3GPP - TSG SA #5 **Tdoc SP-99409**
11-13 October, 1999
Kyongju, Korea

Title: 3G TS 23.127 v. 0.2.1 and v.0.3.0:

Virtual Home Environment (VHE)/Open Service Architecture (OSA) - Stage 2

Date: 1999-10-06

Source: S2

Purpose: For information

Agenda Point: 5.2.3

The attached document contains two versions of the 3G TS 23.171: *Virtual Home Environment (VHE)/Open Service Architecture (OSA) - Stage 2*.

v.0.2.1 is the last approved version by S2.
v.0.3.0 is the version which should have been approved by correspondence last week to be presented as v.1.0.0 at SA#5 meeting. However, due to some technical reasons, this was not performed. S2 will now provide version 1.0.0 before end of October and send it to SA e-mail reflector.

3G TS 23.127 v0.3.0 (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 0.3.0)

Reference

DTS/TSGS-0223127U

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 x.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;

1 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).

2 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.

2.1 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] GSM 03.57: " Digital cellular telecommunication system (Phase 2+); Mobile Station Application Execution Environment (MExE); Service description - Stage2”

[4] GSM 02.78: " Digital cellular telecommunication system (Phase 2+); Customised Applications for Mobile network Enhanced Logic (CAMEL); Service definition - Stage 1" [5] GSM 03.78: ”Digital cellular telecommunication system (Phase 2+); Customised Applications for Mobile network Enhanced Logic (CAMEL); Service 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: “….

2.2 Informative references

[1] UMTS TR 22.70: ”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)

3 Definitions and abbreviations

3.1 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 application 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.

3.2 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.

4 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.

5 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 featuresand 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

5.1 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 functionalitymight 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 interface is specified in terms of a number of interface classes. 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 application using the OSA interface, since these applications are executed in Application Servers.

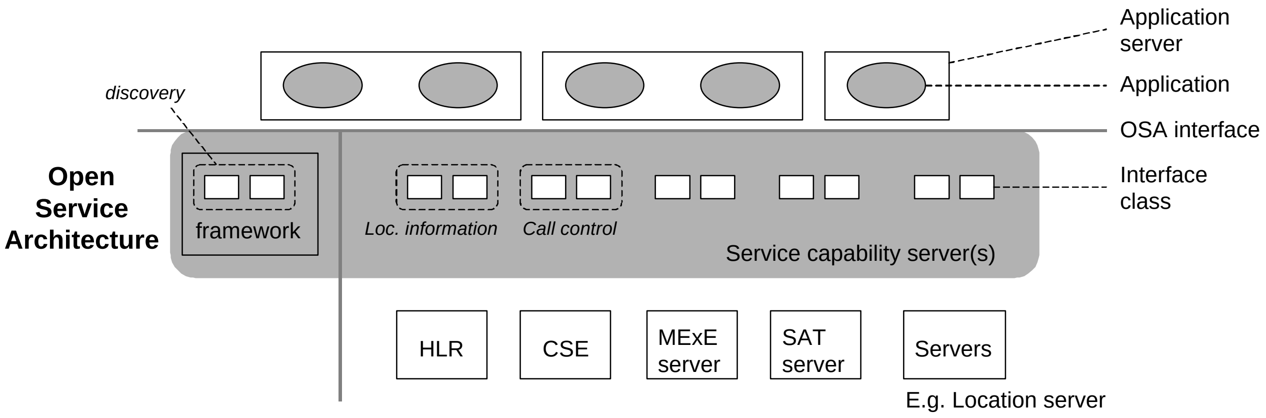


Figure 1 Overview of Open Service Architecture

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 2 shows the case where the OSA interface classes are implemented in one physical entity, called “gateway” in the figure. Figure 3 shows the case where the SCSs are distributed across several ‘gateways’.

