Technical Specification Group Services and System Aspects *TSGS#6(99)543*
Meeting #6, Nice, France, 15-17 December 1999

Source: **TSG SA WG2**

Title: **Coversheet for 23.127 v.1.1.0**

Agenda Item: **5.2.3**

Release 1999 Submission form for 23.127 v.1.1.0

Work Area / Item:		**TS 23.127: Virtual Home Environment/Open Service Architecture**			
Affects:	**UE/MS:**	**CN: X**	**UTRAN:**	**Compatibility Issues:** **Yes:**	**No: X**
Expected Completion Date:		**March 2000**			
Services impacted:		**???**			
Specifications affected:		**TS 22.121, TS … (stage 3)**			
Tasks within work which are not complete:		- Registration of Service Capability Server. - Support for GPRS and SMS online charging. - Description of 'Load Balancing' service capability features - Support for USSD/SMS user interaction			
Consequences if not included in Release 1999:		**Deployment of VHE not possible.**			
Accepted by TSG#	**for late inclusion in Release 1999:**	**yes**			

Abstract of document:

This TS describes the Virtual Home Environment and related Open Service Architecture.

The Virtual Home Environment (VHE) is defined as a concept for personal service environment (PSE) portability across network boundaries and between terminals. The concept of the VHE is such that users are consistently presented with the same personalised features, User Interface customisation and services in whatever network and whatever terminal (within the capabilities of the terminal and the network), wherever the user may be located. For Release99, e.g. CAMEL, MExE and SAT are considered the mechanisms supporting the VHE concept.

The Open Service Architecture (OSA) defines an architecture that enables operator and third party applications to make use of network functionality through an open standardised interface (the OSA Interface). OSA provides the glue between applications and service capabilities provided by the network. In this way applications become independent from the underlying network technology. The applications constitute the top level of the Open Service Architecture (OSA). This level is connected to the Service Capability Servers (SCSs) via the OSA interface. The SCSs map the OSA interface onto the underlying telecom specific protocols (e.g. MAP, CAP, etc.) and are therefore hiding the network complexity from the applications.

Applications can be network/server centric applications or terminal centric applications. Terminal centric applications reside in the Mobile Station (MS). Examples are MExE and SAT applications. Network/server centric applications are outside the core network and make use of service capability features offered through the OSA interface. (Note that applications may belong to the network operator domain although running outside the core network. Outside the core

network means that the applications are executed in Application Servers that are physically separated from the core network entities).

Contentious Issues:

- User profile handling, including secure access to user data.
- Description of ‘Data download’ service capability features (pending decision in TSG SA WG1)
- Description of ‘Message Transfer’ service capability features (pending decision in TSG SA WG1)
- CAMEL ph3 (TS 22.078, TS 23.078, TS 29.078) support (support is agreed, the way how is open for discussion)

3G TS 23.127 1.10.0 (1999-10)

Technical Specification

3rd Generation Partnership Project;
Technical Specification Group Services and System Aspects;
Virtual Home Environment / Open Service Architecture
(3G TS 23.127 version 1.10.0)

Reference
DTS/TSGS-0223xxxU

Keywords
VHE, OSA

3GPP

Postal address

3GPP support office address
650 Route des Lucioles - Sophia Antipolis
Valbonne - FRANCE
Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Internet
http://www.3gpp.org

Contents

Foreword

This Technical Specification has been produced by the 3GPP.

The contents of the present document are subject to continuing work within the TSG and may change following formal TSG approval. Should the TSG modify the contents of this TS, it will be re-released by the TSG with an identifying change of release date and an increase in version number as follows:

Version 3.y.z

where:

- x the first digit:
  - 1 presented to TSG for information;
  - 2 presented to TSG for approval;
  - 3 Indicates TSG approved document under change control.
- y the second digit is incremented for all changes of substance, i.e. technical enhancements, corrections, updates, etc.
- z the third digit is incremented when editorial only changes have been incorporated in the specification;

11 Scope

This document specifies the stage 2 of the Virtual Home Environment and Open Service Architecture.

Virtual Home Environment (VHE) is defined as a concept for personal service environment (PSE) portability across network boundaries and between terminals. The concept of the VHE is such that users are consistently presented with the same personalised features, User Interface customisation and services in whatever network and whatever terminal (within the capabilities of the terminal and the network), wherever the user may be located. For Release99, e.g. CAMEL, MExE and SAT are considered the mechanisms supporting the VHE concept.

The Open Service Architecture (OSA) defines an architecture that enables operator and third party applications to make use of network functionality through an open standardised interface (the OSA Interface). OSA provides the glue between applications and service capabilities provided by the network. In this way applications become independent from the underlying network technology. The applications constitute the top level of the Open Service Architecture (OSA). This level is connected to the Service Capability Servers (SCSs) via the OSA interface. The SCSs map the OSA interface onto the underlying telecom specific protocols (e.g. MAP, CAP, H.323, SIP etc.) and are therefore hiding the network complexity from the applications.

Applications can be network/server centric applications or terminal centric applications. Terminal centric applications reside in the Mobile Station (MS). Examples are MExE and SAT applications. Network/server centric applications are outside the core network and make use of service capability features offered through the OSA interface. (Note that applications may belong to the network operator domain although running outside the core network. Outside the core network means that the applications are executed in Application Servers that are physically separated from the core network entities).

22 References

References may be made to:

a) Specific versions of publications (identified by date of publication, edition number, version number, etc.), in which case, subsequent revisions to the referenced document do not apply; or

b) All versions up to and including the identified version (identified by "up to and including" before the version identity); or

c) All versions subsequent to and including the identified version (identified by "onwards" following the version identity); or

d) Publications without mention of a specific version, in which case the latest version applies.

A non-specific reference to an ETS shall also be taken to refer to later versions published as an EN with the same number.

2.12.1 Normative references

[1] GSM 01.04 (ETR 350): ”Digital cellular telecommunication system (Phase 2+); Abbreviations and acronyms”

[2] GSM 02.57: ”Digital cellular telecommunication system (Phase 2+); Mobile Station Application Execution Environment (MExE); Service description”

[3] UMTS TS 23.057GSM 03.57: " Digital cellular telecommunication system (Phase 2+); Mobile Station Application Execution Environment (MExE); FunctionalService description - Stage2”

[4] UMTS TS 22.078GSM 02.78: " Digital cellular telecommunication system (Phase 2+); Customised Applications for Mobile network Enhanced Logic (CAMEL) (Phase3); Service definition description - Stage 1"

[5] UMTS TS 23.078GSM 03.78: "Digital cellular telecommunication system (Phase 2+); Customised Applications for Mobile network Enhanced Logic (CAMEL) (Phase3); Functional descriptionService definition - Stage 2"

[6] GSM 11.14: "Digital cellular telecommunication system (Phase 2+); Specification of the SIM Application Toolkit for the Subscriber Identity Module - Mobile Equipment; (SIM - ME) interface" < *Editor's note: check whether reference to 22.038 has to be included* >

[7] UMTS TS 22.101: "Universal Mobile Telecommunications System (UMTS): Service Aspects; Service Principles"

[8] UMTS TS 22.105: "Universal Mobile Telecommunications System (UMTS); Services and Service Capabilities"

[9] UMTS TS 22.121: "Universal Mobile Telecommunications System (UMTS); Virtual Home Environment"

[10] UMTS TR 22.905: "….

[11] IETF PPP Authentication Protocols - Challenge Handshake Authentication Protocol [RFC 1994, August1996]

2.22.2 Informative references

[1] UMTS TR 22.970: "Universal Mobile Telecommunications System (UMTS); Virtual Home Environment"

[2] World Wide Web Consortium Composite Capability/Preference Profiles (CC/PP): A user side framework for content negotiation (www.w3.org)

33 Definitions and abbreviations

3.13.1 Definitions

For the purposes of this TS, the following definitions apply:

HE-VASP: Home Environment Value Added Service Provider. This is a VASP that has an agreement with the Home Environment to provide services.

Local Service: A service, which can be exclusively provided in the current serving network by a Value added Service Provider.

Service Capabilities: Bearers defined by parameters, and/or mechanisms needed to realise services. These are within networks and under network control.

Service Capability Feature: Functionality offered by service capabilities that are accessible via the standardised OSA interface

Service Capability Server: Functional Entity providing OSA interfaces towards an application

Services: Services are made up of different service capability features.

Applications: Services, which are designed using service capability features.

OSA Interface: Standardised Interface used by applications to access service capability features.

Personal Service Environment: contains personalised information defining how subscribed services are provided and presented towards the user. The Personal Service Environment is defined in terms of one or more User Profiles.

Home Environment: responsible for overall provision of services to users

User Interface Profile: Contains information to present the personalised user interface within the capabilities of the terminal and serving network.

User Services Profile: Contains identification of subscriber services, their status and reference to service preferences.

User Profile: This is a label identifying a combination of one user interface profile, and one user services profile.

Value Added Service Provider: provides services other than basic telecommunications service for which additional charges may be incurred.

Virtual Home Environment: A concept for personal service environment portability across network boundaries and between terminals.

Further UMTS related definitions are given in 3G TS 22.101 and 3G TR 22.905.

3.23.2 Abbreviations

For the purposes of this TS the following abbreviations apply:

CAMEL	Customised Application For Mobile Network Enhanced Logic
CSE	Camel Service Environment
HE	Home Environment
HE-VASP	Home Environment Value Added Service Provider
HLR	Home Location Register
IDL	Interface Description Language
MAP	Mobile Application Part
ME	Mobile Equipment
MExE	Mobile Station (Application) Execution Environment
MS	Mobile Station
MSC	Mobile Switching Centre
OSA	Open Service Architecture
PLMN	Public Land Mobile Network
PSE	Personal Service Environment
SAT	SIM Application Tool-Kit
SCP	Service Control Point
SIM	Subscriber Identity Module Short Message Service
USIM	User Service Identity Module
VASP	Value Added Service Provider
VHE	Virtual Home Environment

Further GSM related abbreviations are given in GSM 01.04. Further UMTS related abbreviations are given in 3G T 22.905.

44 Virtual Home Environment

The Virtual Home Environment (VHE) is an important portability concept of the 3G mobile systems. It enables end users to bring with them their personal service environment whilst roaming between networks, and also being independent of terminal used.

The Personal Service Environment (PSE) describes how the user wishes to manage and interact with her communication services. It is a combination of a list of subscribed to services, service preferences and terminal interface preferences. PSE also encompasses the user management of multiple subscriptions, e.g. business and private, multiple terminal types and location preferences. The PSE is defined in terms of one or more User Profiles.

The user profiles consist of two kinds of information:

- Interface related information (User Interface Profile) and,

- Service related information (User Services profile).

Please see TS22.121 [9] for more details.

55 Open Service Architecture

In order to implement not known end user services/applications today, a highly flexible Open Service Architecture (OSA) is required. The Open Service Architecture (OSA) is the architecture enabling applications to make use of network capabilities. The applications will access the network through the OSA interface that is specified in this Technical Specification.

The access to network functionality is offered by different Service Capability Servers (SCSs) and appear as service capability features in the OSA interface. These are the capabilities that the application developers have at their hands when designing new applications (or enhancements/variants of already existing ones). The different features of the different SCSs can be combined as appropriate. The exact addressing (parameters, type and error values) of these features is described in stage 3 descriptions. These interface descriptions ("IDLs") are open and accessible to application developers, who can design services in any programming language. The service logic executes toward the OSA interfaces, while the underlying core network functions use their specific protocols.

The aim of OSA is to provide an extendible and scalable architecture that allows for inclusion of new service capability features and SCSs in future releases of UMTS with a minimum impact on the applications using the OSA interface.

To make it possible for application developers to rapidly design new and innovative applications, an architecture with open interfaces is imperative. By using object oriented techniques, like CORBA, it will be possible to use different operating systems and programming languages in application servers and service capability servers. The different servers interwork via the OSA interfaces. The service capability servers will serve as gateways between the network entities and the applications

5.15.1 Overview of the Open Service Architecture

The Open Service Architecture consists of three parts:

- **Applications**, e.g. VPN, conferencing, location based applications. These applications are implemented in one or more Application Servers;

- **Framework**, providing the applications with basic mechanisms that enable applications to make use of the service capabilities in the network. Examples of framework services are Authentication, Registration and Discovery. Before an application can use the network functionality made available through the Service Capability Servers, authentication between the application and framework is needed. After authentication, The discovery service then enables the application to find out from the framework what service capability features are provided by the Service Capability Servers. The service capability features are accessed by the methods defined in the OSA interface classes.

- **Service Capability Servers**, providing the applications with service capability features that are abstractions from underlying network functionality. Examples of service capability features offered by the Service Capability Servers are Call Control, Message Transfer and Location Information. Similar service capability features are possibly provided by more than one Service Capability Servers. For example, Call Control functionality might be provided by SCSs on top of CAMEL and MExE.

