFEP and it is returned to the TCSM.
- Resolve: Set the terminal capabilities for a SFEP and initiate the TFC for the SFEP.
- Activate: Activate a TFC or TFC branch. This allows the CSM to activate the SFC.
- Deactivate: Deactivate a TFC or TFC branch. This allows the CSM to deactivate the SFC.
- Update: Update a TFC following a change of SFEP capabilities
- Delete:
  - Delete a TFC or TFC branch, invalidating the correlation identifier(s). When a SFC or SFC branch is released, the associated TFCs must be released also.
- Associate:
  - Associate TFC branches, specified by correlation identifiers, with a particular NFEP. The TCSM can complete the TFC branches on receiving this notification. This allows the communication session to complete TFCs if required (i.e no ConS/TCon like support).
- RemoveNFEP:
  - Remove an NFEP from a TFC branch. This allows a SFC to migrate from one NFC to another if required. It may also be used when releasing an SFC or SFC branch.

5.6.2.2 Components and interfaces

5.6.2.2.1. Party domain components (TCSM)

The TCSM interacts with the CSM to support requests to aid the setup, modification, and release of SFCs. In particular it can associate SFEPs and their associated TFC branches with particular SFCs and NFCs, modify SFEPs and associated TFC branches, and activate or deactivate TFCs.The TCSM

2. These capabilities determine the transport quality an NFC branch needs to support

is responsible for the completion of the TFC and may interact with the TLA or the CSM to achieve this. The TCSM must also support SFEP registration, but this is not supported by the Ret-RP. Ret-RP functionality is supported by the i_TerminalFlowControl interface on the TCSM.

5.6.2.2.2. Provider domain Components (CSM)

The CSM supports the establishment of SFCs. It allows its clients (service sessions) to add, activate, deactivate or remove SFCs via the associated session control interface. It also allows clients to manipulate individual stream flows. Each stream flow connection has an interface related sol y to it. Similarly, each communication session control interface is dedicated to a single CSM. This functionality is not supported by the Ret-RP.

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

Finally, the CSM interacts with connectivity level components to setup, modify, and delete NFCs.This functionality is supported by the ConS reference point. We usually assume that these components conform to ConS, but different components could be used. To ensure that the communication session can function independently of the underlying connectivity level, we have included operations for completing and removing TFCs at the communication level.

5.6.2.2.3. i_TerminalFlowControl Interface

In the following operations, SFEPs are identified by `t_SFEPName`, NFCs by `t_NFCName`, TFCs by `t_TFCName`, and TFC branches by `t_CorrelationId`.

```
    // module TINAPartyCommSUsage

interface i_TerminalFlowControl
                  : i_BasicTerminalFlowControl
    {
    };
```

5.6.2.2.3.1 Query capabilities that the SFEP can support

```
    void queryCapabilities (
          in t_SFEPNameList sfeps,
          out t_CapabilitySet capabilities
    ) raises (
          e_PartyDomainError,
          e_CSQueryError
    );
```

This operation allows the communication session to query a terminal for the capabilities available to support the specified SFEPs. If successful, this operation returns the terminal's capability set which a table of capability descriptions, and a list of restrictions that apply to use of these capabilities. Otherwise, it raises an exception. This operation will be required for setting up each branch of an SFC, unless capability information is included in the SFEP description.

5.6.2.2.3.2 Select capabilities for an SFC

```
void selectSFCCapabilities (
      in t_SFEPName sfep,
      in t_CapabilityList localCaps,
      in t_SinkAttributes sfcCaps,
      out t_CapabilityList commonCaps,
      out t_CapabilityList transportReqs
) raises (
      e_PartyDomainError,
      e_CapabilityError,
      e_CSQueryError
);
```

This operation allows a CSM to call on a TCSM to select common capabilities for an SFC, given the preferred capabilities from its capability set and capability attributes of other SFEPs in the SFC. Typically, this operation is called on the source SFEP before any capability selection operations are called on the sink SFEPs, see Figure 5-31. It assumes the CSM does not have the knowledge to make a selection. This operation reserves the requested capabilities for the local SFEP. These capabilities are no longer available for other SFEPs and should not be returned by subsequent capability queries.

If successful, the operation returns the common capabilities to select for other SFEPs in the SFC. It also returns a set of transport requirements. These may include protocol and QoS requirements needed by the TCSM's terminal to support the given SFEP capabilities. Otherwise the operation throws an exception.

5.6.2.2.3.3 Select capabilities for an SFEP

```
void selectSFEPCapabilities(
      in t_SFEPName sfep,
      in t_CapabilityList localCaps,
      out t_CapabilityList transportReqs
) raises (
      e_PartyDomainError,
      e_CSQueryError,
      e_CapabilityError
);
```

This operation allows a communication session to set the particular capabilities the SFEP requires. These capabilities are passed by the `localCaps` parameter. This operation is needed to ensure that SFEPs support compatible capabilities. If successful the operation reserves the requested capabilities. These capabilities are no longer available for other SFEPs and should not be returned by subsequent capability queries. It returns also a list of transport quality requirements. Otherwise, an exception is raised. This operation is required to setup SFC branches prior to initiating TFCs. It may be used in conjunction with the `selectSFCCapabilities`() operation.

5.6.2.2.3.4 Initiate a TFC

```
void initiateTFC(
      in t_SFEPNameList sfeps,
      in t_NFCName connection,
      in t_AdministrativeState state,
      out t_TFCName newTFC,
      out t_SFEPCorrelationList correlation,
      out t_ANfepList requiredNfeps
) raises (
      e_PartyDomainError,
      e_TFCError
);
```

This operation initiates a TFC. It can be used to initiate a point-to-point, bidirectional, or point(NFEP)-to-multipoint(SFEP) TFCs. A TFC branch is created for each SFEP named. The `connection` parameter identifies the NFC to be supported. The `state` gives the initial administrate state of all the branches in the TFC.

If successful, this operation returns a correlation identifier and associated SFEP for each branch. The correlation identifier is used in later operations on this interface and also across the TCon reference point to identify the TFC branch. The overall TFC itself is identified by the TFC name. It also returns a list of ANfeps. These ANfeps must support the transport requirements of the associated SFEPs. If these are not consistent the operation will fail. The ANfep descriptions give the protocol, QoS and address requirements of the ANfep. If the operation fails, an exception will be thrown.