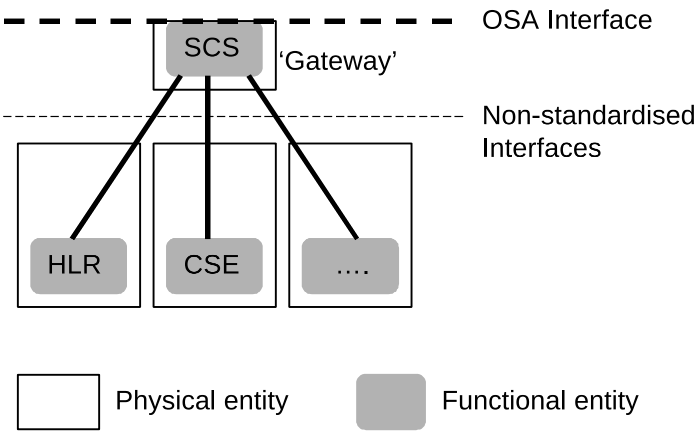


Figure 2 SCSs and network functional entities implemented in separate physical entities

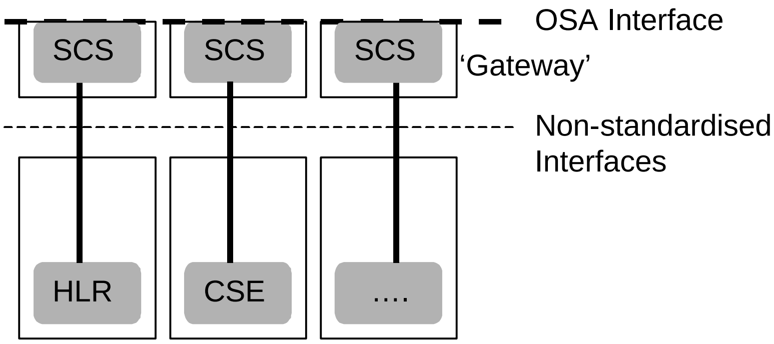


Figure 3 SCSs and network functional entities implemented in separate physical entities, SCSs distributed across several 'gateways'.

__**Option 2**__

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

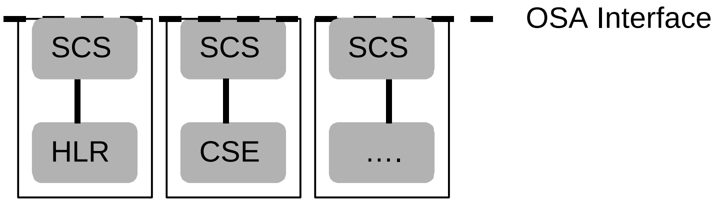


Figure 4 SCSs and network functional entities implemented in same physical entities

__**Option 3**__

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

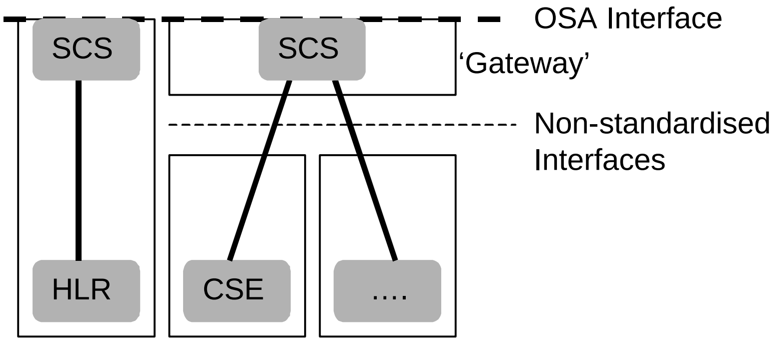


Figure 5 Hybrid implementation (combination of option 1 and 2)

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 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 notificationevent is in this case requested on the Calling and/or Called Party Number of the user.

5.3 Base interface classes

The base class interfaces 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 (ie functionality) are added by each class further in the hierarchy.

5.3.1 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

5.3.2 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

6 Framework service capability features

6.1 Authentication

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.

6.1.1 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.