< Editor's note: text below (until figure) moved from former 5.2.1 to this place>

The OSA service capability features areinterface is specified in terms of a number of interface classes and their methods. The interface classes are divided into two groups:

- **framework interface classes,** describing the methods on the framework

- **service interface classes,** describing the methods on the service capability servers.

The interface classes are further divided into methods. For example, the Call Manager interface class might contain a method to create a call (which realises one of the Service capability features 'Initiate and create session' as specified in [9]).

For description purpose the interface classes belonging to the same subject are grouped together and called network services. For example, the interface classes Call Manager, Call and Leg constitute the Call Control network service.

Note that the "CSE" does not provide the service logic execution environment for applications using the OSA interface, since these applications are executed in Application Servers.

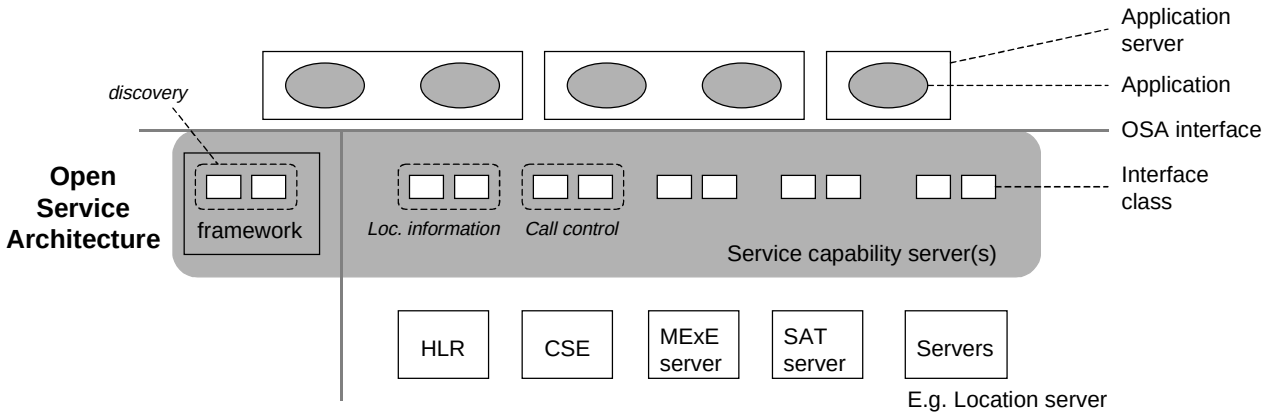


Figure 11 Overview of Open Service Architecture

This specification defines the OSA interface. OSA does not mandate any specific platform or programming language.

The Service Capability Servers that implement the OSA interface classes are functional entities that can be distributed across one or more physical node. For example, the Location interface classes and Call Control interface classes might be implemented on a single physical entity or distributed across different physical entities. Furthermore, a service capability server can be implemented on the same physical node as a network functional entity or in a separate physical node. For example, Call Control interface classes might be implemented on the same physical entity as the CAMEL protocol stack (i.e. in the SCP) or on a different physical entity.

Several options exist:

Option 1

The OSA interface classes are implemented in one or more physical entity, but separate from the physical network entities. Figure 2Figure 2 shows the case where the OSA interface classes are implemented in one physical entity, called "gateway" in the figure. Figure 4Figure 3 shows the case where the SCSs are distributed across several 'gateways'.

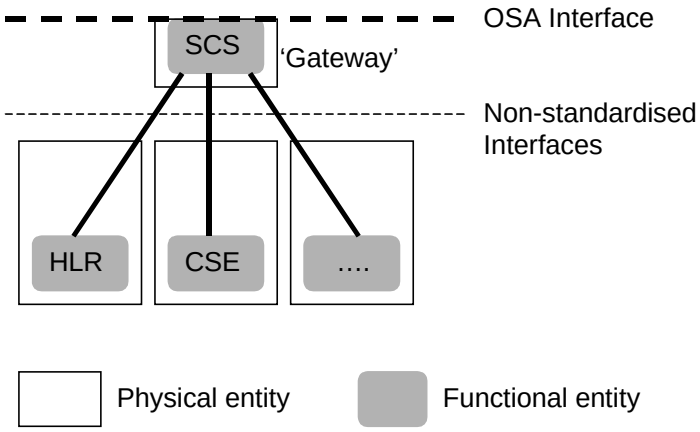


Figure 22 SCSs and network functional entities implemented in separate physical entities

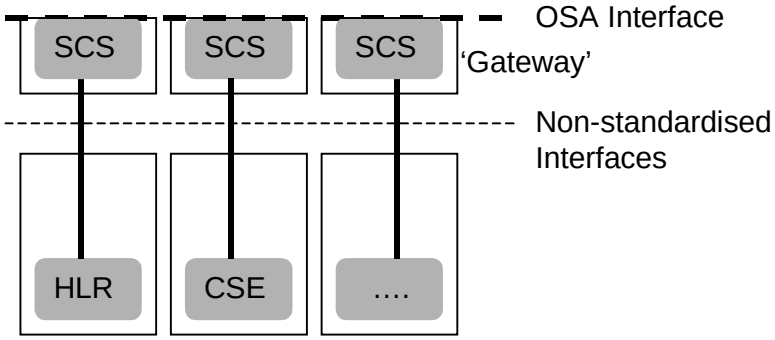


Figure 43 SCSs and network functional entities implemented in separate physical entities, SCSs distributed across several 'gateways'.

Option 2

The OSA interface classes are implemented in the same physical entities as the traditional network entities (e.g. HLR, CSE), see Figure 6Figure 4.

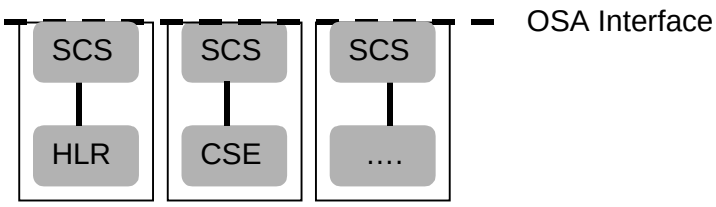


Figure 64 SCSs and network functional entities implemented in same physical entities

Option 3

Option 3 is the combination of option 1 and option 2, i.e. a hybrid solution.

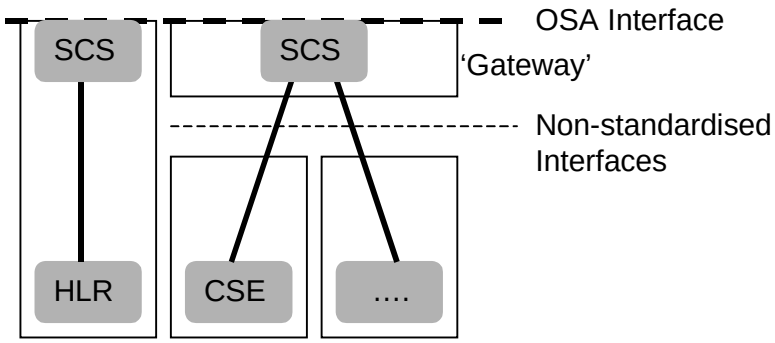


Figure 85 Hybrid implementation (combination of option 1 and 2)

It shall be noted that in all cases there is only one framework.

From the application point of view, it shall make no difference which implementation is chosen, i.e. in all cases the same network functionality is perceived by the application. The applications shall always be provided with the same set of interface classes and a common access to framework and service interface. It is the framework that will provide the applications with an overview of available service capability features and how to make use of them.

~~*<Remark to Application Interface:*~~

~~*The network functionality can be accessed via network and mobile based applications, e.g. from mobile station to control subscriber data in the HLR. In other words, future contributions might show that parts of the OSA application interface might be implemented on service capability servers and parts in mobile terminals>*~~

~~5.2~~5.2 Basic mechanisms in the Open Service Architecture

This section explains what basic mechanisms are executed in OSA prior to offering and activating applications.

Some of the mechanisms are applied only once (e.g. establishment of service agreement), others every time a user subscription is made to an application (e.g. enabling the call attempt event for a new user).

Basic mechanisms between Application and Framework:

- **Authentication**: Once a off line service agreement exists, the application can access the authentication interface. The authentication model of OSA is a peer-to-peer model. The application must authenticate the framework as well as be authenticated by the framework. The application must be authenticated before any other OSA interface is allowed to be used.

- **Authorisation**: Authorisation is distinguished from authentication in that authorisation is the action of determining what a previously authenticated application is allowed to do. Authentication must precede authorisation.

- **Discovery of framework and service interfaces**. After successful authentication, applications can obtain available framework interface classes and use the discovery interface to get the allowed service interface classes. The Discovery interface can be used at any time after successful authentication.

- **Establishment of service agreement**. Before any application can interact with a network service capability a service agreement must be established. A service agreement consist of an off-line (e.g. by physically passing messages) and on-line part. The application has to sign an on-line service agreement before any other access to the network service interface is allowed.

-

Basic mechanism between Framework and Service Capability Server:

- **Registering of service interfaces**. Interface classes offered by a Service Capability Server can be registered at the Framework. In this way the Framework can inform the Application upon request about available service

interface classes (Discovery). This mechanism is in general applied when installing or upgrading a Service Capability Server.

Basic mechanisms between Application Server and Service Capability Server:

- **Request of event notifications**. This mechanism is applied when a user has subscribed to an application and that application needs to be invoked upon receipt of events from the network related to the user. For example, when a user subscribes to screening application, the application needs to be invoked when the user makes a call. A call notification event is in this case requested on the Calling and/or Called Party Number of the user.

5.35.3 Base interface classes

The base class interface classes described in this sub clause are provided for completeness of the documentation. With object oriented design all classes are based on a base class. This base class normally does very little and new methods (i.e. functionality) are added by each class further in the hierarchy.

5.3.15.3.1 Base Interface Class

This class is the foundation of the all interfaces and shall be inherited by all following interfaces. It contains no further methods.

Name Base_Interface

Method

Parameters

Returns

Errors

5.3.25.3.2 Base Service Interface class

This class provides the base for ALL service and framework interfaces described in the following chapters. It allows an application to set a reference to the application, which is used by the OSA interface to respond to the application, which originally initiated the request. For example, when an application wants to be notified upon the receipt of the "called party busy" event, the Service Capability Server must know where to send the notification. This reference can be provided by the application with the setCallBack method across the OSA interface.

Name Base_Service_Interface

Method **`setCallback()`**

This method specifies the reference address of the callback interface that a service uses to invoke methods on the application.

Parameters

AppInterface

Specifies a reference to the application interface, which is used for callbacks.

Returns

Errors

66 Framework service capability features

< Note: It is proposed to start chapter 6 with "establishing a service agreement", i.e. 6.1.1 becomes 6.1. "Authentication then becomes the second section in chapter 6, i.e. 6.2 >

6.1Authentication

The API supports multiple authentication techniques. The procedure used to select an appropriate technique for a given situation is described below. The authentication mechanisms may be supported by cryptographic processes to provide confidentiality, and by digital signatures to ensure integrity. The inclusion of cryptographic processes and digital signatures in the authentication procedure depends on the type of authentication technique selected. In some cases strong authentication may need to be enforced by the Network to prevent misuse of resources. In addition it may be necessary to define the minimum encryption key length that can be used to ensure a high degree of confidentiality.

The authentication interface must be the first interface invoked by an application. Invocations of other interfaces will fail until authentication has been successfully completed.The address of the Authentication Framework interface is administered in the application prior to the API being used. This address is made available by the Home Environment, possibly also for a particular HE-VASP.

6.1.16.1 —Establishing a Service Agreement

Before any application can interact with the network a service agreement will have to be established or an existing agreement will need modification or indeed termination if it is being superseded. An appropriate procedure is required to cater for each of these cases. Off-line agreement may be done by physically passing messages in a secure manner using cryptographic or non-cryptographic techniques. On-line agreement, on the other hand, can only be done in practice using cryptographic techniques.

The procedure outlined below describes on-line establishment of service agreements using cryptographic techniques only, since this is considered to be an integral part of the Authentication framework. However, the procedures may also be a basis for an off-line establishment of service agreements using cryptographic techniques.

A procedure to establish a service agreement begins with the application and Authentication Framework authenticating each other. This uses an authentication mechanism chosen by the Authentication Framework.

After authentication the application and Authentication Framework negotiate a service agreement which will involve each party digitally signing the agreement.

- A application sends an initial message to the Authentication Framework - this will include the authentication capabilities of the application. The Authentication Framework will then choose an authentication mechanism based on information about the authentication capabilities of the framework, application and the service requested. If the application is capable of handling more than one mechanism then the Authentication Framework chooses one preferred authentication option.

- The Authentication Framework sends the identity of the prescribed authentication mechanism to the application. The Authentication Framework will instruct the application to perform the agreed mechanism.