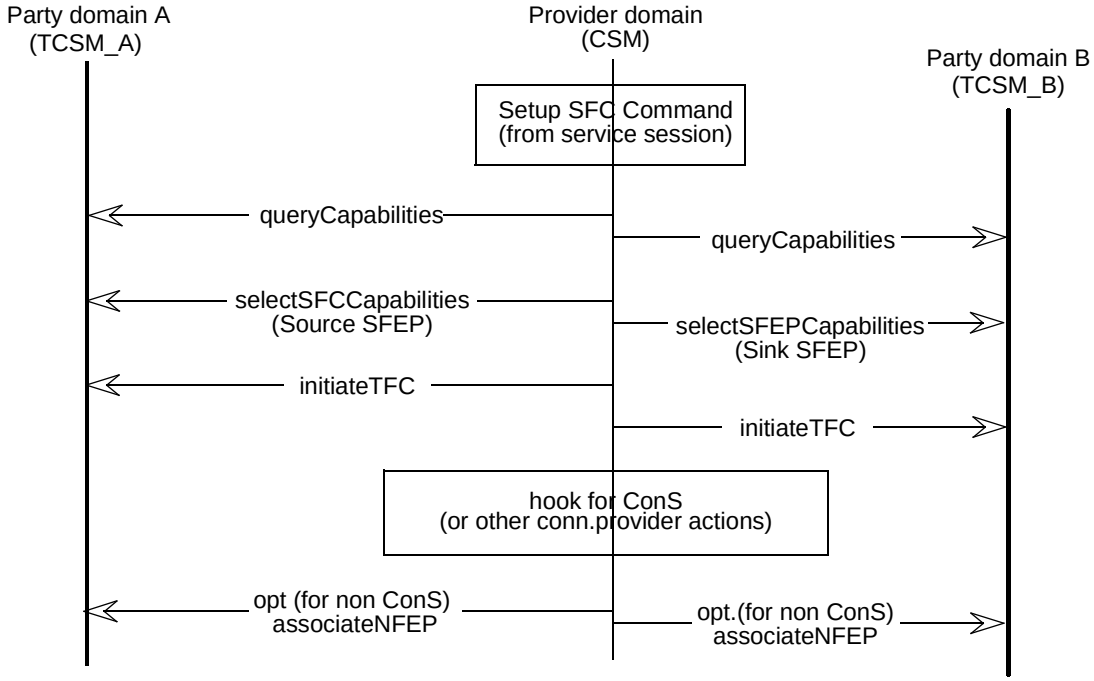


Figure 5-31. Setup a point to point SFC

Figure 5-31 shows the set up sequence for a simple point-to-point SFC. Initially, the provider domain component (the CSM) queries each TCSM associated with the SFC's SFEPs for the SFEPs' capabilities. If it cannot match the capability attributes, it selects the attributes it wants for the source

SFEP, and lets the TCSM select the common capabilities for the sink SFEP. It selects the sink SFEP's capabilities using those chosen by the source SFEP's TCSM. It then initiates a TFC for each terminal. Next, it determines the NFC using the returned ANFeps, and may make calls on the ConS interfaces to setup the NFC connection. If there ConS is not used, it may need to make an associate call to each terminal once the NFEP is determined to complete the TFC setup (and hence the SFC setup).

5.6.2.2.3.5 Initiate a TFC for multiple NFEPs

```
void initiateMultiNFEPTFC(
      in t_SFEPName sfep,
      in t_NFCNameList connections,
      in t_AdministrativeState state,
      out t_TFCName newTFC,
      out t_NFCCorrelationList correlation
) raises (
      e_PartyDomainError,
      e_TFCError
);
```

This operation initiates a TFC. It can be used to initiate a point-to-point, or point(SFEP)-to-multipoint(NEP) TFC. The `connections` parameter identifies the NFCs to be supported. Each NFC is associated with a TFC branch for which an NFEP must be selected. The `state` gives the initial administrate state of all the branches in the TFC. If successful, this operation returns a correlation identifier and associated NFC for each branch. It also returns a list of ANfeps with each NFC. If the operation fails, an exception will be thrown.

5.6.2.2.3.6 Add a TFC SFEP branch

```
void addTFCBranches(
      in t_TFCName aTFC,
      in t_SFEPNameList sfeps,
      in t_AdministrativeState state,
      out t_SFEPCorrelationList correlation
) raises (
      e_PartyDomainError,
      e_TFCError
);
```

This operation allows the CSM to request the addition of a branch to an existing TFC. The TFC is identified by the `aTFC` parameter. The new branches to be added are identified by the `sfeps` parameter. The initial state of these branches is determined by the `state` parameter. If successful, this operation returns a correlation identifier and associated SFEP for each branch. If the operation fails, an exception will be thrown. Add branch operations can be used to add a branch to an SFC or establish a new SFC over an existing NFC.

5.6.2.2.3.7 Add a TFC NFEP branch

```
void addMultiNFEPTFCBranches(
      in t_TFCName aTFC,
      in t_NFCNameList connections,
      in t_AdministrativeState state,
      out t_NFCCorrelationList correlation
) raises (
      e_PartyDomainError,
      e_TFCError
);
```

This operation allows the CSM to request the addition of a branch to an existing TFC. The TFC is identified by the `aTFC` parameter. Branches are identified by their associated NFCs. The new branches to be added are identified by the `connections` parameter. The initial state of these branches is determined by the `state` parameter. If successful, this operation returns a correlation identifier and associated NFC for each branch. It also returns a list of ANfeps with each NFC. If the operation fails, an exception will be thrown.

5.6.2.2.3.8 Delete TFC or its branches

```
void deleteTFCBranches(
      in t_TFCName aTFC,
      in boolean all,
      in t_CorrelationIdList branches
) raises (
      e_PartyDomainError,
      e_CSQueryError
);
```

This operation allows the communication session to delete a TFC or some of its branches. The TFC is identified by the `aTFC` parameter. The branches are identified by a list of correlation identifiers in the `branches` parameter. If the `all` flag is set true, all branches of the TFC are removed. Otherwise, only the identified branches are deleted. Once a TFC branch is deleted, the correlation identifier is invalid and the associated SFEPs and NFEPS are released. If the operation is successful, it returns. Otherwise an exception is raised. This operation is required to remove branches of an SFC, following similar steps to those in Section 5-33. See also Section 5.6.2.2.3.18.

5.6.2.2.3.9 Activate a TFC or its branches

```
void activateTFCBranches(
      in t_TFCName aTFC,
      in boolean all,
      in t_CorrelationIdList branches
) raises (
      e_PartyDomainError,
      e_CSQueryError
);
```

This operation allows the communication session to activate a TFC or some of its branches. The TFC is identified by the `aTFC` parameter. The branches are identified by a list of correlation identifiers in the `branches` parameter. If the `all` flag is set true, all branches of the TFC are activated. Otherwise, only the identified branches are activated. If the operation is successful, it returns. Otherwise an

exception is raised. This operation is required to activate branches of a SFC. Figure 5-33 shows a sequence of steps to activate a TFC. After receiving an activate SFC request, the CSM activates the associated NFCs and then makes `activateTFCBranches`() requests to each terminal.

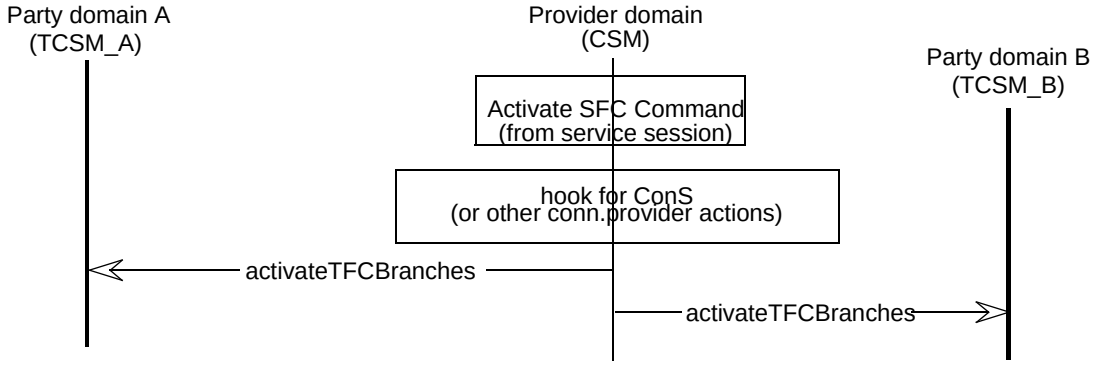


Figure 5-32. Activate a SFC.

5.6.2.2.3.10 Deactivate a TFC or its branches

```
void deactivateTFCBranches(
      in t_TFCName aTFC,
      in boolean all,
      in t_CorrelationIdList branches
) raises (
      e_PartyDomainError,
      e_CSQueryError
);
```

This operation allows the communication session to deactivate a TFC or some of its branches. The TFC is identified by the aTFC parameter. The branches are identified by a list of correlation identifiers. If the `all` flag is set true, all branches of the TFC are deactivated. Otherwise only the identified branches are deactivated. If the operation is successful, it returns. Otherwise an exception is raised. This operation is required to deactivate branches of an SFC. The steps required are similar to those in Figure 5-33 except, that the `deactivateTFC` request should precede the deactivate NFC request.