< 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

6.1.2 Authentication interface class

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 `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

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 `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

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 Framework to application

Parameters **challenge**

The challenge presented by the application to be responded to by the framework. The challenge mechanism used will be in accordance with 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 `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 Framework to application

Parameters

terminationTex

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 `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 `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 `terminationText`. If a match is made, the service agreement is terminated, otherwise an error is returned.

Returns

Errors

Direction Application to framework

Method `obtainFrameworkInterface()`

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

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

6.2 Authorisation

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

6.3 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

6.4 Registration

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

6.5 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

7 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.

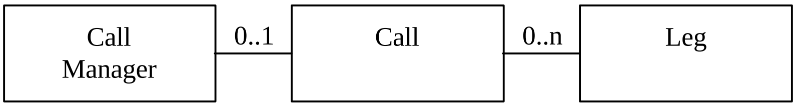


Figure 6 Call control class hierarchy

7.1.1 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 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 **`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.

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 `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 `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 `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.

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 `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

7.1.2 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, 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.

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 `routeCallToOrigination_Req()`

This asynchronous method requests routing of a call to the first call party, via a controlling call leg (which is implicitly created). The call object must already have been 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 `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 `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 `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 `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.

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

7.1.2.1.1 State Diagram

Figure 7 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>

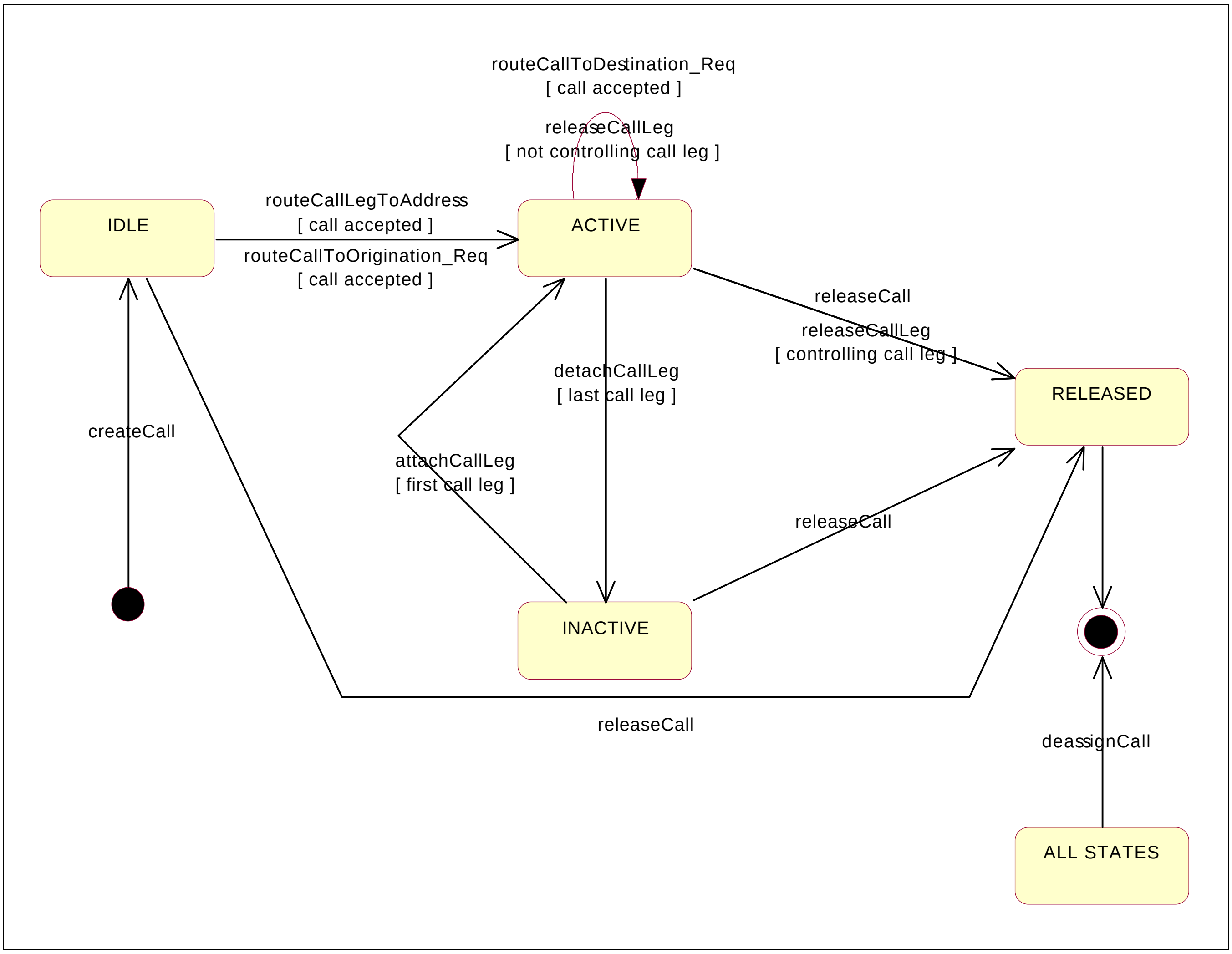


Figure 7 - State diagram for the `Call` interface from an application point of view

.

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_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 `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

7.1.3 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 `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 `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