- The application and Authentication Framework interact to authenticate each other. Depending on the mechanism prescribed, this procedure may consist of a number of messages e.g. a challenge/ response protocol. It is assumed that any cryptographic process for enciphering the link is handled at a lower layer (and is outside the scope of this specification).

- The application is now authorised and can access the Discovery Framework Interface using the `obtainInterface` method. The application uses functions of the Discovery framework interface to look for the services it needs. Using the `selectService()` and `signServiceAgreement()` methods it requests the use of a service.

The application and Authentication Framework can then negotiate a service agreement. Optionally, the Authentication Framework may request re-authorisation. Each party then digitally signs the agreement.

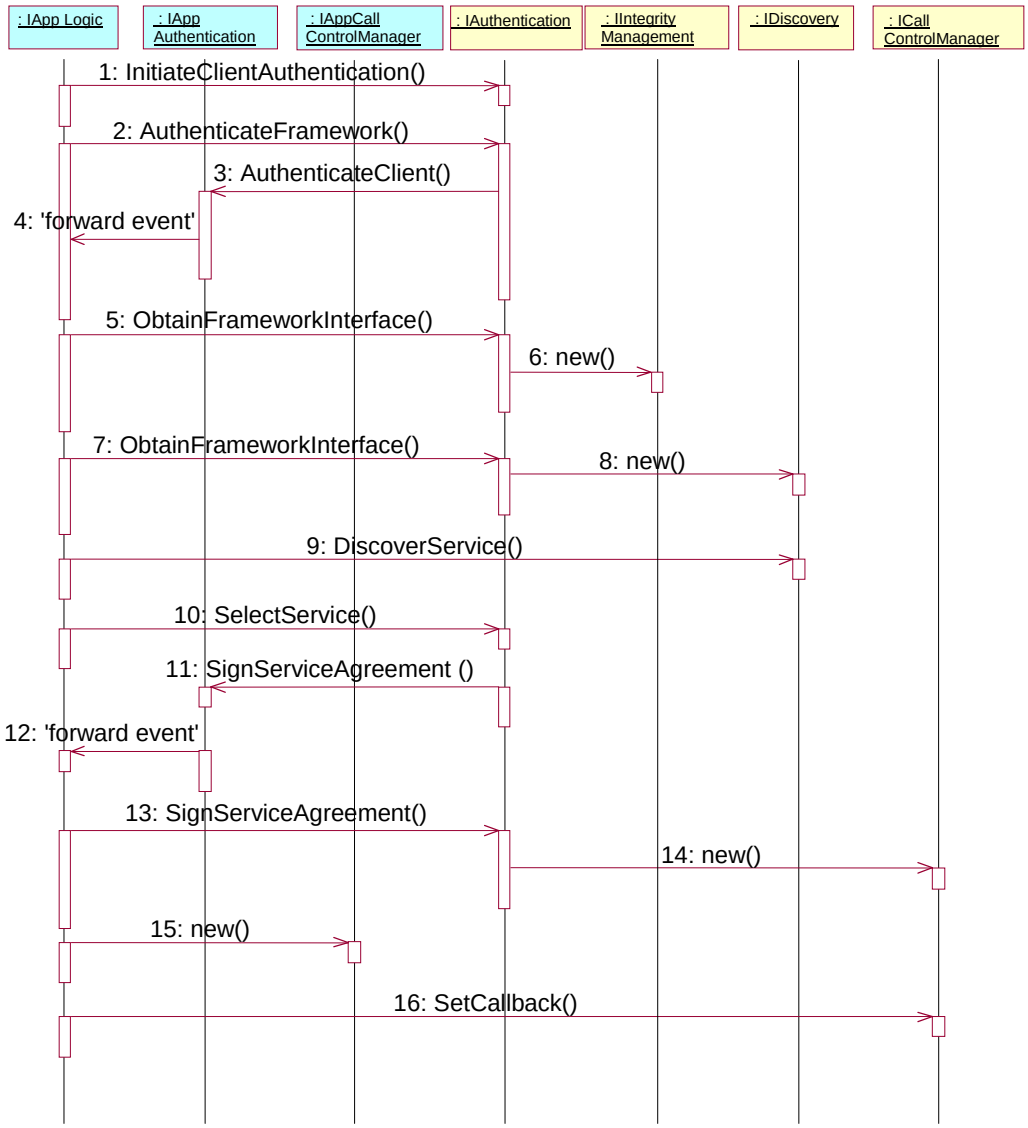


Figure 96 Sequence diagram illustrating authentication, service selection and signing of service agreement

< Editor's note: clarifying text needed to explain what is meant by "client" in the interface class defintions in chapter 6 and 7 >

< Editor: in the remainder of chapters 7 and 8, remove "service interface", "application interface", "framework interface"; remove "name" row by "direction" row

6.26.2 Authentication

The OSA authentication interface supports multiple authentication techniques. The procedure used to select an appropriate technique for a given situation is described below. The authentication mechanisms may be supported by cryptographic processes to provide confidentiality, and by digital signatures to ensure integrity. The inclusion of cryptographic processes and digital signatures in the authentication procedure depends on the type of authentication technique selected. In some cases strong authentication may need to be enforced by the network to prevent misuse of resources. In addition it may be necessary to define the minimum encryption key length that can be used to ensure a high degree of confidentiality.

The authentication interface must be the first interface invoked by an application. Invocations of other interfaces will fail until authentication has been successfully completed. The address of the Authentication framework interface is administered in the application prior to the interface being used. This address is made available by the Home Environment, possibly also for a particular HE-VASP.

The OSA authentication interface is defined with the following interface classes and methods:

6.1.2Authentication interface class

<Note: order of methods is changed compared to V100, to align more with sequence diagram shown in section 6.1. This however not shown with revision marks. Textual changes are highlighted with revision marks >

Method `InitiateClientAuthentication()`

The application uses this method to initiate the authentication process. The mechanism returned by the framework is the mechanism preferred by the framework. This should be within the application capability. If a mechanism within the application's capability cannot be found, the framework must return an error.

Direction Application to framework

Parameters

ApplicationID

This is the ID for the application.. The application ID can be used by the framework to reference the correct application Public Key (the key management system is currently outside of the scope of the specification).

AppInterface

Specifies a reference to the application interface, which is used for callbacks.

ClientCapability

This is the authentication capability of the application. This is a list of capabilities separated by a comma.

Returns

PrescribedMechanism

This is the mechanism returned by the framework to indicate the mechanism preferred by the framework for the authentication process. If the value of the `prescribedMechanism` returned by the framework is not understood by the application, it is considered a catastrophic error and the application must abort.

Errors

`INVALID_APPLICATIONID`

Returned by the framework if the framework cannot find the `applicationID` parameter. The value of the parameter `prescribedMechanism` is `NULL` in this situation

`INVALID_CLIENT_CAPABILITY`

If the value of the `clientCapability` parameter is not valid. The value of the parameter `prescribedMechanism` is set to `NULL`.

Method `AuthenticateFramework()`

This method is used by the application to authenticate the framework. The framework must respond with the correct responses to the challenges presented by the application. The application ID received in the `initiateClientAuthentication()` can be used by the gateway to reference the correct application public key (the key management system is currently outside of the scope of this specification).

Direction Application to framework

Parameters

Challenge

The challenge presented by the application to be responded to by the framework. The challenge mechanism used will be in accordance with [11]the IETF *PPP Authentication Protocols – Challenge Handshake Authentication Protocol* [RFC 1994, August1996]. The challenge will be encrypted with the mechanism prescribed by the `InitiateClientAuthentication()`.

Returns

Response

This is the response of the framework to the challenge of the application in the current sequence. The response will be the challenge, decrypted with the mechanism prescribed by the `initiateClientAuthentication()`.

Errors

-

Method `AuthenticateClient()`

This method is used by the framework to authenticate the application. The application must respond with the correct responses to the challenges presented by the framework. The Gateway ID (The address of the gateway being used – which will be pre-provisioned in the application environment) can be used by the application to reference the correct gateway Public Key. The key management system is currently outside of the scope of the specification.

Direction

Framework to application

Parameters

Challenge

The challenge presented by the framework to be responded to by the application. The challenge will be encrypted with the mechanism prescribed by the `initiateClientAuthentication()`.

Returns

Response

This is the response of the application to the challenge of the framework in the current sequence. The response will be the challenge, decrypted with the mechanism prescribed by the `initiateClientAuthentication()`.

Errors

-

Method `terminateClientAuthentication()`

This method is used by the application to terminate authentication with the framework. The application ID received in the `initiateClientAuthentication()` can be used by the gateway to reference the correct application public key or another secret key (the key management system is currently outside of the scope of this specification).

Direction

Application to framework

Parameters

TerminationText

This is the termination text that is signed by the application using the private key of the application or another secret key.

DigitalSignature

This is the digital signature of the termination text. The framework uses this to check the decrypted `terminationText`. If a match is made, the authentication is terminated, otherwise an error is returned.

Returns

-

Errors

-

Method `TerminateAppClientAuthentication()`

This method is used by the framework to terminate authentication with the application.

Direction Framework to application

Parameters

TerminationText

This is the termination text that is signed by the framework using the private key of the framework or another secret key.

DigitalSignature

This is the digital signature of the termination text. The application uses this to check `terminationText`. If a match is made, the authentication is terminated, otherwise an error is returned.

Returns -

Errors -

~~**Direction** Application to framework~~

Method `ObtainFrameworkInterface()`

This method is used by the application to obtain other framework interfaces. Only by using this method can the application obtain the interface references to the other framework interfaces.

Direction **Application to framework**

Parameters

FrameworkId

The name of the framework interface to which a reference to the interface is requested. The interfaces allowed include discovery, `event notification` and OA & M. This parameter uniquely defines the service of interest from the application.

AppInterface

Specifies a reference to the application interface, which is used for callbacks. If an application interface is not needed, then the value of this parameter should be `NULL`.

Returns

FrameworkInterface

This is the interface reference to the interface asked for by the application.

Errors `INVALID_INTERFACEID`

Returned if the framework is given an invalid interface name

Method `SelectService()`

This method is used by the application to identify the service that the application is interested in.

Direction Application to framework

Parameters

ServiceID

This uniquely defines the service required.

ServiceProperties

The names and values of the trading data properties that the service should support.

Returns

ServiceToken

This is a free format text token returned by the framework, which can be signed as part of a service agreement. This will contain operator specific information relating to the service level agreement for use of the API.

Errors

-

Method `signAppServiceAgreement()`

This method is used by the framework to ask the application to sign an agreement on the service to continue the authentication process.

Direction

Framework to application

Parameters

ServiceToken

This is the token passed back from the framework in a previous `selectService()` method call. This token is used to identify the service requested by the application.

AgreementText

This is the agreement text that is to be encrypted by the application using the private key of the application.

SigningAlgorithm

This is the algorithm used to compute the digital signature.

Returns

DigitalSignature

This is the encrypted version of the agreement text given by the application.

Errors

-

Method `SignServiceAgreement()`

This method is used by the application to ask the framework to sign an agreement on the service to continue the authentication process.

Direction

Application to framework

Parameters

ServiceToken

The token returned by the framework in a previous `selectService()` method call to identify the service requested by the application.

AgreementText

This is the agreement text that is to be encrypted by the framework using the private key of the framework.

SigningAlgorithm

This is the algorithm used to compute the digital signature. The signing algorithm must be known to the framework and mandated by the prescribed mechanism returned by the framework.

Returns

DigitalSignature

This is the encrypted version of the agreement text given by the framework.

ServiceManagerInterface

This identifies the address of the service manager interface for the requested service.

Errors `INVALID_SIGNING_ALGORITHM`

Returned by the framework when the signing algorithm does not match with the prescribed mechanism.

Method `TerminateServiceAgreement()`

This method is used by the application to ask the framework to terminate an agreement on the service.

Direction Application to framework

Parameters

ServiceToken

The token returned by the framework in a previous `selectService()` method call to identify the service requested by the application.

TerminationText

This is the termination text that is to be digitally signed by the application. The signing algorithm used is the same as for the function `signServiceAgreement()`.

DigitalSignature

This is the digital signature of the termination text. The framework uses this to check the t`erminationText`. If a match is made, the service agreement is terminated, otherwise an error is returned.

Returns -

Errors -

Method `TerminateAppServiceAgreement()`

This method is used by the framework to terminate an agreement with the application on the service.

Direction Framework to application

Parameters

ServiceToken

This is the token passed back from the framework in a previous `selectService()` method call. This token is used to identify the service requested by the application.

TerminationText

This is the termination text that is digitally signed by the framework. The signing algorithm used is the same as for the function `signServiceAgreement()`.

DigitalSignature

This is the digital signature of the termination text. The application uses this to check the `terminationText`. If a match is made, the service agreement is terminated, otherwise an error is returned.