5.6.2.2.3.11 Update TFC or its branches

```
void updateTFCBranches(
      in t_TFCName aTFC,
      in boolean all,
      in t_CorrelationIdList branches,
      in boolean noDisruption,
      out TINACommSCommonTypes::t_NFEPUpdate reqChange,
      out t_ANfepList mods
) raises (
      e_PartyDomainError,
      e_TFCError
);
```

This operation allows the communication session to update a TFC or TFC branches after a change of SFEP capabilities, where the branches must all be associated with the same NFEP. If all branches are affected, the `all` flag is set true. Otherwise, the `branches` parameter lists affected branches by their correlation identifier.The `noDisruption` flag tells the TCSM if the TFC can be disrupted or not. If it is not possible to comply with this condition, an exception is thrown.

If the operation is successful, it returns the type of change required to the NFEP (none, modify current NFEP, or new NFEP required). It also returns a ANfep list that describes modifications to the existing NFEP or suggests possible new ANfeps. If there is an error, an exception is raised.

This operation supports the modification of SFC branches' quality requirements. Figure 5-33 shows an example of modifying SFC quality. After receiving the modify command, the CSM queries each terminal to determine currently available capabilities. (These change as capabilities are released or reserved for the SFEPs.) It then selects the new capabilities for the source and sink respectively. Next it calls an `updateTFC()` operation for each terminal. The terminals respond with update required to the supporting NFEPs. These may be modified or require the selection of new NFEPs. If the former, the supporting NFC can be modified. If the latter, the existing NFC may need to be replaced (in whole or part), and new NFEPs associated with the TFCs.

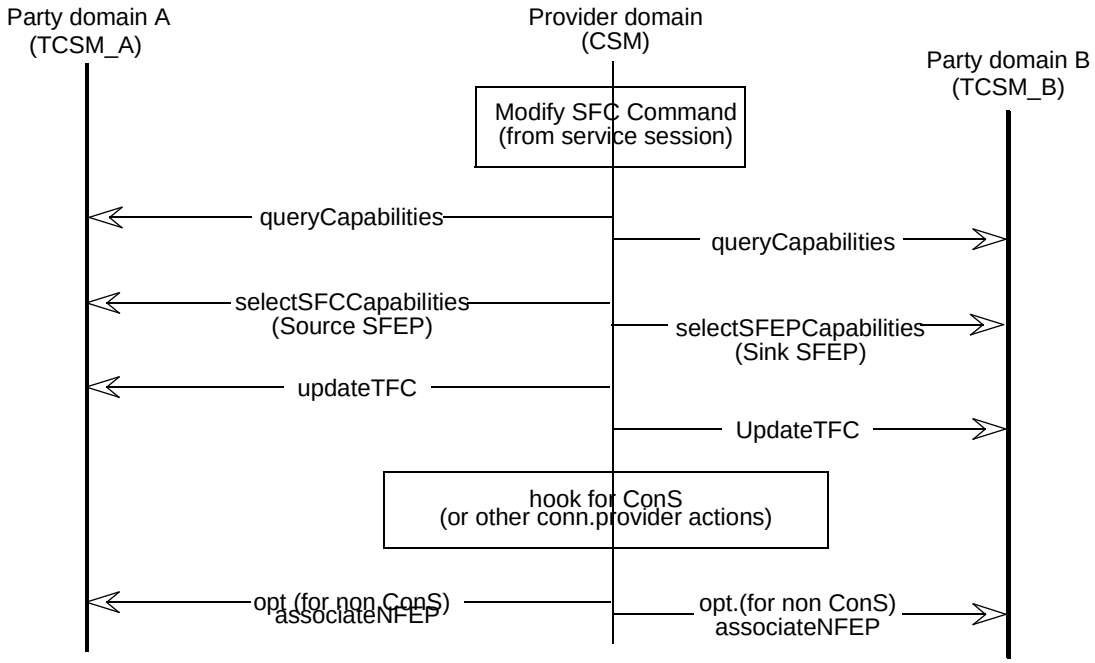


Figure 5-33. Modify an SFC's quality requirements.

5.6.2.2.3.12 Update TFC or its branches

```
void updateMultiNFEPTFCBranches(
      in t_TFCName aTFC,
      in boolean all,
      in t_CorrelationIdList branches,
      in boolean noDisruption,
      out TINACommSCommonTypes::t_BranchUpdateList mods
) raises (
      e_PartyDomainError,
      e_TFCError
);
```

This operation allows the communication session to update a TFC or TFC branch after a change of SFEP capabilities, where the branches are terminated with different NFEPs. If all branches are affected, the `all` flag is set true. Otherwise, the `branches` parameter lists affected branches by correlation identifiers. The `noDisruption` flag tells the TCSM if the TFC can be disrupted or not. If it is not possible to comply with this condition, an exception is thrown.

If the operation is successful, it returns the a list of updates for each branch in the `mods` parameter. This parameter includes type of change required to the NFEP and an ANfep list that describes modifications to the existing NFEP or suggests possible new ANfeps. If there is an error, an exception is raised. This operation supports the modification of SFC branches.

5.6.2.2.3.13 Resolve SFEP capabilities to initiate TFC

```
void resolveSFEPforTFC(
      in t_SFEPSelectList sfeps,
      in t_NFCName connection,
      in t_AdministrativeState state,
      out t_TFCName newTFC,
      out t_SFEPCorrelationList correlation,
      out t_ANfepList requiredNfeps
) raises (
      e_PartyDomainError,
      e_TFCError,
      e_CapabilityError
);
```

This operation allows the communication session to set capabilities for one or more SFEPs and initiate the associated TFC. It assumes that the common SFEP capabilities are already known. The `sfeps` parameter lists SFEPs with their desired capabilities. These are used to select capabilities for each SFEP before initiating the TFC. In all other respects, this operation acts like a `initiateTFC`() operation. Desired transport requirements, such as protocols and QoS requirements should be returned in the ANfep descriptions. This operation is provided to improve the efficiency of TFC setup by combining two sequential operations. Further efficiency savings can be made if the SFEPs passed to the communication session already contain capability information. In this case, only a single operation for each terminal is required at the communication level to setup an SFC.

5.6.2.2.3.14 Resolve SFEP capabilities to initiate multiple NFEP TFC

```
void resolveSFEPforMultiNFEPTFC(
      in t_SFEPSelect sfep,
      in t_NFCNameList connections,
      in t_AdministrativeState state,
      in t_SinkAttributes sfcCaps,
      out t_TFCName newTFC,
      out t_CapabilityList commonCaps,
      out t_NFCCorrelationList correlation
) raises (
      e_PartyDomainError,
      e_TFCError,
      e_CapabilityError
);
```

This operation allows the communication session to set capabilities for an SFEP and initiate the associated TFC for one or more associated NFCs. It assumes that the common SFEP capabilities are not known and can be determined by the TCSM. The capabilities of the local SFEP as well as other SFEPs associated with the SFC are passed. The local capabilities indicate those capabilities preferred by the CSM. The TCSM ensures a common set of capabilities exist and that it can setup the requested local capabilities.

Once the capabilities are selected, it initiates the local TFC. This may be over multiple NFEPs as indicated by the NFCs. From this point it proceeds as for a `initiateMultiNFEPTFC()` operation, except that, as well as returning the branch correlation and ANFEP information, it also returns the common capabilities. Desired transport requirements for the local SFEP, such as protocols and QoS requirements, should be returned in the ANfep descriptions. As before, this operation is provided to improve the efficiency of TFC setup by combining two sequential operations.