7.1.3.1.1 State Diagram

Figure 8 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>

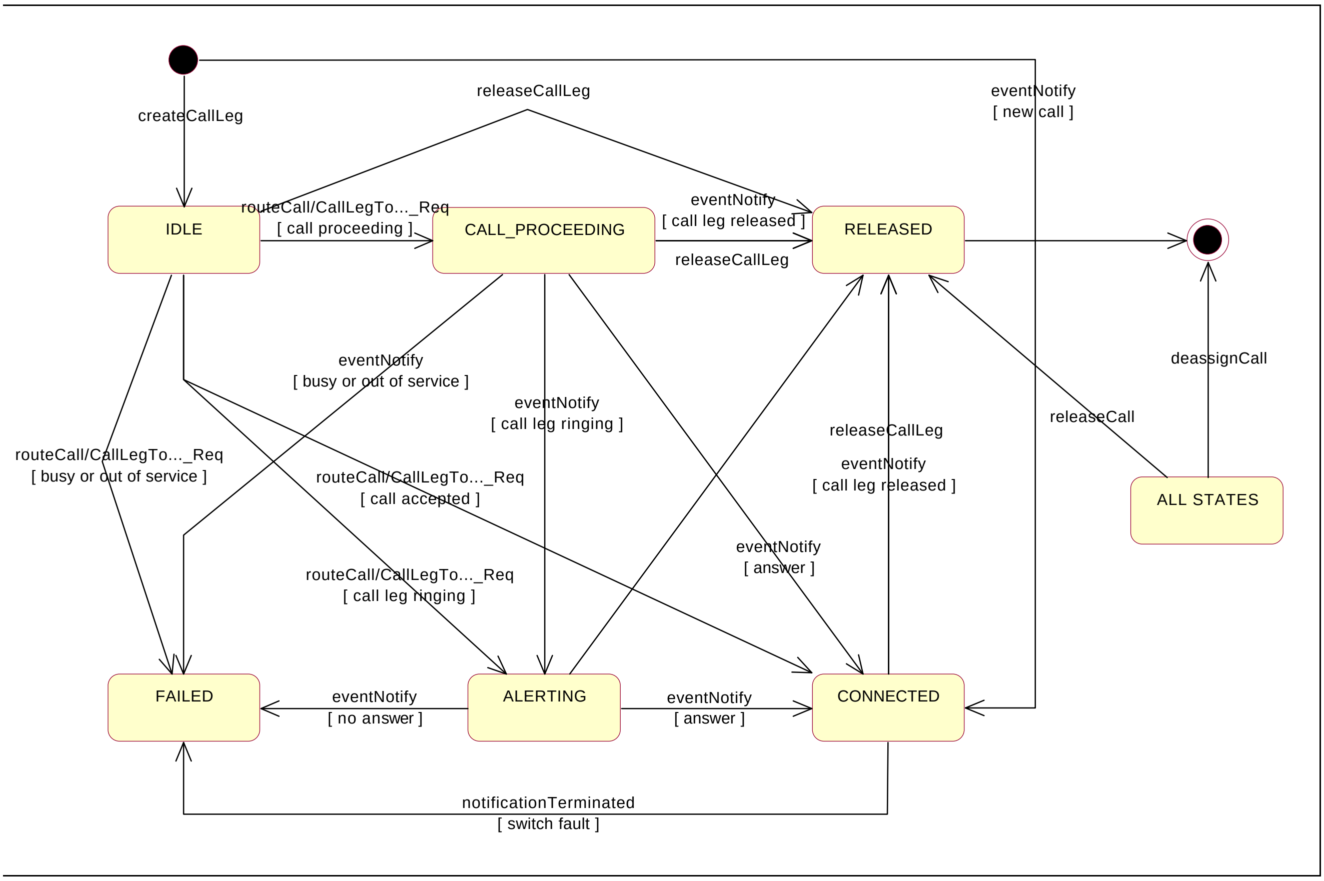


Figure 8 - State diagram for the `CallLeg` interface from an application point of view

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 **`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

7.2 Security/privacy

7.3 Address Translation

7.4 User Location

Method **`EnableLocationNotification()`**

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.

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 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 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 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

7.6 Terminal Capabilities

7.7 Message Transfer

7.8 Data Download

7.9 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 Charging

7.10.1 CAMEL Call Leg

This class inherits from the base 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

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

8 Annex - Relation between OSA interface class methods and CAMEL operations (informative)

The table below shows how OSA interface class methods can be mapped onto CAMEL CAP Phase3 operations. 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, but between OSA interface classes and messages generated by TCAP, the protocol layer below CAP.

<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)**