Returns -

Errors -

6.26.3 Authorisation

< Editor's note: to be completed in e-mail discussion >

6.36.4 Event Notification

Method **`EnableNotification()`**

This method is used to enable generic notifications so that events can be sent to the application.

Direction Application to framework

Parameters

AppInterface

If this parameter is set (i.e. not NULL) it specifies a reference to the application interface, which is used for callbacks. If set to NULL, the application interface defaults to the interface specified via the `obtainInterface()` method (refer to Authentication interface).

EventCriteria

Specifies the event specific criteria used by the application to define the event required.

Returns

AssignmentID

Specifies the ID assigned by the framework for this newly enabled event notification.

Errors -

Method **`DisableNotification()`**

This method is used by the application to disable generic notifications from the framework.

Direction Application to framework

Parameters

EventCriteria

Specifies the event specific criteria used by the application to define the event to be disabled.

AssignmentID

Specifies the assignment ID given by the framework when the previous `enableNotification()` was called.

Returns -

Errors `INVALID_ASSIGNMENTID`

Returned if the assignment ID does not correspond to one of the valid assignment Ids.

Method **`EventNotify()`**

This method notifies the application of the arrival of a generic event.

Direction Framework to application

Parameters

EventInfo

Specifies specific data associated with this event.

AssignmentID

Specifies the assignment id which was returned by the framework during the `enableNotification()` method. The application can use assignment id to associate events with event specific criteria and to act accordingly.

Returns -

Errors -

Method **`NotificationTerminated()`**

This method indicates to the application that all generic event notifications have been terminated (for example, due to faults detected).

Direction Framework to application

Parameters -

Returns -

Errors -

6.46.5 Registration

<Editor's note: to be completed in e-mail discussion >

The Registration service is used to register the functionality of the service capability servers at the Framework. In this way, the Framework can inform the application about the network functionality available when the application issues a DiscoverService. Upon introduction of a new or update of an existing service capability server, the functionality provided by this service capability server can be registered at the Framework.

6.56.6 Discovery

Method **`DiscoverService()`**

The discoverService operation is the means by which a user (client application) is able to obtain the reference (address) to the services that meet its requirements. The client specifies criteria about the service it is interested in and the framework returns the identifier for the service that meet the criterias.

Direction Application to framework

Parameters

ServiceProperties

The names and values of the trading data properties that the service should support, includes Service Type name, Service constraints.

Returns

ServiceID

This is the unique identity of the service.

Errors -

77 Non-Framework service capability features

The service capability features provided to the application by service capabilities servers to enable access to network resources. *<Editor's note: information flows might be needed to express what information is exchanged (and in what order) between application and service capability servers; this needs then also be reflected in the document structure>*

Note: when the direction of a method in an interface class is “application to network”, this means that the method is invoked from the application to an SCS residing on the network side of the OSA interface.

7.1 Call Control

The Call control network service consist of three interface classes:

1. Call manager, containing management function for call related issues
2. Call, containing methods to control a call
3. Leg, containing methods to control individual legs in a call

A call can be controlled by one Call Manager; A call can consist of up to n Legs, where n is determined by the Service Capability used.

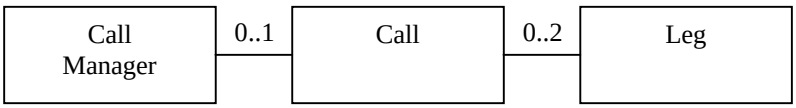


Figure 107 Call control class hierarchy

The Call Control service capability features are described in terms of the methods in the Call Control interface classes. Table 1Table 1 gives an overview of the Call Control methods and to which interface classes these methods belong.

CallManager	**Call**	**Leg**
CreateCall	RouteCallToDestination_Req	RouteCallLegToAddress
EnableCallNotification	RouteCallToDestination_Res	GetCall
DisableCallNotification	RouteCallToDestination_Err	GetCallLegState
CallNotificationTerminated	RouteCallToOrigination_Req	GetCallLegType
CallEventNotify	RouteCallToOrigination_Res	GetCallLegInfo_Req
CallFaultDetected	RouteCallToOrigination_Err	GetCallLegInfo_Res
	ReleaseCall	GetCallLegInfo_Err
	DeassignCall	GetAddresses
	GetCallInfo_Req	CallLegEventReport_Req
	GetCallInfo_Res	CallLegEventReport_Res
	GetCallInfo_Err	CallLegEventReport_Err
	GetCallState	
	GetCallLegs	
	GetControlLeg	
	CreateCallLeg	
	AttachCallLeg	

CallManager	Call	Leg
	DetachCallLeg	

Table 11 Overview of Call Control interface classes and their methods

<Editor's note: it is proposed to put the Call Control methods in section 7.1 in a different order than in version 1.0.0, to group related methods together (to increase readibility of the spec). For example, place RoteCallToDestination Req, Res, Err directly after each other >

7.1.17.1.1 Call Manager

The generic call manager interface class provides the management functions to the generic call Service Capability Features. The application programmer can use this interface, to create call objects and to enable or disable call-related event notifications.

Method **`CreateCall()`**

This method is used to create a new call object.

Direction Application to network

Parameters

AppCall
Specifies the application interface for callbacks from the call created.

Returns

Call
Specifies the interface reference of the call created.

CallSessionID

Specifies the call session ID of the call created.

Errors

Method **`EnableCallNotification()`**

This method is used to enable call notifications ~~so that events can~~ to be sent to the application.

Direction Application to network

Parameters

AppInterface
If this parameter is set (i.e. not NULL) it specifies a reference to the application interface, which is used for callbacks. If set to NULL, the application interface defaults to the interface specified via the `setCallback()` method.

EventCriteria
Specifies the event specific criteria used by the application to define the event required. Examples of events are "incoming call attempt reported by network", "answer", "no answer", "busy".

Returns

AssignmentID

Specifies the ID assigned by the generic call control manager interface for this newly-enabled event notification.

Errors -

Method `DisableCallNotification()`

This method is used by the application to disable call notifications.

Direction Application to network

Parameters

EventCriteria

Specifies the event specific criteria used by the application to define the event to be disabled. Examples of events are “incoming call attempt reported by network”, “answer”, “no answer”, “busy”.

AssignmentID

Specifies the assignment ID given by the generic call control manager interface when the previous `enableNotification()` was called.

Returns -

Errors `INVALID_ASSIGNMENTID`

Returned if the assignment ID does not correspond to one of the valid assignment Ids.

Method `CallEventNotify()`

This method notifies the application of the arrival of a call-related event.

Direction Network to application

Parameters

Call

Specifies the reference to the call interface to which the notification relates.

EventInfo

Specifies data associated with this event. These data include originatingAddress, originalDestinationAddress, redirectingAddress and AppInfo (see for more explanation on these data the `RouteCallToDestination()` method).

AssignmentID

Specifies the assignment id which was returned by the `enableNotification()` method. The application can use assignment id to associate events with event specific criteria and to act accordingly.

AppInterface

Specifies a reference to the application interface which implements the callback interface for the new call.

Returns -

Errors -

~~**Method** `CallOverloadCeased()`~~

~~This method indicates that the network has detected that the overload has ceased and has automatically removed any load controls on calls requested to a particular address range or calls made to a particular destination within the generic call control service.~~

~~**Direction** Network to application~~

~~**Parameters** **AddressRange**~~

Specifies the address range within which the overload has ceased.

OverloadType
Specifies the type of overload that has ceased.

Returns

Errors

Method `callAborted()`

This method indicates to the application that the call object has aborted or terminated abnormally. No further communication will be possible between the call and application.

Direction Network to application

Parameters

call
Specifies the call interface that has aborted or terminated abnormally.

callSessionID
Specifies the call session ID of the call that has aborted or terminated abnormally.

Returns -

Errors -

Method `CallFaultDetected()`

This method indicates to the application that a fault has been detected in the call.

Direction Network to application

Parameters

Call
Specifies the call interface in which the fault has been detected.

CallSessionID
Specifies the call session ID of the call in which the fault has been detected.

Fault
Specifies the fault that has been detected.

Returns -

Errors -

Method `callNotificationTerminated()`

This method indicates to the application that all event notifications have been terminated (for example, due to faults detected).

Direction Network to application

Parameters -

Returns -

Errors -

7.1.27.1.2 Call

The generic call interface represents the interface to the generic call Service Capability Feature. It provides a structure to allow simple and complex call behaviour to be used.

Method `RouteCallToDestination_Req()`

This asynchronous method requests routing of the call (and inherently attached parties) to the destination party (specified in the parameter `TargetAddress`). The destination party is attached to the call via a passive leg. This means that the call is not automatically released if the destination party disconnects from the call; only the leg with which the destination party was attached to the call is released in that case. `RouteCallToDestination_Req` implicitly creates the passive leg, i.e. the application does not have to issue a `CreateCallLeg` method.~~, via a passive call leg (which is implicitly created)~~.

Direction Application to network

Parameters

CallSessionID

Specifies the call session ID of the call.

ResponseRequested

Specifies the set of observed events that will result in a `routeCallToDestination_Res()` being generated.

TargetAddress

Specifies the destination party to which the call should be routed.

OriginatingAddress

Specifies the address of the originating (calling) party.

OriginalDestinationAddress

Specifies the original destination address of the call, i.e. the address as specified by the originating party. This parameter should be equal to the `OriginalDestinationAddress` as received by the application in the `EventInfo` parameter of the `CallEventNotify` method.

RedirectingAddress

Specifies the last address from which the call was redirected.

AppInfo

Specifies application-related information pertinent to the call (such as alerting method, tele-service type, service identities and interaction indicators).

Returns -

Errors -

Method `routeCallToDestination_Res()`

This asynchronous method indicates that the request to route the call to the destination was successful, and indicates the response of the destination party (for example, the call was answered, not answered, refused due to busy, etc.). If the call is answered, then a (passive) call leg object will be created for that leg of the call.

Direction Network to application

Parameters

CallSessionID
Specifies the call session ID of the call.

CallLeg
Specifies the interface of the call leg associated with the destination party.

CallLegSessionID
Specifies the call leg session ID of the call leg associated with the destination party.

EventReport
Specifies the result of the request to route the call to the destination party. It also includes the mode that the call object is in, the call leg generating the report (if applicable) and other related information.

Returns -

Errors -

Method `routeCallToDestination_Err()`

This asynchronous method indicates that the request to route the call to the destination party was unsuccessful - the call could not be routed to the destination party (for example, the network was unable to route the call, the parameters were incorrect, the request was refused, etc.).

Direction Network to application

Parameters

callSessionID
Specifies the call session ID of the call.

error
Specifies the error which led to the original request failing.

Returns -

Errors -

Method `RouteCallToOrigination_Req()`

This asynchronous method requests routing of a call to the first call party, via a controlling call leg (which is implicitly created). When a user attached to the controlling leg disconnects from the call, the whole call is released (as opposed to a user attached via a passive leg that disconnects). The call object must already have been created. This method is typically used by an application to set up a call “out of the blue”, i.e. the call is not initiated by an originating party.

Direction Application to network

Parameters

CallSessionID
Specifies the call session ID of the call.

ResponseRequested
Specifies the set of observed events that will result in a `routeCallToOrigination_Res()` will be generated.

TargetAddress

Specifies the origination party to which the call should be routed.

OriginatingAddress

Specifies the address of the originating (calling) party.

AppInfo

Specifies application-related information pertinent to the call (such as alerting method, tele-service type, service identities and interaction indicators).

Returns -

Errors -

Method `routeCallToOrigination_Res()`

This asynchronous method indicates that the request to route a call to the first call party was successful, and indicates the response of that party (for example, the call was answered, not answered, refused due to busy, etc.). If the call is answered, then a (controlling) call leg object will be created for that leg of the call.

Direction Network to application

Parameters

callSessionID

Specifies the call session ID of the call.

callLeg

Specifies the interface of the call leg associated with the origination party.

callLegSessionID

Specifies the call leg session ID of the call leg associated with the origination party.

eventReport

Specifies the result of the request to route the call to the origination party. It also includes the mode that the call object is in, the call leg generating the report (if applicable) and other related information.

Returns -

Errors -

Method `RouteCallToOrigination_Err()`

This asynchronous method indicates that the request to route the call to the originating party was unsuccessful (for example, the network was unable to route the call, the parameters were incorrect, the request was refused, etc.).

Direction Network to application

Parameters

CallSessionID

Specifies the call session ID of the call.

Error

Specifies the error which led to the original request failing.

Returns -

Errors -

Method `ReleaseCall()`

This method requests the release of the call and associated objects.

Direction Application to network

Parameters

CallSessionID

Specifies the call session ID of the call.

Cause

Specifies the cause of the release.

Returns -

Errors -

Method `DeassignCall()`

This method requests that the relationship between the application and the call and associated objects be de-assigned. It leaves the call in progress, however, it purges the specified call object so that the application has no further control of call processing. If a call is de-assigned that has event reports, call information reports or call Leg information reports requested, then these reports will be disabled and any related information discarded.