5.6.2.2.3.15 Resolve SFEP capabilities to initiate bidirectional TFC

```
void resolveSFEPforBiTFC(
      in t_SFEPSelectList sfeps,
      in t_NFCName connection,
      in t_AdministrativeState state,
      in t_SinkAttributes sfcInCaps,
      in t_SinkAttributes sfcOutCaps,
      out t_TFCName newTFC,
      out t_CapabilityList commonInCaps,
      out t_CapabilityList commonOutCaps,
      out t_SFEPCorrelationList correlation,
      out t_ANfepList requiredNfeps
) raises (
      e_PartyDomainError,
      e_TFCError,
      e_CapabilityError
);
```

This operation allows the communication session to set capabilities for two SFEPs associated with a bidirectional TFC. It assumes that the common SFEP capabilities are not known in either direction and can be determined by the TCSM. The capabilities of the local SFEPs as well as other SFEPs associated with the two SFCs (one in each direction as SFCs are not bidirectional) are passed. The

local capabilities indicate those capabilities preferred by the CSM for the input and output SFEP respectively. The TCSM ensures a common set of capabilities exist in both direction and that it can setup the requested local capabilities for both SFEPs.

Once the capabilities are selected, it initiates the local TFC for the two TFC branches. From this point it proceeds as for a `initiateTFC()` operation, except that as well as returning the branch correlation and ANFEP information it also returns the two sets of common capabilities. Desired transport requirements for the local SFEPs, such as protocols and QoS requirements, should be returned in the ANfep descriptions. As before, this operation is provided to improve the efficiency of TFC setup by combining two sequential operations.

5.6.2.2.3.16 Resolve SFEP capabilities and add TFC branches

```
void resolveSFEPforTFCBranches(
      in t_TFCName aTFC,
      in t_SFEPSelectList sfeps,
      in t_AdministrativeState state,
      out t_SFEPCorrelationList correlation
) raises (
      e_PartyDomainError,
      e_TFCError
);
```

This operation allows the CSM to request the selection of SFEP capabilities and the initiation of additional TFC branches for each SFEP. If successful, this operation returns a correlation identifier for each branch with the associated SFEP. If the operation fails, an exception will be thrown.

5.6.2.2.3.17 Associate NFEP with TFC branches operation

```
void associateNFEP(
      in t_CorrelationIdList branches,
      in t_TinaName aNFEP
) raises (
      e_PartyDomainError,
      e_CSNFEPError
);
```

This operation allows the communication session to associate a NFEP with TFC branches identified by the correlation identifiers, completing the TFC branch set up. Branches may be completed at the TCon level, assuming a similar operation between the TLA and TCSM. This operation is included to allow use of non ConS/TCon providers. The `aNFEP` parameter identifies the NFEP that has been selected by the connectivity layer. If the operation is successful it returns and completes the TFC. Otherwise, it raises an exception. This operation is optionally used to setup SFC branches. It may be used in conjunction with the `addTFCBranches()` operation to setup a SFC over an existing NFC.

5.6.2.2.3.18 Remove NFEP from TFC branches operation

```
void removeNFEP(
      in t_TinaName aNFEP,
      in boolean all,
      in t_CorrelationId branches
) raises (
      e_PartyDomainError,
      e_CSNFEPError
);
```

This operation allows the communication session to remove a NFEP from TFC branches. The branches from which it is to be removed are identified by correlation identifiers. This automatically deactivates the TFC branch. This can be used to start deleting an SFC or SFC branch. It could also be used to swap an NFEP from connection with one SFEP to connection with another. This operation does not remove the entire TFC branch (though it is no longer operational). Rather it prepares the branch for a setup change. If the operation is successful it returns. Otherwise, it raises an exception.

This operation is optionally used to delete SFC branches or change the NFEP with which TFC branches are associated (and hence change the SFC to NFC mapping). Figure 5-33 shows the steps needed to remove a SFC. After receiving the command, the CSM may optionally remove the NFEPs from the TFCs, deactivating the TFCs. It then call the appropriate NFC deletion operations over the ConS reference point. Finally it deletes the TFCs. To change SFC to NFC mapping, the TCSM would call the `removeNFEP`() operation followed by the `associateNFEP`() operation.

5.6.2.2.4. Unsupported functionality

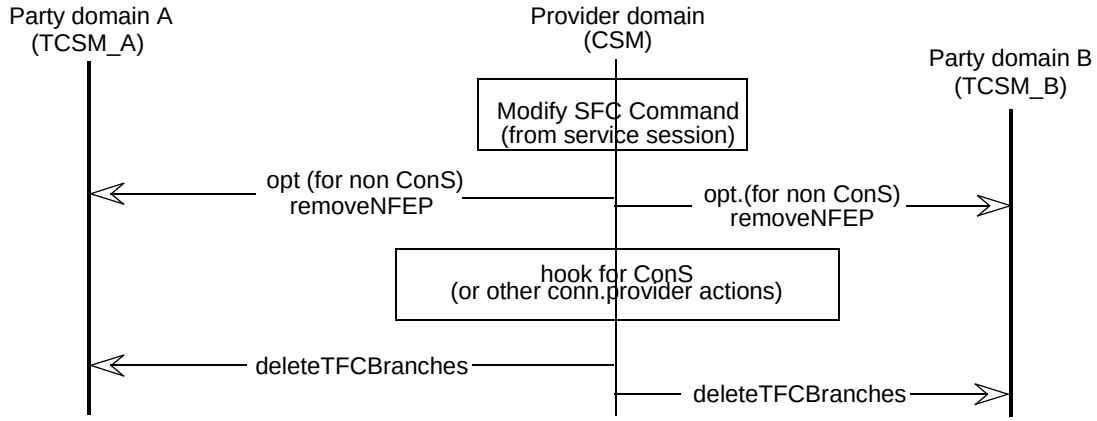


Figure 5-34. Deleting a SFC.

There is still work on going for the communication session. This may result in some additions or modifications to the functionality of this interface or the addition of further interfaces.

- Possible modifications: There may be some modifications in the QoS parameter definitions and this may affect capability selection and query operations.
- Additional functions on the i_TerminalFlowControl interface (or derived interface):
  - Additional query functionality:
    - Query bidirectional support;

  - Query multiplexing support (SFEP to multiple NFCs);
  - Query multiplexing support (NFC to multiple SFEPs;
  - Query demultiplexing support for an SFEP. (i.e. combining flows from multiple NFCs - this would also mean supporting multipoint-to-point TFC topologies.)

- Additional capability selection functionality: Currently capabilities can be modified by performing new selection operations. It may be possible to modify an existing capability set by adding, removing or modifying capabilities within the set. Extensions include:
  - Add capability;
  - Modify capability;
  - Delete capability.
- Add more TFC initiation and branch addition operations: e.g. point SFEP to multipoint SFEPs or point SFEP to multipoint NFEPs/SFEPs.
- Add more combined operations: A wider range of combined operations could be useful for TFC initiation. Also, operations that could handle multiple TFC setups could be useful.

• Additional interfaces:

  Additional interfaces, particularly to support TCSM to CSM interactions are desirable.

- Notification interface for CSM (base on i_GeneralStreamInfo interface)
- Notification control interface: to allow a CSM (or other provider domain components) to direct notifications to itself or some other component.

6. Document Stability

The stability of this document is really measured in terms of the stability of the specifications it defines. This section indicates the relative stability of the parts that comprise Ret-RP.

The stability of the access and usage parts of Ret-RP are discussed separately in the following sections. However, there a number of issues which are common to both parts, and are discussed below.