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
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
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
ReleaseCall	ReleaseCall
DeassignCall	Cancel
GetCallInfo_Req	CallInformationRequest

OSA interface class method	**CAP operation (phase3)**
GetCallInfo_Res	CallInformationReport
GetCallInfo_Err	TCAP Return Error
GetCallState	-
GetCallLegs	-
CreateCallLeg	-
AttachCallLeg	-
DetachCallLeg	-
GetControlLeg	-
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
GetCallLegState	-
GetAddresses	-
GetCallLegInfo_Req	CallInformationRequest
GetCallLegInfo_Res	CallInformationReport
GetCallLegInfo_Err	TCAP Return Error
GetCallLegType	-
GetCall	-
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
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

OSA interface class method	CAP operation (phase3)
StatusNotificationTerminated	? [check latest version CAP phase3]
SetChargeInfo	FurnishChargingInformation
SetAdviceOfCharge	SendChargingInformation
SuperviseCall_Req	ApplyCharging
SuperviseCall_Res	ApplyCharging_Response
SuperviseCall_Err	TCAP Return Error

9 Annex - Example of use of OSA (informative)

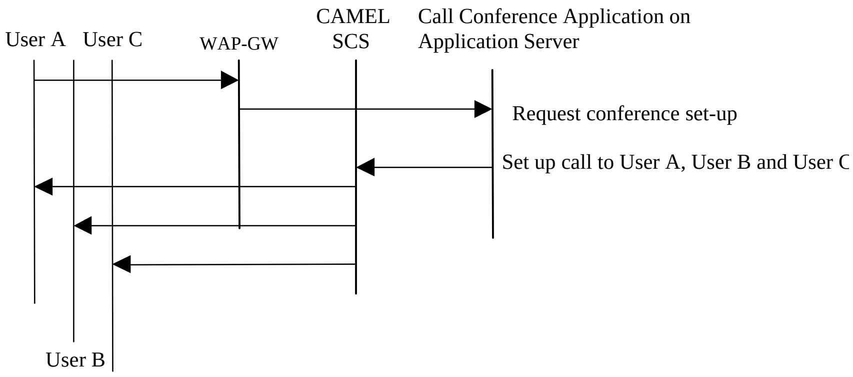
The following example shows how the OSA, described on a high level, could be used to execute an application. Note that OSA enables the use of various Service Capability Servers by an application.