Direction Application to network

Parameters

CallSessionID

Specifies the call session ID of the call.

Returns -

Errors -

Method `GetCallInfo_Req()`

This asynchronous method requests information associated with the call to be provided at the appropriate time (for example, to calculate charging). This method must be invoked before the call is routed to a target address. The call object will exist after the call is ended if information is required to be sent to the application at the end of the call. The call information will be sent after any call event reports.

Note: At the end of the call with respect to either a particular call leg or the entire call, the call information must be sent before the objects of concern are deleted.

Direction Application to network

Parameters

CallSessionID

Specifies the call session ID of the call.

CallInfoRequested

Specifies the call information that is requested.

Returns -

Errors -

Method `GetCallInfo_Res()`

This asynchronous method reports all the necessary information requested by the application, for example to calculate charging.

Direction Network to application

Parameters

CallSessionID
Specifies the call session ID of the call.

CallInfoReport
Specifies the call information requested.

Returns -

Errors -

Method `GetCallInfo_Err()`

This asynchronous method reports that the original request was erroneous, or resulted in an error condition.

Direction Network to application

Parameters

CallSessionID
Specifies the call session ID of the call.

Error
Specifies the error which led to the original request failing.

Returns -

Errors -

Method `SetCallChargePlan()`

Allows an application to include charging information in network generated CDR.

Parameters

CallSessionID
Specifies the call session ID of the call.

CallChargePlan
Application specific charging information.

Returns -

Errors -

Method `SuperviseCall_Req()`

The application calls this method to supervise a call. The application can set a granted connection time for this call. If an application calls this function before it calls a `routeCallToDestination_Req()` or a user interaction function the time measurement will

start as soon as the call is answered by the B-party or the user interaction system.

Direction Application to network

Parameters

CallLegSessionID
Specifies the call leg session ID of the call leg.

Duration
Specifies the granted duration of the call/session in:

- time in milliseconds for the connection, or;
- Total data transferred in …

TarrifSwitch
Specifies an optional tariff switch indicating a change in tariff.

Treatment
Specifies how the network should react after the granted connection time expired.

Returns -

Errors -

Method `SuperviseCall_Res()`

This asynchronous method reports a call supervision event to the application.

Direction Network to application

Parameters

CallSessionID
Specifies the call session ID of the call.

Report
Specifies the situation, which triggered the sending of the call supervision response.

UsedTime
Specifies the used time for the call supervision (in milliseconds).

Returns -

Errors -

Method `SuperviseCall_Err()`

This asynchronous method reports a call supervision error to the application.

Direction Network to application

Parameters

CallSessionID
Specifies the call session ID of the call.

Error
Specifies the error which led to the original request failing.

Returns -

Errors -

Method `GetCallState()`

This method requests the current state of the call.

Direction Application to network

Parameters

CallSessionID

Specifies the call session ID of the call.

Returns

State

Specifies the current state of the call (e.g. “idle”, “active”, “inactive”, “released”).

Errors -

Method `GetCallLegs()`

This method requests the identification of the call leg objects associated with the call object.

Direction Application to network

Parameters

CallSessionID

Specifies the call session ID of the call.

Returns

CallLegList

Specifies the call legs associated with the call. The references passed in this list are in the same index order as the IDs passed in the call leg session ID list.

CallLegSessionIDList

Specifies the call leg session IDs associated with the call. The IDs passed in this list are in the same index order as the references passed in the call leg list.

Errors -

Method `CreateCallLeg()`

This method requests the creation of a new call leg object The call leg will be associated with the call, but not attached. The call leg can be attached to the call (using `attachCallLeg`) when the call leg is in the connected state (i.e. it has been answered).

Direction Application to network

Parameters

CallSessionID

Specifies the call session ID of the call.

CallLegType

Specifies the type of call leg created (e.g. generic or terminal, controlling or passive).

AppCallLeg

Specifies the application interface for callbacks from the call leg created.

Returns

CallLeg

Specifies the interface of the call leg created.

CallLegSessionID

Specifies the call leg session ID of the call leg created.

Errors -

Method `AttachCallLeg()`

This method requests that the call leg be attached to the call object. This will allow transmission on all associated bearer connections to other parties in the call. The call leg must be in the connected state for this method to complete successfully.

Direction Application to network

Parameters

CallSessionID

Specifies the call session ID of the call.

CallLeg

Specifies the interface of the call leg to attach to the call.

CallLegSessionID

Specifies the call leg session ID to attach to the call.

Returns -

Errors -

Method `detachCallLeg()`

This method requests that the call leg be detached from the call object. This will prevent transmission on any associated bearer connections to other parties in the call. The call leg must be in the connected state for this method to complete successfully.

Direction Application to network

Parameters

CallSessionID

Specifies the call session ID of the call.

CallLeg

Specifies the interface of the call leg to detach from the call.

CallLegSessionID

Specifies the call leg session ID to detach from the call.

Returns -

Errors -

Method `getControlLeg()`

This method requests the identification of the controlling call leg of this call.

Direction Application to network

Parameters **CallSessionID**

Specifies the call session ID of the call.

Returns **CallLeg**

Specifies the interface of the controlling call leg of this call.

CallLegSessionID

Specifies the call leg session ID of the controlling leg of this call.

Errors -

7.1.2.1.1State Diagram

Figure 11Figure 8 shows the state model for the generic call interface from applications point of view. The state model is simplified because most of the state is held within the associated call legs. The call is created by an application (via the `createCall()` method on the `CallManager` interface) or implicitly by the Generic Call Control Service when a new call event notification arrived.

It shall be noted that this state diagrams relates to the OSA interface and not to the underlying mechanism used to perform the call control.

<Editor's note: A mapping to CAMEL states might be needed in the stage 3>

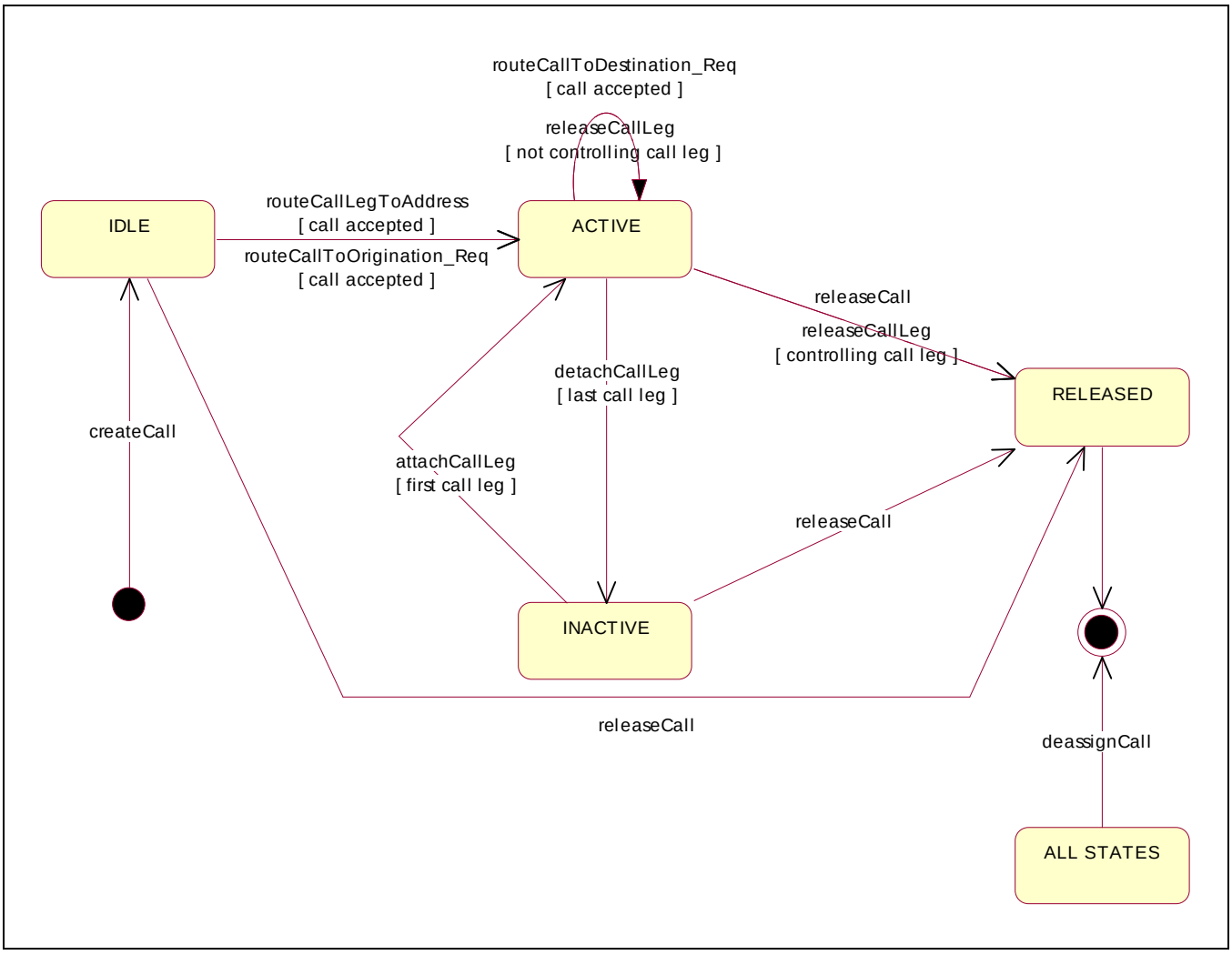


Figure 118 - State diagram for the `Call` interface from an application point of view

Sequence Diagrams

The following section will describe some scenario's to illustrate the use of the above described methods

Enable Call notification

The first task to perform in order to allow users to use applications is to enable users to reach that application. This is done with the method `enableCallNotification()` to be sent for every user subscribing to an applaiction.

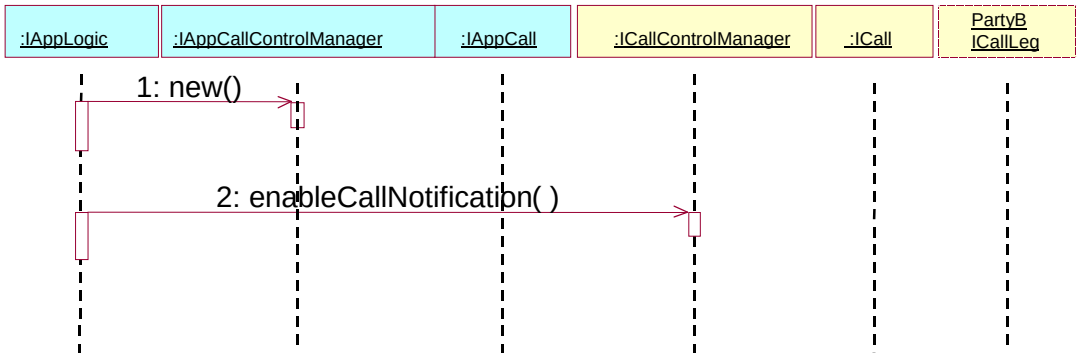


Figure 139 Enable call notification

Number translation

After the an user has subscribed to a service, it is possible to perform the needed actions. The example in Figure 14Figure 10 shows a simple number translation application.

After the call is triggered (according to the criteria in a previous `enableCallNotification()`), the SCS notifies the application with an `eventCallNotify()` message. This allows the application to perform the needed actions and continue the call set-up via a `routeCAllToDestination_Req()` message. The SCS relays the result of the call set-up (both positive and negative) to the application, which ends after that.