This section of the document was last updated on 22 December 1997. Obviously, the projects detailed below are continuing to implement prototypes based upon Ret-RP, and will make more comments on aspects currently not implemented. Also the IDL specifications will be compiled using different platforms and language mappings. The purpose of highlighting the date here is that the document may have new versions, with a new date for the whole document.However this section has only been updated according to the date above, and so the implementation projects may have made significant progress over that detailed below. Hopefully as problems with the specification are updated in the rest of the document, they will be updates here also.

It is hoped that readers that find problems with the specification send comments to the Editors to help in updating the specification.

IDL specifications:

The Ret-RP specifications are defined in the OMG Interface Definition Language (IDL). These specifications have been compiled using IDL compilers to check their syntax, and ensure that they can be used with current commercial IDL compilers, to build industry-strength consumer and retailer components.

Table 6-1 defines the IDL compilers that have been used to compile the Ret-RP IDL.

Table 6-1. CORBA Platforms used to test syntax of IDL specifications

Product	Version	Supplier	Access Part	Usage Part TINA Service Session Model	Usage Part TINA Comm Session Model
Orbix multi-threaded	v2.1	IONA Technologies	Compiled IDL and stubs	Compiled IDL and stubs	Compiled IDL Stubs not compiled.
OrbixWeb	v2.0.1 patch level 5 v3.2 beta release 2	IONA Technologies	Compiled IDL and stubs	Compiled IDL and stubs	Compiled IDL and stubs
Visibroker for Java	v2.5 v3.0	Visigenics	Compiled IDL and stubs	Not Compiled	Not Compiled
NEO	2.0	Sun Microsystems	Compiled IDL and stubs	Compiled IDL Stubs not compiled.	Compiled IDL Stubs not compiled.

The IDL specifications have been compiled in order to check their syntax. A number of syntax problems, such as name clashes and unusable types, have been identified and removed from the specifications. The current specifications pass the syntax checking of these compilers, and generate stub code to be included in components implementing the interface.

For some parts of Ret-RP, the stubs have also been compiled using C++ and Java language mappings for the platforms indicated. This gives a good indication that the specification will compile on other platforms, and using other language mappings. However, they have only been tested on the platforms specified.(Please report any problems, or error found when compiling or implementing the Ret-RP specification to the editors of this document.)

Implementations:

The following is the status of implementation projects, which include parts of the Ret-RP. The information below is correct as of 1st December 97. Many of the projects are continuing development, and will, by the time you read this, have implemented more of Ret-RP, and may have insightful comments on the specifications.

Table 6-2. Implementation Projects and their progress.

Project	Participants	Implementations
Eurescom P715 contact: P715@research.kpn.com	British Telecommunications plc (BT) Deutsche Telecom AG (DT) France Télécom (FT) FINNET Group Royal PTT Nederland NV (KPN) Telecom Éireann	A service platform based at each partner site, interconnected by ISDN. Addresses Ret, ConS, TCon RPs implementing selected parts of each RP. Also addresses RtR and 3PTY RPs.
ACTS VITAL		Implementations followed roughly early versions of Ret-RP, but with some significant approach in isolated areas. Implementation for 1998 will follow Ret-RP.
Global One TINA Technical Trial	Global One partners, (Sprint, France Télécom, Deutsche Telecom AG)	ORBs: PowerBroker, VisiBroker, HP-Distributed Smalltalk and Orbix. Implemented: selected operations of access part. No implementation of usage part.

6.1 Stability of Access Part Specifications

The Access Part is the most stable and longest lived part of the Ret-RP. In general, the access part is stable. Where comments have been received on a particular operation, they are summarised below. Some operations are labelled as draft, and so updates to their definitions are possible.

Table 6-3. Stability of Access Part interfaces

Interface	Operation	Updates
i_ConsumerInitial	requestAccess()	Draft Definition: comments outstanding: Section 6.1.1, "i_ConsumerInitial"
	inviteUserOutsideAccessSession() cancelInviteUserOutsideAccessSession()	Draft Definition: comments outstanding: see Section 6.1.4, "Invitations".
i_ConsumerAccess	getInterfaceTypes() getInterface() getInterfaces()	Stable.
	cancelAccessSession()	Draft Definition: no comments, but no implementations at present, see Section 6.1.2, "i_ConsumerAccess"

Table 6-3. Stability of Access Part interfaces

Interface	Operation	Updates
i_ConsumerInvite	inviteUser() cancelInviteUser()	Comments outstanding: see Section 6.1.4, "Invitations".
i_ConsumerTerminal	getTerminalInfo()	Draft Definition: comments for operations allowing more flexibility in querying the consumer domain, and the sort of information that should be passed using this interface.
i_Consumer AccessSessionInfo	All operations.	Comment on use of oneway, see Section 6.1.6, "Info operations".
i_Consumer SessionInfo	All operations.	Comment on use of oneway, see Section 6.1.6, "Info operations".
i_RetailerInitial	requestNamedAccess()	Stable. Implemented.
	requestAnonymousAccess()	Stable, but no implementations at present.
i_RetailerAuthenticate	getAuthenticationMethods() authenticate() continueAuthentication()	Stable, but no complete implementations at present.
i_RetailerAccess	Operations inherited from i_ProviderAccessInterfaces, i_ProviderAccessGetInterfaces and i_ProviderAccessRegisterInterfaces.	Stable.
i_RetailerNamedAccess	All operations, except below:	Stable
	listSubscribedServices()	Comments outstanding: see Section 6.1.8, "Subscribed Services".
	listSessionAnnouncements() joinSessionWithAnnouncement()	Comments outstanding: see Section 6.1.5, "Announcements".
	replyToInvitation()	Comments outstanding: see Section 6.1.4, "Invitations".
i_RetailerAnonAccess	All operations inherited from i_RetailerAccess	Draft Definition: see Section 6.1.7, "Anonymous Users".
i_DiscoverServicesIterator	All operations.	Stable: no comments, but no implementations at present

6.1.1 i_ConsumerInitial

The purpose of the i_ConsumerInitial interface is to provide an initial contact point for the retailer wishing to contact the consumer. All of the operations defined on this interface are draft at present, and therefore subject to change. (The requestAccess() operation is discussed below. Other operations are discussed in Section 6.1.4, "Invitations"

The requestAccess() operation allows the retailer to request that an access session is established between the consumer and the retailer. Comments from VITAL have suggested allowing some sort of invitation to be passed to the consumer as part of this request. This ‘invitation’ could describe why

the access session was being requested, and include an invitee identifier to indicate the user with which the access session is requested. Additional reply codes would also be required, e.g. UNKNOWN for an unknown user.

This sort of invitation would be different to a session invitation, as a session invitation can already be delivered to this interface.

The Global One TT have not made specific comments on this interface. However they have made a suggestion with regard to invitations that is applicable to both invitations sent outside access sessions and retailers requesting consumers to establish an access session. They suggest that an interface reference is included with the invitation (either to join a session, or establish an access session.) The interface is used by the consumer domain to reply to the invitation, join a session, or request the access session. See Section 6.1.4, "Invitations" for more details.

6.1.2 i_ConsumerAccess

Operations which provide the retailer domain with access to the consumer domain interfaces are stable, with no changes expected.

The `cancelAccessSession`() operation is draft only. It allows the retailer to end an access session with the consumer. The retailer can use this operation to terminate an access session without the consumer's permission. Comments are invited on this operation. No specific problems have been identified as yet though, but updates are possible.

6.1.3 i_ConsumerTerminal

This interface is used by the retailer domain to retrieve information about the capabilities of the terminal in the consumer domain. Currently, it defines only a a single operation (`getTerminalInfo`()) which is used to retrieve all the information abut the consumer domain. This is not really practical. It is envisaged that a 'query' based set of operations will be defined for retrieving this information, allowing the retailer to retrieve only the information that is necessary. Comments or suggestions on this approach will be gratefully received.

6.1.4 Invitations

Invitations are sent from the retailer to the consumer to invite a specific user to join a service session.