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.


User A
User C
WAP-GW
CAMEL SCS
Call Conference Application on Application Server
Request conference set-up
Set up call to User A, User B and User C
User B

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”

10 History

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
October 1999	0.3.0	Version sent to S2 e-mail list and proposed to send to SA plenary

Rapporteur: Erwin van Rijssen, Ericsson

Email: Erwin.van.Rijssen@etm.ericsson.se Telephone: + 31 161 24.26.02

3G TS 23.xyz 0.~~1~~2.0 (1999-09~~7~~)

Technical Specification

3rd Generation Partnership Project;
Technical Specification Group Services and System Aspects;
Virtual Home Environment / Open Service Architecture
(3G TS 23.xyz version 0.2~~1~~.1~~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;

1 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, CAMEL, MExE and SAT are considered the network mechanisms supporting VHE.

The Open Service Architecture (OSA) defines an architecture that enables applications to make use of network functionality through an open standardised interface (the OSA~~Application~~ 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 centric applications or client/server applications with clients residing in the Mobile Station (MS). Examples of the latter case are MExE and SAT.

A key feature to support VHE is the ability to build services using the OSA~~ Application~~ Interface.

2 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.

2.1 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] GSM 03.57: " Digital cellular telecommunication system (Phase 2+); Mobile Station Application Execution Environment (MExE); Service description - Stage2”

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

[5~~3~~] GSM 03.78: ”Digital cellular telecommunication system (Phase 2+); Customised Applications for Mobile network Enhanced Logic (CAMEL); Service definition - Stage 2”

[6~~4~~] 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~~5~~] UMTS TS 22.101: ”Universal Mobile Telecommunications System (UMTS): Service Aspects; Service Principles”

[8~~6~~] UMTS TS 22.105: ”Universal Mobile Telecommunications System (UMTS); Services and Service Capabilities”

[9~~7~~] UMTS TS 22.121: ”Universal Mobile Telecommunications System (UMTS); Virtual Home Environment”

2.2 Informative references

[1] UMTS TR 22.70: ”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)

3 Definitions and abbreviations

3.1 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 ~~application~~ OSA interface

Service Capability Server: ~~service capabilities like CAMEL, HLR, MexE server etc.~~Entity providing OSA interfaces towards an application

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

Applications~~/ Clients~~: ~~These are s~~Services, which are designed using service capability features.

~~**Interfaces:**~~

~~**-application ~**~~

~~**-framework ~**~~

~~**-service ~**~~

~~**Application**~~ **OSA Interface:** Standardised Interface used by application/clients to access service capability features. ~~There are 3 different interfaces that are distinguished:~~

~~**-application interface:** This is the interface as send from the application~~

~~**-framework interface:** This is the interface towards the OSA Framework~~

- ~~**service interface:** This is the interface as send from the Service Capability Servers~~

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.

3.2 Abbreviations

For the purposes of this TS the following abbreviations apply:

~~AI~~	~~Application Interface (prefix to interface class method)~~
CAMEL	Customised Application For Mobile Network Enhanced Logic
CSE	Camel Service Environment
~~FI~~	~~Framework Interface (prefix to interface class method)~~
HE	Home Environment
HE-VASP	Home Environment Value Added Service Provider
HLR	Home Location Register
~~LCS~~	~~LoCation Services~~
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
SIM	Subscriber Identity Module Short Message Service
~~SI~~	~~Service Interface (prefix to interface class method)~~
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 UMTS TS 22.01.