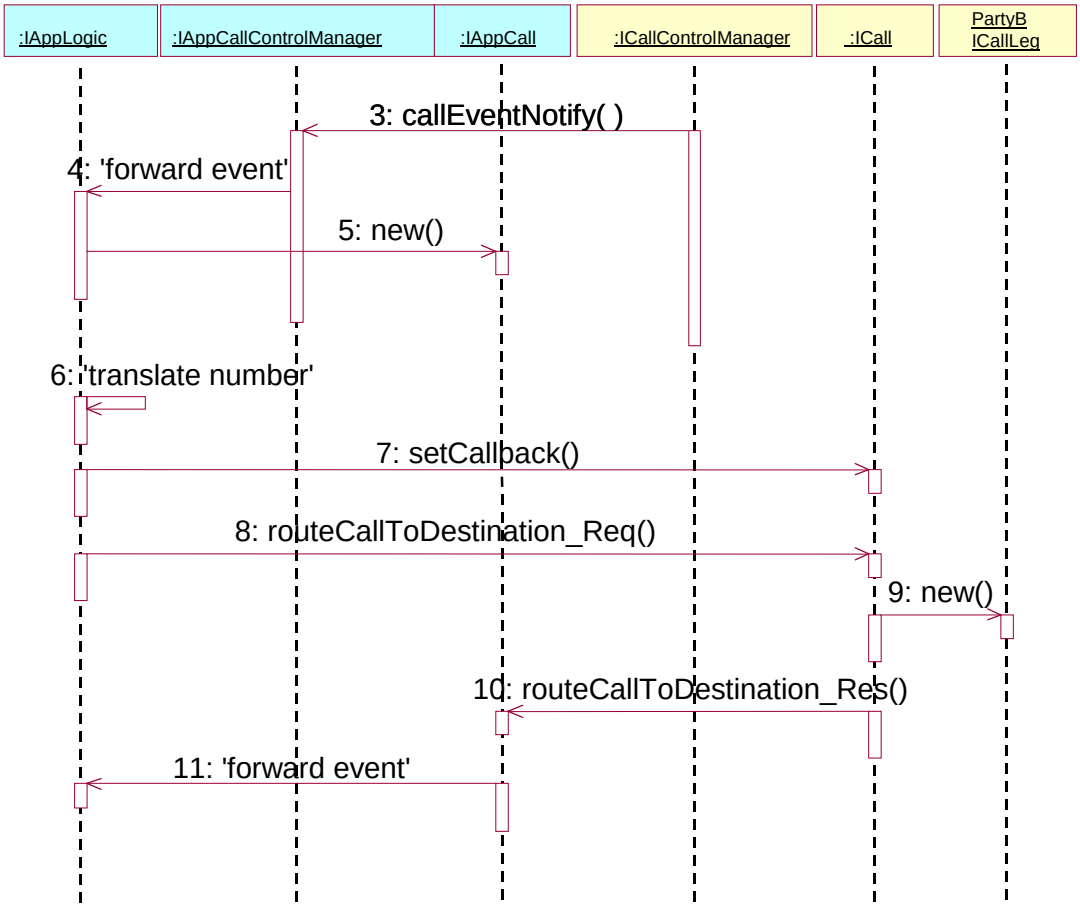


Figure 1410 Simple number translation

Call baring

The next example (Figure 16Figure 11) shows how a call baring application can be implemented:

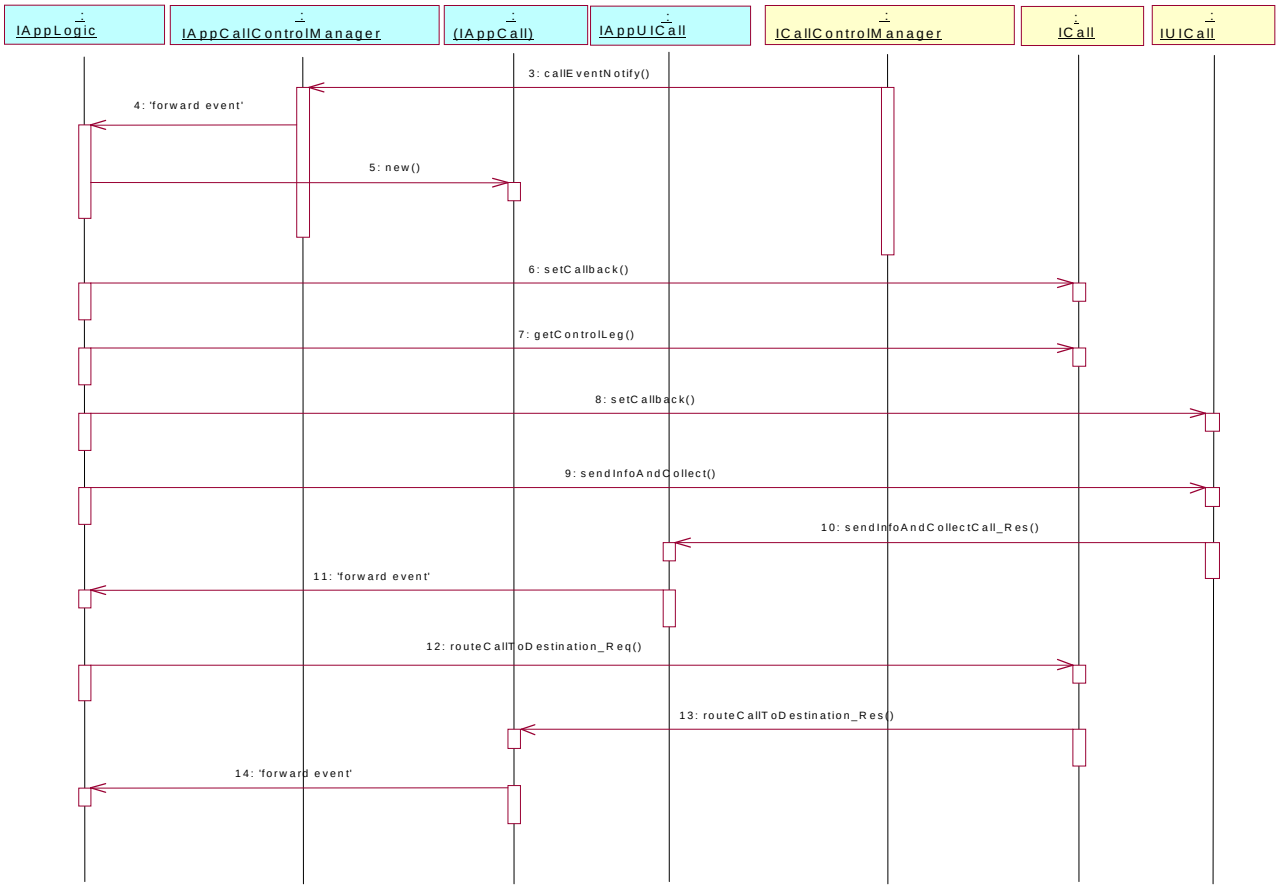


Figure 1611 Call baring application

Pre-paid with advice of charge

The next example () shows how a pre-paid application can be implemented:

With a pre-paid application it is the application that will determine the charging for the call. This means that the application will hold the whole tariffing scheme needed and needs to control the whole call. For the call shown the following condition apply:

- It is a long call
- Two tariff changes take place during the call.
- The application will inform the user about the applicable charging (the methods needed for this are described in sub-claus 7.10.1 CAMEL Call Leg7.10.1 CAMEL Call Leg).

After the application is triggered it sends a superviseCall Req() message indicating that the application will be responsible for charging the call. Before the call will be routed to the requested destination (8), the application sends the allowed time for the call (7) and informs the user about the charging applicable (using the Advice of Charge functionality in the core network) for this call (6). The information send exist of two sets of AoC information and a tariff switch. The application will be notified via the superviseCall Res() message if the tariff switch expired during the supervised period. This allows the application to send a new set of AoC information and a new tariff switch.

The application is notified of the expiration of the allowed time (9) and determines if the user has enough account left to continue with the call.

1 If there is enough account left a new time slot is allowed

2 Is there not enough account, the user will be notified and the call terminated after some time in order to allow the user to finish the call graciously.

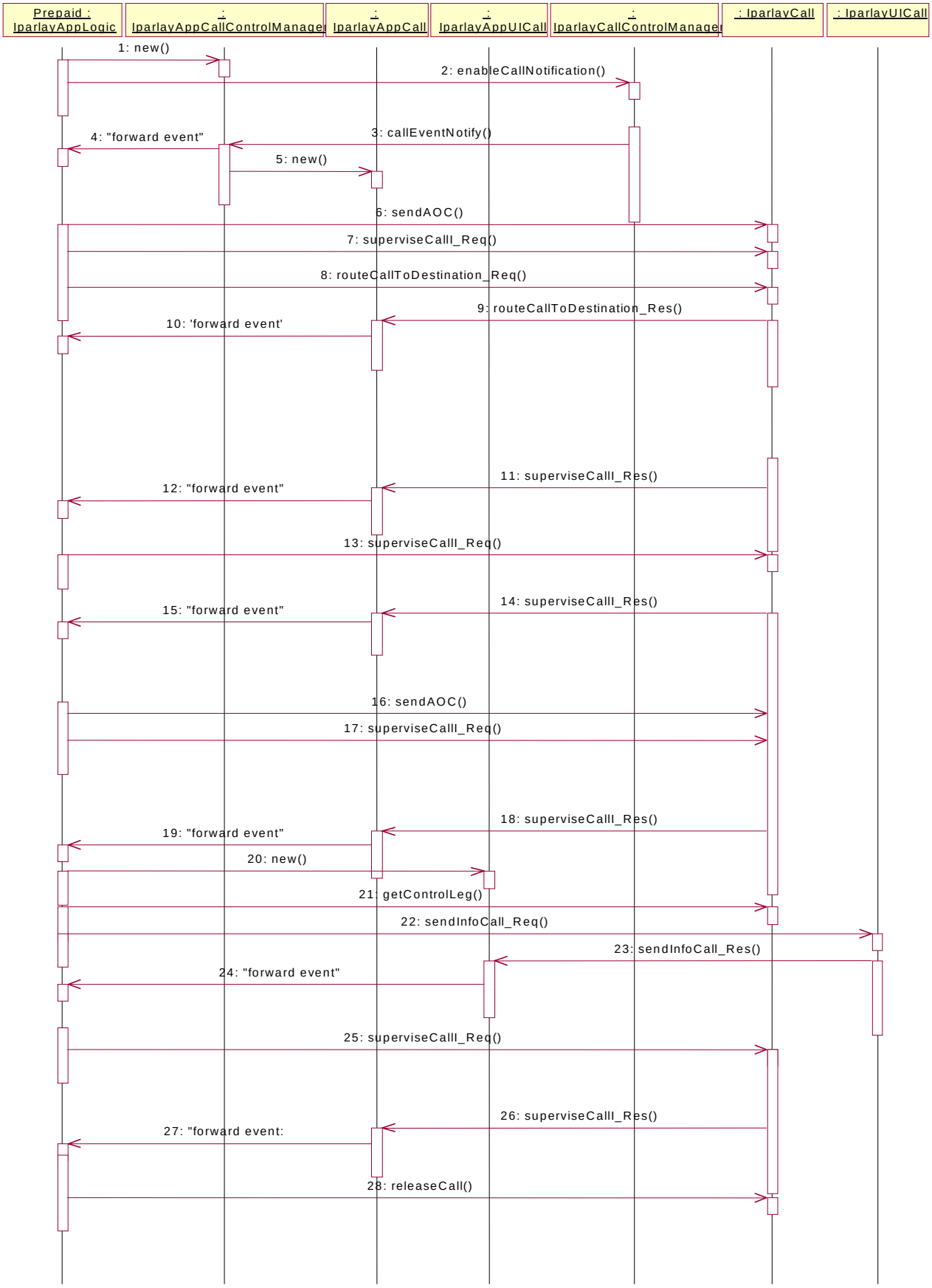


Figure 1812 Pre-paid with AoC

7.1.37.1.3 Call Leg

The generic call leg interface represents the logical call leg associating a call with an address. The call leg tracks it own states and allows charging summaries to be accessed.

Method `routeCallLegToAddress()`

This method initiates routing of the call leg to the given target address. The outcome of the call routing attempt can be requested and reported using `callLegEventReport_Req` and `callLegEventReport_Res / callLegEventReport_Err.`

Direction Application to network

Parameters

callLegSessionID
Specifies the call leg session ID of the call leg.

targetAddress
Specifies the destination party to which the call should be routed.

originatingAddress
Specifies the address of the originating (calling) party.

originalCalledAddress
Specifies the original address to which the call was initiated.

redirectingAddress
Specifies the last address from which the call was redirected.

appInfo
Specifies application-related information pertinent to the call (such as alerting method, tele-service type, service identities and interaction indicators).

Returns -

Errors -

Method `callLegEventReport_Req()`

This asynchronous method sets, clears or changes the criteria for the events that the call leg object will be set to observe.

Direction Application to network

Parameters

CallLegSessionID
Specifies the call leg session ID of the call leg.

EventReportsRequested
Specifies the events that the call leg object will observe and report.

Returns -

Errors -

Method `callLegEventReport_Res()`

This asynchronous method reports that an event has occurred that was requested to be reported (for example, a mid-call event, the party has requested to disconnect, etc.).

Direction Network to application

Parameters

CallLegSessionID

Specifies the call leg session ID of the call leg.

EventReport

Specifies the result of the request to route the call to the destination party. It also includes the mode that the call object is in, the call leg generating the report (if applicable) and other related information.

Returns -

Errors -

Method `CallLegEventReport_Err()`

This asynchronous method indicates that the request to manage call leg reports was unsuccessful, and the reason (for example, the parameters were incorrect, the request was refused, etc.).

Direction Network to application

Parameters

CallLegSessionID

Specifies the call leg session ID of the call leg.

Error

Specifies the error which led to the original request failing.

Returns -

Errors -

Method `getCallLegState()`

This method requests the current state of the call leg.

Direction Application to network

Parameters

callLegSessionID

Specifies the call leg session ID of the call leg.