The specification for invitations (`t_SessionInvitation` and `t_InvitationReply` parameters) are defined according to the Internet Engineering Task Force working group MMUSIC, (Multimedia Multiparty Session Control) proposal for draft standard 'Session Initiation Protocol' (SIP). The TINA specification allow invitations to 'internet' sessions to be delivered using the Ret-RP, and potentially allow TINA invitations to be delivered using SIP. However, SIP is still an evolving standard, and changes to the TINA specification for invitations may need to be updated to allow interoperability between the specifications. Also, an emerging internet standard for video conferencing (H.323) also defines an invitation specification, and may supercede SIP. TINA may need to choose between these protocols if it wishes to retain interoperability of invitations.

Global One TTT have suggested that the invitation include a reference to an interface on which to reply to the invitation and/or join the session. This interface could be separate from the `i_RetailerAccess` interface that is currently used to reply to invitations, and join sessions. This suggestion comes from a specific implementation problem encountered due to the Global One TTT. The suggestion is not entirely in keeping with the TINA Service Architecture, however it is not an unacceptable suggestion, but does require further discussion.

6.1.5 Announcements

Announcements are made by sessions to ‘publicise’ the session. They are broadcast to ‘groups’ of users, which can use Ret-RP to retrieve a list of announcements which match specific properties.

The specification for announcements are draft at present. It contains a list of announcement properties, but no specifications for announcement properties have been defined. This means that announcements are currently retailer specific.

TINA will need to define a structure for announcements which allows interoperability between retailers. Also, TINA may wish to allow interoperability with other session announcement mechanisms, such as the Internet Engineering Task Force working group MMUSIC, (Multimedia Multiparty Session Control) proposal for draft standard ‘Session Announcement Protocol’ (SAP)

6.1.6 Info operations

Info operations are sent to party domains, when the service session has taken an action. They are sent to inform the party domain of the action, (which may need to update some internal state, or inform the user). The operations never allow the party domain to return a result.

All Info operations are specified as oneway. This is because the operations are for information only, they never allow the party domain to return a result, and so it is not necessary for the service session to block waiting for the party domain to return.

However, oneways may have an unfortunate side effect: the ORB does not have to deliver the oneway. There is still some debate as to whether the ORB must deliver the oneway with guaranteed delivery, or not[1]. A system exception can be caught if the oneway is not delivered.

The purpose of specifying Info’s as oneway is to ensure that the party domain does not ‘block’ the service session, by waiting instead of returning from the Info operation immediately. However, it is not clear that using oneways is the best way to specify this in the IDL. There are other mechanisms that the session can use to avoid being blocked when it invokes an Info operation.

6.1.7 Anonymous Users

Anonymous users are users that do not wish to reveal their identity to the retailer, and establish a ‘log-term’ relationship with the retailer. Anonymous users may wish only make use of free retailer services; or pay for charged services on a use-by use basis.

Currently anonymous users are not fully supported by Ret-RP. Ret-RP does define an operation (`requestAnonymousAccess`() on `i_RetailerInitial`) to an establish an access session as an anonymous user, and it provides an interface (`i_RetailerAnonAccess`) to be used during the access session. However, `i_RetailerAnonAccess` currently does not include any operations.

No operations have been defined on `i_RetailerAnonAccess`, because it was unclear as to which operations would be usable by anonymous users. An inheritance hierarchy has been defined that will allow operations to be shared between `i_RetailerAnonAccess` and `i_RetailerAnonAccess` for named users. Operations for anonymous users should be defined for the next version of Ret-RP.

1. The Generic Interoperation Protocol (GIOP) prescribed for interactions between ORBs forces them to use a reliable network protocol that would ensure delivery of oneways. However ORBs do not have to use a GIOP-compliant protocol for intra-ORB messages. All current ORBs do

6.1.8 Subscribed Services

The consumer requests from the retailer a list of their subscribed services, in order to determine which services can be started, and what applications in the consumer domain may be used with the service. Currently, there is only a single operation (`listSubscribedServices()` on the `i_RetailerNamedAccess` interface) to access this information. This operation can return a lot of information on each service (`t_ServiceInfo`), with the consumer potentially subscribed to many services.

The Global One TTT have suggested added another operation to `i_RetailerNamedAccess` interface which provides supplementary information on each service. `listSubscribedServices()` would indicate if there is supplementary information available for each service. It is likely that an operation (`getServiceInfo`()) will be added to the next version of Ret-RP.

6.1.9 Synchronous versus Asynchronous interactions

The TINA Service Session Model was designed with a simple model of synchronous interactions. A Request (Req operations) is invoked by a party, on the provider domain session. The request does not return immediately. The party domain must wait for the session to do some processing before the request will return. The session may make Indications (Ind), Execution (Exe) and possibly Information (Info) operations before the Req returns to the party.

This synchronous model for interactions is very simple to understand, and capture in event trace diagrams. It is also simple to program components to follow this model. However, this model does imply that the party domain components must block when making a request. This is not necessarily true, as it is upto the programmer to decide if they wish the party components to block, or to program using threads, or the CORBA Dynamic Invocation Interface. However, the simpliest way to program this is for the party domain components to block.

Other models have been suggested that follow an asynchronous model for interactions. There are two ways to model this: as pseudo-asynchronous, and true asynchronous.

In the former, the party domain would make a normal synchronous request, but the provider domain session would return immediately, before processing the request. The session would then process the request and then invoke an operation on a party domain interface to return the result of the request. This is a pseudo-asynchronous mode, as the asynchronous model is comprised of many synchronous requests, which the receiving component ‘promises’ to reply both immediately, with no result, and later with a real result. The problem with this approach is ensuring that the receiving components don’t force the requester to block, by not returning immediately.

In the latter, true asynchronous uses asnychronous calls to send the requests and replies. This means that the receiver cannot block the requester by not replying immediately.

It is likely that a new Session Model, using pseudo-asynchronous interactions, will be defined as part of the continuing definitions for Ret-RP. Some discussion of the advantages of this is given in Section 5.5.9.2, "Asynchronous and synchronous responses".

6.1.10 Implementation Problems

6.1.10.1 Problems with 'Any'

A number of projects and individuals have complained about the use of the IDL type 'any'. Type any is used to indicate that a value of an arbitrary type can be passed as a parameter, or return value. Type any is generally used in the type t_Property, as part of a name and value pair, (the value being of type any). This is used to allow Ret-RP to be extended by retailer-specific properties, or to be extended as part of another version of Ret-RP. The purpose of using any is to allow this extension without changing the IDL of existing operations. As part of t_Property, new value types can be defined, and a name assigned to each new type. Each new name-value pair can now be carried by any operation that can carry a t_Property. Operations receiving a t_Property can use the name to determine the type of the value, or even use a typecode carried in the any, and look this up in the interface repository to find the value type. This was the method chosen for extension of the reference point.

However, there are some problems associated with any.

Firstly, programmers tend to dislike them, as it is sometimes necessary to use the interface repository to discover the type of the any. In general, this should not be necessary for the value in t_Property, as the type of the value is defined by the t_Property name.

Secondly, Global One TTT found problems in passing any values across multiple ORBs. The any was correctly passed between clients and servers on the same ORB, but was unreadable when they were on different ORBs. It is not clear if this is a problem between 2 specific ORBs, or whether any's generally cannot be passed across multiple ORBs. More information of this problem would be gratefully received by the Editors of Ret-RP.

6.2 Stability of Usage Part Specifications

The usage part of Ret-RP can be split into: the TINA Service Session Model, and TINA Communication Session Model; and further divided into the feature sets of each model.

In general, the usage part of Ret-RP has been the subject of fewer implementations than the access part, and consequently, is more likely to require minor alterations, or contain minor errors that would be discovered through prototyping. In any case, the IDL and stubs have been compiled according to Table 6-1.