~~4Introduction to VHE and OSA~~

~~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.~~

~~For service continuity reasons (evolution from 2G systems), VHE encompasses both 2G services (including 2G supplementary services) and, all the new services/applications which will be implemented using standardised service capability servers. For Release '99 the service capability servers are gsmSCF, MExE and SAT and access to. As regards MExE, with subparts WAP and Java, the standardisation is the responsibility of the WAP Forum (Java is for further study). In Release '99 the new 3G services will be executed in the Home Environment domain also when roaming.. 2G supplementary services are executed according to GSM roaming principles.~~

~~In order to implement the VHE, including also today not known end user services/applications, a highly flexible Open Service Architecture (OSA) is required. It would interface the core networks with all the legacy features, and connect to the fast moving applications world.~~

~~To make it possible for operators and 3rd party application developers, possibly with no telecom competence, 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 CORBA interfaces. The service capability servers will serve as gateways between the legacy systems and the applications environment.~~

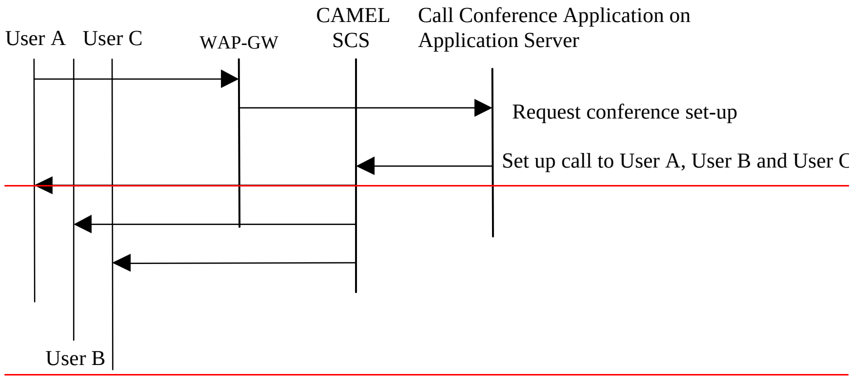
~~The following example shows how the OSA, described on a high level, could be used to execute an application. Note that OSA enables the use of various Service Capability Servers by an application.~~

~~*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.~~


User A
User C
WAP-GW
CAMEL
SCS
Call Conference Application on
Application Server
Request conference set-up
Set up call to User A, User B and User C
User B

~~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"~~

~~5~~4 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~~7~~] for more details.

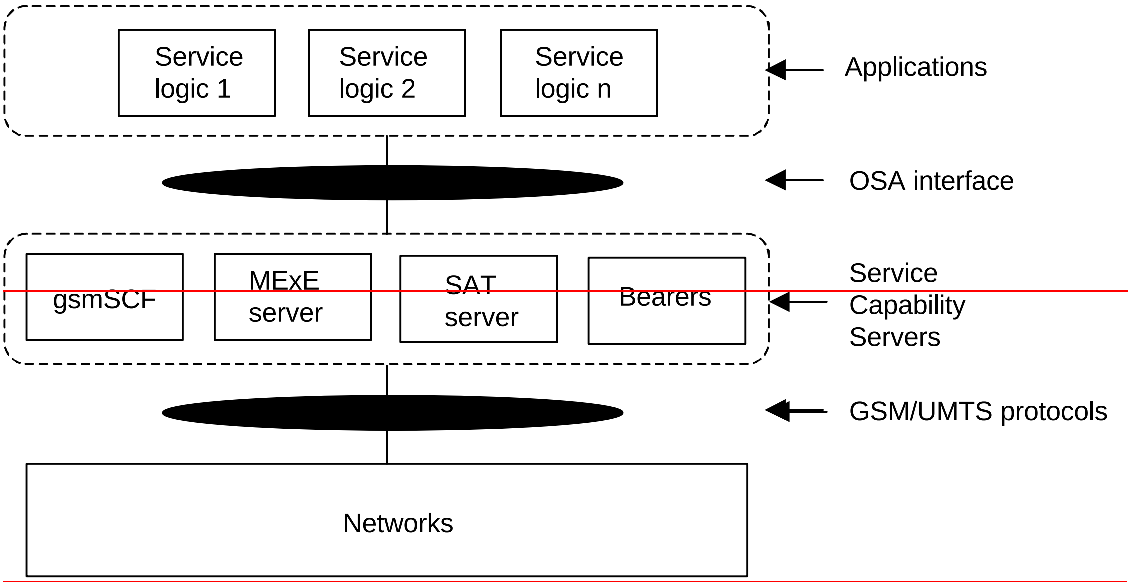
~~The moving out of the applications from the core networks leads to a preference to locate these user profiles in a common database, where they are easily accessible from both the core network functions and the applications. The database is located in the HE domain."~~