Returns

state

Specifies the current state of the call leg.

Errors -

Method `getAddresses()`

This method requests the address details associated with the call leg.

Direction Application to network

Parameters

callLegSessionID

Specifies the call leg session ID of the call leg.

Returns

addressList

Specifies the addresses associated with the call leg.

Errors -

Method `getCallLegInfo_Req()`

This asynchronous method requests information associated with the call leg to be provided at the appropriate time (for example, to calculate charging). Note: in the call leg information must be accessible before the objects of concern are deleted.

Direction Application to network

Parameters

callLegSessionID

Specifies the call leg session ID of the call leg.

callLegInfoRequested

Specifies the call leg information that is requested.

Returns -

Errors -

Method `GetCallLegInfo_Res()`

This asynchronous method reports all the necessary information requested by the application, for example to calculate charging.

Direction Network to application

Parameters

CallLegSessionID

Specifies the call leg session ID of the call leg.

CallLegInfoReport

Specifies the call leg information requested.

Returns -

Errors -

Method `GetCallLegInfo_Err()`

This asynchronous method reports that the original request was erroneous, or resulted in an error condition.

Direction Network to application

Parameters

CallLegSessionID

Specifies the call leg session ID of the call leg.

Error

Specifies the error which led to the original request failing.

Returns -

Errors -

Method `getCallLegType()`

This method requests whether the call leg is a controlling or passive call leg.

Direction Application to network

Parameters

callLegSessionID

Specifies the call leg session ID of the call leg.

Returns

callLegType

Specifies the call leg type.

Errors -

Method `getCall()`

This method requests the call associated with this call leg.

Direction Application to network

Parameters

CallLegSessionID

Specifies the call leg session ID of the call leg.

Returns

Call

Specifies the interface of the call associated with this call leg.

CallSessionID

Specifies the call session ID of the call associated with this call leg.

Errors -

7.1.3.1.1State Diagram

Figure 19Figure 13 shows the state model for the generic call leg interface from an application point of view. This represents most of the call setup states. The call leg is created by an application (via the `createCallLeg()` method on the `Call` interface) or implicitly by the Generic Call Control Service.

It shall be noted that this state diagrams relates to the OSA interface and not to the underlying mechanism used to perform the call control.

<Editor's note: A mapping to CAMEL states might be needed in the stage 3>

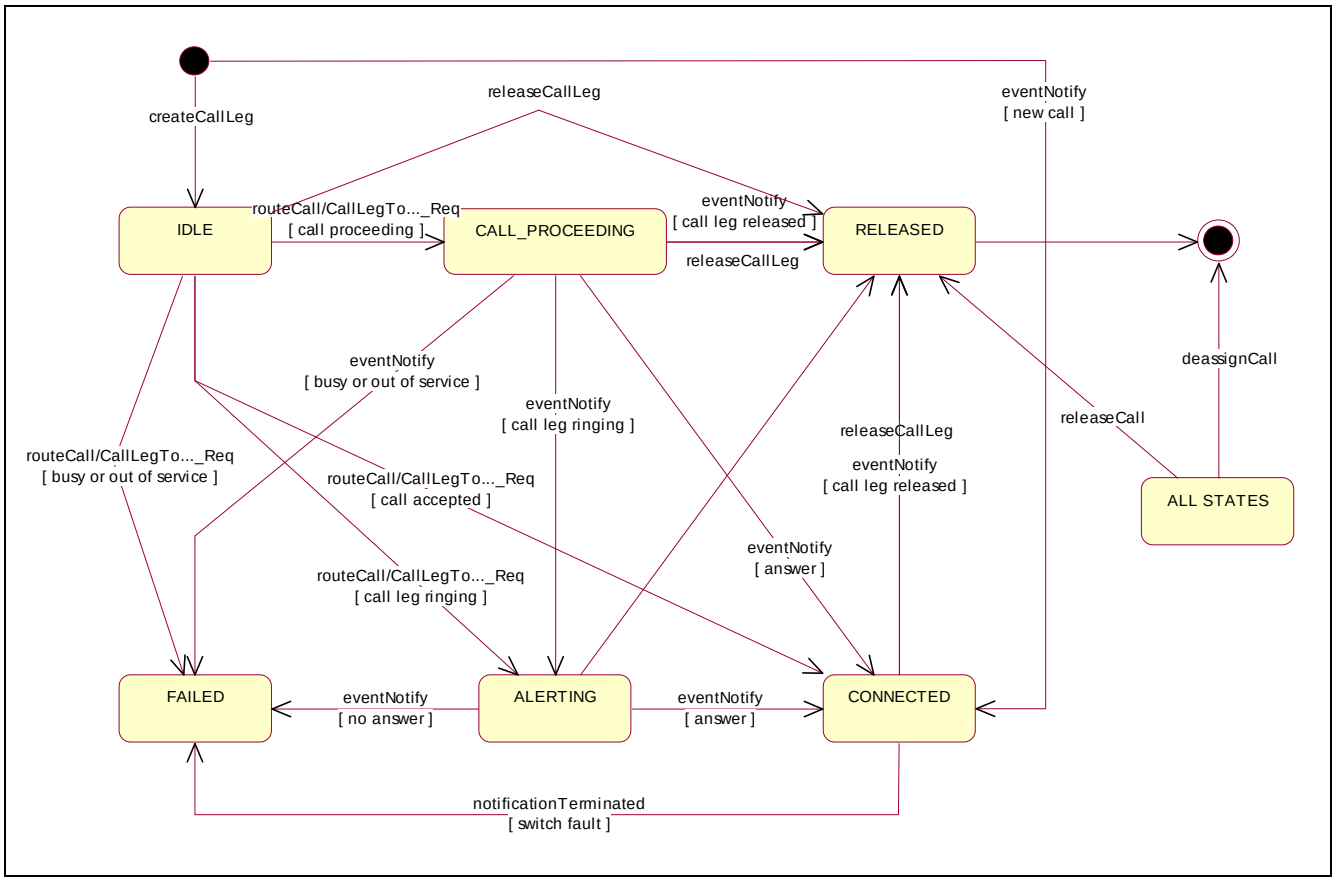


Figure 1913 - State diagram for the `CallLeg` interface from an application point of view

7.2Security/privacy

< It is proposed to delete this section, since security is ensured through the mechanisms provided by the Framework >

7.37.2 Address Translation

7.47.3 User Location

Method **`EnableLocationNotification()`**

This method is used to enable user status location notifications so that events can be sent to the application.

Direction Application to network

Parameters

AppInterface

If this parameter is set (i.e. not NULL) it specifies a reference to the application interface, which is used for callbacks. If set to NULL, the application interface defaults to the interface specified via the `setCallback()` method.

EventCriteria

Specifies the event specific criteria used by the application to define the event required.

Returns

AssignmentID

Specifies the ID assigned by the generic call control manager interface for this newly-enabled event

notification.

Errors -

Method `DisableLocationNotification()`

This method is used by the application to disable ~~call~~ location notifications.

Direction Application to network

Parameters

eventCriteria
Specifies the event specific criteria used by the application to define the event to be disabled.

assignmentID
Specifies the assignment ID given by the generic call control manager interface when the previous `enableNotification()` was called.

Returns -

Errors `INVALID_ASSIGNMENTID`

Returned if the assignment ID does not correspond to one of the valid assignment Ids.

Method `getUserLocation()`

This method is used by an application to get the location of a user directly.

Direction Application to network

Parameters

userIdentity

Identifies the user

Returns

locationInformation
Specifies the current location of the user. The following information elements may be returned:

- CellId or location area identifier
- VLR number
- Geographical information
- Location number

locationInformationAge
Indicates the time that the location information was updated.

Errors -

Method `locationReport()`

This method notifies the application of the arrival of a mobility-related event.

Direction Network to application

Parameters

eventInfo
Specifies data associated with this event.

assignmentID

Specifies the assignment id which was returned by the `enableNotification()` method. The application can use assignment id to associate events with event specific criteria and to act accordingly.

Returns -

Errors -

Method **`locationNotificationTerminated()`**

This method indicates to the application that all event notifications have been terminated (for example, due to faults detected).

Direction Network to application

Parameters -

Returns -

Errors -

~~7.5~~7.4 User Status

Method **`enableStatusNotification()`**

This method is used to enable user status notifications so that events can be sent to the application.

Direction Application to network

Parameters

appInterface

If this parameter is set (i.e. not NULL) it specifies a reference to the application interface, which is used for callbacks. If set to NULL, the application interface defaults to the interface specified via the `setCallback()` method.

eventCriteria

Specifies the event specific criteria used by the application to define the event required:

- Check for subscriber being reachable
- Check for subscriber being not reachable

Returns

assignmentID

Specifies the ID assigned by the generic call control manager interface for this newly-enabled event notification.

Errors -

Method **`disableStatusNotification()`**

This method is used by the application to disable ~~call~~ user status notifications.

Direction Application to network

Parameters

eventCriteria

Specifies the event specific criteria used by the application to define the event to be disabled.

assignmentID

Specifies the assignment ID given by the generic call control manager interface when the previous `enableNotification()` was called.

Returns -

Errors `INVALID_ASSIGNMENTID`

Returned if the assignment ID does not correspond to one of the valid assignment Ids.

Method `GetUserStatus()`

Direction Application to network

Parameters **`userIdentity`**

This parameter identifies the user for which the status has to be reported

Returns **status**

Reports the status of the user. The following status can be reported:

- Assumed idle
- Busy
- NotReachable
- No information available

Errors -

Method `statusEventNotify()`

This method notifies the application of the arrival of a call-related event.

Direction Network to application

Parameters **status**

Reports the status of the user.

assignmentID

Specifies the assignment id which was returned by the `enableNotification()` method. The application can use assignment id to associate events with event specific criteria and to act accordingly.

Returns -

Errors -

Method `statusNotificationTerminated()`

This method indicates to the application that all status event notifications have been terminated (for example, due to faults detected).

Direction Network to application

Parameters -

Returns -

Errors -

7.67.5 Terminal Capabilities

7.77.6 Message Transfer

7.6.17.6.1 Generic User Interaction

The Generic User Interaction class interface is used by applications to interact with end users. The API only supports Call User Interaction.

The class interface is represented by the `UIManager` and `UI` class interfaces, that interface to services provided by the network.

7.6.2Generic User Interaction Manager

Inherits from the generic service interface.

The Generic User Interaction Manager class interface provides the management functions to the Generic User Interaction class interface.

Method **`CreateUI()`**

This method is used to create a new user interaction object.

Direction Application to UserInteractionManager

Parameters **AppUI**
Specifies the application interface for callbacks from the user interaction created.

Returns **UserInteraction**
Specifies the interface of the user interaction created.

UserInteractionSessionID
Specifies the user interaction session ID of the user interaction created.

Errors

Method **`UserInteractionTerminated()`**

This method indicates to the application that the User Interaction service instance has terminated or closed abnormally. No further communication will be possible between the User Interaction service instance and application.

Direction GenericUserInteractionManager to Application

Parameters **UserInteraction**

Specifies the interface of the user interaction service that has terminated.

Returns

UserInteractionSessionID

Specifies the user interaction session ID of the user interaction that has terminated.

Errors

Method `UserInteractionFaultDetected()`

This method indicates to the application that a fault has been detected in the user interaction.

Direction UserInteractionManager to Application

Parameters

UserInteraction

Specifies the interface of the user interaction service in which the fault has been detected.

Returns

UserInteractionSessionID

Specifies the user interaction session ID of the user interaction in which the fault has been detected.

Fault

Specifies the fault that has been detected.

Errors

7.6.3Call User Interaction

Inherits from the Generic User Interaction class Iinterface. This interface can be used as a specialised version of the Generic User Interaction Interface.

The Call User Interaction class iInterface provides functions to send information to, or gather information from, the user (or call party) to which a call leg is connected. An application can use the Call User Interaction Serviceclass iInterface only in conjunction with another serviceclass interface which provides mechanisms to connect a call leg to a user. At present, only the Call Control serviceclass interface supports this capability.

Method `SendInfoCall_Req()`

This asynchronous method plays an announcement or sends other information to the user.

Direction Application to CallUserInteraction

Parameters

UserInteractionSessionID

Specifies the user interaction session ID of the user interaction.

CallLegSessionID

Specifies the call leg session ID on which to perform the user interaction. If the value of this parameter is `NULL`, then the controlling call leg is assumed.

Some implementations may impose restrictions on which legs can be used for an user interaction.

InfoID

Specifies the ID of the information to send to the user.

ResponseRequested

Specifies if a response is required from the call user interaction service, and any action the service should take.

Returns

Errors

Method `SendInfoCall_Res`

This asynchronous method informs the application about the start or the completion of a `sendInfoCall_Req()`. This response is called only if the `responseRequested` parameter of the `sendInfoCall_Req()` method was set to `UICALL_RESPONSE_REQUIRED`.