6.2.1 TINA Service Session Model

For the usage part, the TINA Service Session Model has been the subject of most implementation, and external interest. The Multiparty related feature sets (MultipartyFS, MultipartyIndFS, ControlSRFS), in particular, have been studied by the TINA Auxiliary project VITAL, who have implemented a prototype that closely corresponds to the definition for Ret-RP.

Other feature sets have been studied as well: the control and voting feature sets (ControlSRFS, VotingFS), and the Participant-oriented Stream Binding feature sets (ParticipantSBFS, ParticipantSBIndFS). Less VITAL partners were involved in these more sophisticated testings, but exhaustive tests were successfully performed. It must be noted that VITAL actually implemented a simplified variant of the Participant-oriented Stream Binding feature sets, essentially based on the previous Ret version, but using the new data types. The simplified version supports similar functionality but is easier to implement, and less prone to interpretation misunderstandings.

Table 6-4. Stability of TINA Session Model feature sets.

Feature Set	Interfaces	Updates
BasicFS	i_ProviderBasicReq	Stable, but suspendSessionReq() often not implemented.
BasicExtFS	i_PartyBasicExtReq	Stable, but no implementations at present.
MultipartyFS	i_PartyMultipartyExe i_PartyMultipartyInfo i_ProviderMultipartyReq	Stable, but suspendPartyReq(), and suspendMyParticipationReq() often not implemented
MultipartyIndFS	i_PartyMultipartyInd	Stable.
VotingFS	i_PartyVotingInfo i_ProviderVotingReq	Stable.
ControlSRFS	i_PartyControlSRInd i_PartyControlSRInfo i_ProviderControlSRReq	Stable, but the WritePermission often not implemented.
ParticipantSBFS	i_PartyPaSBExe i_PartyPaSBInfo (i_ConnInfo) i_ProviderPaSBReq	Two types of implementations at present: one implementation of this version, and an implementation of a simplified variant, used in VITAL.
ParticipantSBIndFS	i_PartyPaSBInd	Same as for ParticipantSBFS.

6.2.2 TINA Communication Session Model

The TINA Communication Session Model is much newer than the rest of Ret-RP. Consequently, the specifications have undergone a shorter period of review, and no implementation at present. As such this session model may require significant modification in light of implementations.

However, the basic features defined by the TINA Communication Session Model are relatively small in comparison to the other session model, and so it is more likely to be self-consistent than the TINA Service Session Model was initially.

6.2.2.1 TINA Communication Session Model additional functionality

There is still work on going for the TINA communication session model. This may result in some additions or modifications to the functionality of current interfaces or the addition of further interfaces and feature sets.

- Possible modifications: There may be some modifications in the QoS parameter definitions and this may affect capability selection and query operations.
- Additional functions on the `i_TerminalFlowControl` interface (or derived interface):
  - Additional query functionality:
    - Query bidirectional support;
    - Query multiplexing support (SFEP to multiple NFCs);
    - Query multiplexing support (NFC to multiple SFEPs;

    - Query demultiplexing support for an SFEP. (i.e. combining flows from multiple NFCs - this would also mean supporting multipoint-to-point TFC topologies.)

  - Additional capability selection functionality: Currently capabilities can be modified by performing new selection operations. It may be possible to modify an existing capability set by adding, removing or modifying capabilities within the set. Extensions include:
    - Add capability;
    - Modify capability;
    - Delete capability.
  - Add more TFC initiation and branch addition operations: e.g. point SFEP to multipoint SFEPs or point SFEP to multipoint NFEPs/SFEPs.
  - Add more combined operations: A wider range of combined operations could be useful for TFC initiation. Also, operations that could handle multiple TFC setups could be useful.

- Additional interfaces:

  Additional interfaces, particularly to support TCSM to CSM interactions are desirable.

  - Notification interface for CSM (base on `i_GeneralStreamInfo` interface)
  - Notification control interface: to allow a CSM (or other provider domain components) to direct notifications to itself or some other component.

7. References

7.1 TINA Baselines

[1] *TINA reference points*, Document No EN_TCJ.030_3.1_96, version 3.1, June 1996.

[2] *TINA Glossary of Terms,* Version 2.1, TINA-C, Jan. 1997; public.
File: /u/tinac/97/integration/docs/glossary/v2.1/GLOSSARY.ps

[3] *Information Modeling Concepts,* TINA-C, April 1995; public.
File:/u/tinac/94p2/viewable/info.ps

[4] *Computational Modelling Concepts,* Version 3.2, TINA-C, May1996; TINA-C internal.
File:/u/tinac/96/dpe/docs/computational_model/v3.2/cmc.ps

[5] *Service Architecture, Definition of Service Architecture*, TINA-C: Version 5.0, 16 June 1997.
File:/u/tinac/97/services/docs/sa/sa5.0/final/main.ps

[6] *Service Architecture, Annex*, TINA-C, 1997: Version 5.0, 16 June 1997.
File:/u/tinac/97/services/docs/sa/sa5.0/final/annex.ps

[7] *TINA Business Model and Reference Points,* Version 4.0 , TINA-C, May1997; public.
File:/u/tinac/97/integration/viewable/bm_rp.ps

[8] *Request for Refinements and Solutions, The Ret Reference Point*, Version 2.0, 9 August 1996.
File: /u/tinac/96/integration/rfrs/RFR-96-01/rfrs_ret.ps

[9] *TINA Network Resource Architecture,* Version 3.0, TINA-C, Febr. 1997; public.
File: /u/tinac/resources/viewable/nra_v3.0.ps.

7.2 Responses to RFR/S for Ret-RP

[10] TINA-C Core Team, *Response to a request for Refinements and Solutions, The Ret Reference Point*, Version 1.1, Contact person: Martin Yates. Email: mjyates@tinac.com

[11] Alcatel response to RFR/S for Ret-RP, Authors: VITAL project, *Contribution to the TINA Ret-RP*, Report Code: AC003/SES/WP2/DR/R/RET/X1, Date:28.10.96, Contact person: Hans Vanderstraeten. Email: hvds@rc.bel.alcatel.be, (alt. Marcel Mampey, mmam@rc.bel.alcatel.be)

[12] BT, *Response to a request for Refinements and Solutions, The Ret Reference Point*, Version 1.0, Contact person: Martin Ellis. Email: mart@drake.bt.co.uk

[13] Ericsson, *Response to a request for Refinements and Solutions, The Ret Reference Point*, Version 1.0, Contact person: Jim Holehouse. Email: etjuse@etlxdmx.ericsson.se

[14] France Télécom CNET, *Response to a request for Refinements and Solutions, The Ret Reference Point*, Version 1.0, Contact person: Fabrice Dupuy. Email: dupuy@lannion.cnet.fr

[15] Telia Research AB, Sweden, *Response to a request for Refinements and Solutions, The Ret Reference Point*, Version 1.0, Contact person: Lennart Hedenström. Email: Lennart.R.Hedenstrom@telia.se

7.3 Other documents

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

[17] Object Management Group (OMG), *The Common Object Request Broker Architecture and Specification*, Ver2.0, July 1995.

[18] M. Handley, H. Schulzrinne, E. Schooler, *SIP: Session Initiation Protocol*, Internet Engineering Task Force, INTERNET-DRAFT, 26 March 97. draft-ieft-mmusic-sip-02.ps.