Direction CallUserInteraction to Application

Parameters

UserInteractionSessionID

Specifies the user interaction session ID of the user interaction.

CallLegSessionID

Specifies the call leg session ID of the call leg associated with the send info operation. If the value of this parameter is NULL, then the controlling call leg is assumed.

Response

Specifies the type of response received from the user.

Returns

Errors

Method `SendInfoCall_Err`

This asynchronous method indicates that the request to send information was unsuccessful.

Direction CallUserInteraction to Application

Parameters

UserInteractionSessionID

Specifies the user interaction session ID of the user interaction.

CallLegSessionID

Specifies the call leg session ID of the call leg associated with the send info operation. If the value of this parameter is NULL, then the controlling call leg is assumed.

Error

Specifies the error which led to the original request failing.

Returns

Errors

Method `SendInfoAndCollectCall_Req()`

This asynchronous method plays an announcement or sends other information to the user and collects some information from the user. The announcement usually prompts for a number of characters (for example, these are digits or text strings such as "YES" if the user's terminal device is

a phone).

Direction Application to CallUserInteraction

Parameters

UserInteractionSessionID

Specifies the user interaction session ID of the user interaction.

CallLegSessionID

Specifies the call leg session ID on which to perform the user interaction. If the value of this parameter is `NULL`, then the controlling call leg is assumed.

Some implementations may impose restrictions on which legs can be used for an user interaction.

InfoID

Specifies the ID of the information to send to the user.

Criteria

Specifies additional properties for the collection of information, such as the maximum and minimum number of characters, end character, first character timeout and inter-character timeout.

Returns

Errors

Method `sendInfoAndCollectCall_Res`

This asynchronous method returns the information collected to the application.

Direction CallUserInteraction to Application

Parameters

UserInteractionSessionID

Specifies the user interaction session ID of the user interaction.

CallLegSessionID

Specifies the call leg session ID of the call leg associated with the send info and collect operation. If the value of this parameter is NULL, then the controlling call leg is assumed.

Response

Specifies the type of response received from the user.

Info

Specifies the information collected from the user.

Returns

Errors

Method `sendInfoAndCollectCall_Err`

This asynchronous method indicates that the request to send information and collect a response was unsuccessful.

Direction CallUserInteraction to Application

Parameters

UserInteractionSessionID

Specifies the user interaction session ID of the user interaction.

CallLegSessionID

Specifies the call leg session ID of the call leg associated with the send info and collect operation. If the value of this parameter is NULL, then the controlling call leg is assumed.

Error

Specifies the error which led to the original request failing.

Returns

Errors

Method `releaseUICall()`

This method requests the release of the call user interaction. The user interaction call service interrupts the current action on all associated call legs.

Direction Application to CallUserInteraction

Parameters

UserInteractionSessionID

Specifies the user interaction session ID of the user interaction.

Returns

Errors

Method `AbortLegAction_Req()`

This asynchronous method aborts a user interaction operation, e.g. a `sendInfoCall_Req()`, from the specified call leg. The call and call leg are otherwise unaffected. The user interaction call service interrupts the current action on the specified leg.

Direction Application to CallUserInteraction

Parameters

UserInteractionSessionID

Specifies the user interaction session ID of the user interaction.

CallLegSessionID

Specifies the call leg session ID on which to abort the user interaction and disconnect from the user interaction. If the value of this parameter is NULL, then the controlling call leg is assumed.

Returns

Errors

Method `abortLegAction_Res()`

This asynchronous method confirms that the request to abort a user interaction operation on a call leg was successful.

Direction CallUserInteraction to Application

Parameters

UserInteractionSessionID

Specifies the user interaction session ID of the user interaction.

CallLegSessionID

Specifies the call leg session ID of the call leg associated with the abort user interaction operation. If the value of this parameter is NULL, then the controlling call leg is assumed.

Returns

Errors

Method **`abortLegAction_Err()`**

This asynchronous method indicates that the request to abort a user interaction operation on a call leg resulted in an error.

Direction CallUserInteraction to Application

Parameters

UserInteractionSessionID

Specifies the user interaction session ID of the user interaction.

CallLegSessionID

Specifies the call leg session ID of the call leg associated with the abort user interaction operation. If the value of this parameter is NULL, then the controlling call leg is assumed.

Error

Specifies the error which led to the original request failing.

Returns

Errors

~~7.8~~7.7 Data Download

~~7.9~~7.7 User Profile Management

<Editor's note: management of supplementary service data might be included here (or in a separate section it that appears to be more appropriate>

~~7.10~~7.10 Charging

~~7.10.1~~7.10.1 CAMEL Call Leg

This class inherits from the Call Leg interface class and adds CAMEL specific methods.

Method **`setAdviceOfCharge()`**

This method allows the application to the charging information that will be send to the end-users handset.

Direction Application to network

Parameters **CallLegSessionID**

Specifies the call leg session ID of the call leg.

AdviceOfChargeInformation

Specifies two sets of Advice of Charge parameter according to GSM …..

TariffSwitch

Specifies the tariff switch that signifies when the second set of AoC parameters becomes valid.

Returns -

Errors -

Annex A - Relation between OSA interface class methods and ~~CAMEL operations~~MAP/CAP information flows (informative)

The table below shows how OSA interface class methods can be mapped onto ~~CAMEL~~ MAP/CAP ~~Phase3 operations~~information flows. Note that the table below does not contain Framework interface classes. In some cases there is no mapping between OSA interface classes and ~~CAP operations~~information flows, but between OSA interface classes and messages generated by ~~TCAP, the protocol layer below CAP~~lower protocol layers.

<Editor's note: the mapping might slighty change due to CAMEL phase3 scope changes>

<Editor's note: mapping on parameter level might be done as part of stage3 >

OSA interface class method	**~~CAP operation (phase3)~~MAP/CAP information flow**
CreateCall	No CAP operation, since execution of this method only results in the creation of a Call object instance in the Service Capability Server
EnableCallNotification	AnyTimeModification Note: AnyTimeModification only allows for activation of O/T-CSI, not for the creation
DisableCallNotification	AnyTimeModification Note: AnyTimeModification only allows for de-activation of O/T-CSI, not for the deletion
CallAborted	~~TCAP U-ABORT or Disconnect event~~Protocol abort
CallFaultDetected	All "no connection" events that refer to an error rather than to a normal "no connection" situation (like e.g. "busy", "no answer")
CallEventNotify	InitialDP
CallNotificationTerminated	-
RouteCallToDestination_Req	(InitiateCallAttempt and Reconnect) or Connect or Continue and RequestReport_BCSM (in case the application needs to be notified on certain events)
RouteCallToDestination_Res	EventReport_BCSM
RouteCallToDestination_Err	~~TCAP Return Error~~Protocol error

OSA interface class method	~~CAP operation (phase3)~~MAP/CAP information flow
RouteCallToOrigination_Req	InitiateCallAttempt and RequestReport_BCSM (in case the application needs to be notified on certain events)
RouteCallToOrigination_Res	EventReport_BCSM
RouteCallToOrigination_Err	~~TCAP Return Error~~Protocol error
ReleaseCall	ReleaseCall
DeassignCall	Cancel
GetCallInfo_Req	CallInformationRequest
GetCallInfo_Res	CallInformationReport
GetCallInfo_Err	~~TCAP Return Error~~Protocol error
GetCallState	Internal method~~-~~
GetCallLegs	Internal method~~-~~
CreateCallLeg	Internal method~~-~~
AttachCallLeg	Internal method~~-~~
DetachCallLeg	Internal method~~-~~
GetControlLeg	Internal method~~-~~
RouteCallLegToAddress	(InitiateCallAttempt and Reconnect) or Connect or Continue and RequestReport_BCSM (in case the application needs to be notified on certain events)
CallLegEventReport_Req	RequestReport_BCSM
CallLegEventReport_Res	EventReport_BCSM
CallLegEventReport_Err	~~TCAP Return Error~~Protocol error
GetCallLegState	Internal method~~-~~
GetAddresses	Internal method~~-~~
GetCallLegInfo_Req	CallInformationRequest
GetCallLegInfo_Res	CallInformationReport
GetCallLegInfo_Err	~~TCAP Return Error~~Protocol error
GetCallLegType	Internal method~~-~~
GetCall	Internal method~~-~~
EnableLocationNotification	AnyTimeModification Note: AnyTimeModification only allows for activation of M-CSI, not for the creation
DisableLocationNotification	AnyTimeModification Note: AnyTimeModification only allows for de-activation of M-CSI, not for the deletion
GetUserLocation	AnyTimeInterrogation

OSA interface class method	CAP operation (phase3)MAP/CAP information flow
LocationReport	LocationUpdate [check latest version CAP phase3]
LocationNotificationTerminated	? [check latest version CAP phase3]
EnableStatusNotification	AnyTimeModification Note: AnyTimeModification only allows for activation of M-CSI, not for the creation
DisableStatusNotification	AnyTimeModification Note: AnyTimeModification only allows for de-activation of M-CSI, not for the deletion
StatusNotificationTerminated	? [check latest version CAP phase3]
SetChargeInfo	FurnishChargingInformation
SetAdviceOfCharge	SendChargingInformation
SuperviseCall_Req	ApplyCharging
SuperviseCall_Res	ApplyCharging_Response
SuperviseCall_Err	TCAP Return ErrorProtocol error

9Annex B - Example of use of OSA (informative)

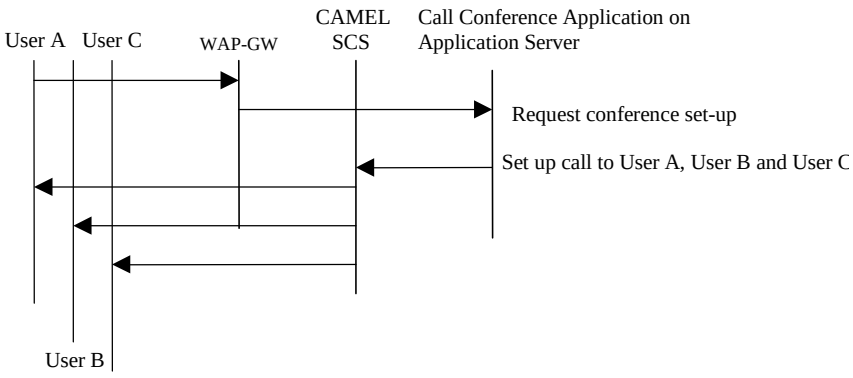
The following example shows how the OSA, described on a high level, could be used to execute an application. Note that OSA enables the use of various Service Capability Servers by an application.

A user participates in a three-party conference discussing where to dine tonight (voice call). The Web is browsed to pick a suitable restaurant. The location of the restaurant in relation to the position of the user is displayed on the user's terminal (location/positioning application). The restaurant choice is agreed amongst the conference participants and a voice call is set up to book a table at the restaurant.

In some more detail this example could look as follows:

1) Conference call set-up:

Ordered via a WAP communication to the Call Conference Application.



2) Web-browsing:

User A uses WAP to browse information on the Internet (or a specific Home Environment provided service) to find a restaurant guide.

User A → WAP-GW → Internet.

After having found two restaurant choices which all conference participants agree to, it is decided to choose the one that is closest to where User A is located. User A then contacts the "Locator Application" which helps him to decide which of the two restaurants that is closest to him, and how to get there.

3) Location/Positioning info:

User A → "Locator Application" → CAMEL SCS (positioning part).

The "Locator Application" determines which of the two restaurants that is the closest, translates the positioning information into a description how to get there, either in text for WAP or alternatively included as an attachment in an e-mail.

4) Table reservation:

Lastly, User A makes a table reservation by calling the restaurant. This can either be done via the WTAI interface between a WTA application or by requesting an application in the network to establish the call as described in 1. above.

User A → Voice call controlled via WTAI → Restaurant"

10History

Date	Version	Comment
July 1999	0.1.0	Initial Draft produced in Hazlet, New Jersey, USA
September 1999	0.2.0	Version presented to S2 plenary in Bonn, Germany (not including all agreed changes yet from VHE/OSA adhoc session)
September 1999	0.2.1	Output of Bonn meeting (Presented to SA for information, since V1.0.0 was not available due to 3GPP e-mail exploder problems).
October 1999	0.3.0	Version sent to S2 e-mail list and proposed to send to SA plenary
October 1999	0.3.1	Small editorial updates in Interface section (subclauses 6 and 7)
October 1999	1.0.0	Version 0.3.1 raised to V1.0.0 by S2 e-mail approval and sent to SA e-mail list for information
November 1999	1.1.0	Updated after comments in S2#9 according to S2-99C06 (S2-99B32, S2-99B33 and S2-99B36)

Rapporteur: Erwin van RijssenRob Schmersel, Ericsson

Email: Erwin.van.RijssenRob Schmersel@eraetm.ericsson.se Telephone: + 46 8 75.71.30231 161 24.26.02