[19] *Service Component Specification, Computational Model and Dynamics.* Version 1.0b (draft 0.1), september 8th 1997. (Final version in Oct 97). TINA-C Core Team.
Available at: /u/tinac/97/services/docs/scs/compmod/draft0.1/final/comp.ps

[20] *Network Resource Component Specification.* Version 2.1, august 18th 1997. (Final version in Oct 97). TINA-C Core Team.
Available at: /u/tinac/97/resources/network/docs/ncs/v2.1/ncs.ps

8. Acronyms

RFR/SRequest for Refinement/Solution

RPReference Point

Annex A: Conventions for Reference Points

This annex describes the module naming conventions for IDL modules, and implicitly the module structure. Also, the complete list with defined modules and contained interfaces is presented for reference. Last, some recorded problems with IDL compilation are listed.

A.1 Module Naming Conventions

This part described the naming and structuring conventions used in the definition of the IDL for Ret RP. These conventions do not only apply to Ret, but can also be a guideline for specifications for other reference points.

The naming for the modules, reflecting the separation described in this document, is as follows:

<modulename> := TINA<common> |
TINA<sessionrole><part> |
TINA<reference point><domain><part>

The first set of <modulename> applies to the common parts. The second and third set apply to business role parts. Essentially the third set inherits from the second set and refers to reference point specific business roles.

<common> := CommonTypes |
AccessCommonTypes |
UsageCommonTypes |
StreamCommonTypes
<feature set>Types

<reference point> := Ret |
<other acronyms, e.g. Tcon, Cons, RtR>

<sessionrole> := User | (for access part)
Party | (for usage part)
Provider

<domain> := Consumer |
Retailer

<part> := Initial |
Access |
<usage part>

<usage part>:= <feature set>Usage

<feature set> := Basic
BasicExt
Multiparty
MultipartyInd
Voting
ControlSR
PaSB
PaSBInd
CommS

A.2 Defined Modules and Interfaces

Table 8-1 lists the modules currently defined for Ret, together with the interfaces they contain:

Table 8-1. Modules, Interfaces and dependencies

Module	Contained interfaces	Dependencies
TINACommonTypes	none	none
TINAAccessCommonTypes	none	TINACommonTypes
TINAStreamTypes	none	TINACommonTypes
TINAUserInitial	i_UserInitial	TINAAccessCommonTypes
TINAUserAccess	i_UserAccessGetInterfaces i_UserAccess i_UserInvite i_UserTerminal i_UserAccessSessionInfo i_UserSessionInfo	TINAAccessCommonTypes
TINAProviderInitial	i_ProviderInitial i_ProviderAuthenticate	TINAAccessCommonTypes
TINAProviderAccess	i_ProviderAccessGetInterfaces i_ProviderAccessRegisterInterfaces i_ProviderAccessInterfaces i_ProviderAccess i_ProviderNamedAccess i_ProviderAnonAccess	TINAAccessCommonTypes TINAStreamTypes
TINARetConsumerInitial	i_ConsumerInitial	TINAUserInitial
TINARetConsumerAccess	i_ConsumerAccess i_ConsumerInvite i_ConsumerTerminal i_ConsumerAccessSessionInfo i_ConsumerSessionInfo	TINAUserAccess
TINARetRetailerInitial	i_RetailerInitial i_RetailerAuthenticate	TINAProviderInitial
TINARetRetailerAccess	i_RetailerAccess i_RetailerNamedAccess i_RetailerAnonAccess i_DiscoverServicesIterator	TINAProviderAccess
TINAUsageCommonTypes	none	TINACommonTypes
TINAProviderBasicUsage	i_ProviderBasicReq	TINAUsageCommonTypes TINASessionModel
TINAPartyBasicExtUsage	i_PartyBasicExtReq	TINAUsageCommonTypes TINASessionModel
TINAPartyMultipartyUsage	i_PartyMultipartyExe i_PartyMultipartyInfo (optional)	TINAUsageCommonTypes
TINAProviderMultipartyUsage	i_ProviderMultipartyReq	TINAUsageCommonTypes

Table 8-1. Modules, Interfaces and dependencies

Module	Contained interfaces	Dependencies
TINAPartyMultipartyIndUsage	i_PartyMultipartyInd	TINAUsageCommonTypes
TINAPartyVotingUsage	i_PartyVotingInfo	TINAUsageCommonTypes
TINAProviderVotingUsage	i_ProviderVotingReq	TINAUsageCommonTypes
TINAControlSRTypes	none	TINAUsageCommonTypes
TINAPartyControlSRUsage	i_PartyControlSRInd i_PartyControlSRInfo	TINAUsageCommonTypes TINAControlSRTypes
TINAProviderControlSRUsage	i_ProviderControlSRReq	TINAUsageCommonTypes TINAControlSRTypes
TINASBCommSCommonTypes	none	none
TINAStreamCommonTypes	none	TINACommonTypes TINASBCommSCommonTypes
TINAPaSBTypes	none	TINAUsageCommonTypes TINAStreamCommonTypes
TINAPartyPaSBUsage	i_PartyPaSBExe i_GeneralStreamInfo i_PartyGeneralStreamInfo i_PartyPaSBInfo	TINAPaSBTypes
TINAProviderPaSBUsage	i_ProviderPaSBReq	TINAPaSBTypes
TINAPartyPaSBIndUsage	i_PartyPaSBInd	TINAPaSBTypes
TINACommSCommonTypes	none	TINASBCommSCommonTypes TINAConSCommSCommonTypes
TINAConSCommSCommonTypes	none	TINASBCommSCommonTypes
TINAPartyCommSUsage	i_BasicTerminalFlowControl i_TerminalFlowControl	TINASBCommSCommonTypes TINAConSCommSCommonTypes TINACommSCommonTypes

A.3 Recorded problems with IDL

In the process of defining, writing and compiling the IDLs, some problems have been recorded that have had their impact on how some of the IDL is defined:

1. In order to work with OrbixWeb (current mapping) all IDL interfaces or types should be scoped within a MODULE.
2. In order to work with NEO, all IDL files should NOT start with a comment on the first line
3. In order to work with all ORBs, any types or interfaces can only be forward declared with the file in which they are later defined. i.e.forward declaration of an interface which is properly defined in a different file is not acceptable.
4. Typedefs of Object (i.e. CORBA::Object) should not be used.
5. Preprocessor commands (e.g. #define #ifndef etc) should not be terminated with comments. e.g.

```
#define i_retailerInitial_idl // Retailer Initial
```

6. Structures or exceptions should not contain submembers called type. e.g

```
struct aStruct {
       short type;
       boolean another;
};
```

7. All IDL files should end with an endline. i.e. after the last comment or preprocessor command, there should be an empty line.
8. The code for arrays of octets is not properly generated using Orbix. It's a known bug in Orbix. workaround: sequences
9. The typedefs of atomic types are not generated using the java mapping. This explains why in the application code some TINA types cannot be used.
10. A sequence definition using a scoped element compiles without problems with the Orbix IDL compiler. However, the generated C++ code is not compilable. Work-around: Always define sequences in the same module where the element type is defined. Example:

```
// Generated C++ does not compile:

// File A.idl
module A {
      typedef string t_string;
};

// File B.idl
#include "A.idl"

module B {
      typedef sequence<A::t_string> t_stringList;
};
```

11. In order to work with idldoc (an IDL to HTML document processor), unions must have a tag associated with each case body. Two or more tags cannot be associated with a single body. (All other IDL compilers, and the CORBA specification allow this. However Orbix doesn't allow the discriminator (switch tag) to be set. It can only be 'inferred' by setting the case body variable, and so Orbix doesn't work with multiple tags either.)

```
// Does not compile with idldoc:
union t_union switch (t_discriminator) {
      case okayTag: short okayCase;
      case firstCase:
      case secondCase: octet bodyForBoth;
};

// Does compile with idldoc:
union t_union switch (t_discriminator) {
      case okayTag: short okayCase;
      case firstCase: octet bodyForFirstCase;
      case secondCase: octet bodyForSecondCase;
};
```