<Editor's note: improve the text on 2G and 3G services; text to be provided by Siemens >

The mechanisms that constitute VHE enable the development of 3G services/applications, but may also be used to provide 2G services/applications for continuity reasons. For Release '99 the service capability servers include gsmSCF, MExE server and SAT server and access to bearers. In Release '99 the new 3G services will be executed in the Home Environment domain also when roaming.. 2G supplementary services are executed according to GSM roaming principles.

~~6~~5 Open Service Architecture

~~The figure below shows an overall picture of the OSA.~~



~~**Figure** Open Service Architecture~~

~~The applications constitute the top level of the Open Service Architecture (OSA). The application servers implement the services/applications offered to the end users. This level is connected to the Service Capability Servers (SCSs) via the OSA interface. The SCSs are placed on the border between the core networks and the applications environment, and are thus shielding the applications from telecom specific protocols (e.g. MAP, CAP, H.323, SIP etc.).~~

~~Many new services will be so called client/server applications, with clients residing in the Mobile Station (MS). Typically MExE and SAT applications will have clients in the MS.~~

In order to implement the VHE, including also today not known end user services/applications, 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 interface to the network through the OSA interface that is specified in this Technical Specification.

The network functionality is offered by ~~the~~ different Service Capability Servers (SCSes) and appear as ~~is called~~ 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). The different features of the different SCSs can be combined as appropriate. The exact content 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 CORBA interfaces, while the underlying core network functions use their specific protocols.

The aim of OSA is to provide an architecture that allows for inclusion of new SCSes in future releases of UMTS with a minimum impact on the applications using the OSA interface. This means that OSA is scalable, in that more SCSs can be defined as required.

To make it possible for operators and 3rd party application developers, possibly with no telecom competence, 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 CORBA interfaces. The service capability servers will serve as gateways between the legacy systems and the applications environment.

~~The OSA is scalable, in that more SCSs can be defined as required.~~

~~4Introduction to VHE and OSA~~

~~<Editor's note: purpose is to introduce VHE and OSA in a few sentences and to explain what has to be standardised >~~

~~- A few sentences about VHE and OSA, explaining that:~~

~~VHE = portability across terminals and networks of PSE~~

~~OSA = architecture that enables applications (both operator and 3rd party developed) to make use of the functionality provided by service capability serverss, in UMTS release 99: CAMEL, MExE, SAT and "access to bearers"~~

~~<Editor's note: example to be added to show that OSA enables applications to use functionality from various service capability servers (e.g. route planning application using HLR to get location information and CAMEL to setup call to the user>~~

~~5Virtual Home Environment~~

~~<Editor's note: some text on VHE >~~

~~< possible issues to be discussed include User Profile related issues and possible interworking between Service Capability Servers like CAMEL, MExE and SAT. >~~

~~6Open Service Architecture~~

~~<Editor's note: this chapter briefly explains what OSA is, including:~~

~~-showing overall OSA architecture~~

~~-introducing the components that constitute OSA (such as Application Servers and Service Capability Servers~~

~~-explaining basic mechanisms in OSA, e.g. how applications find out what functionality is provided by the Service Capability Servers>~~

~~6.1~~5.1 Overview of the Open Service Architecture

~~<Editor's notes:~~

~~-the figure shown below is a starting point. Contributions are invited to improve the figure~~

~~-improvement agreed during drafting (not reflected yet): "CSE" will be changed into "gsmSCF"~~

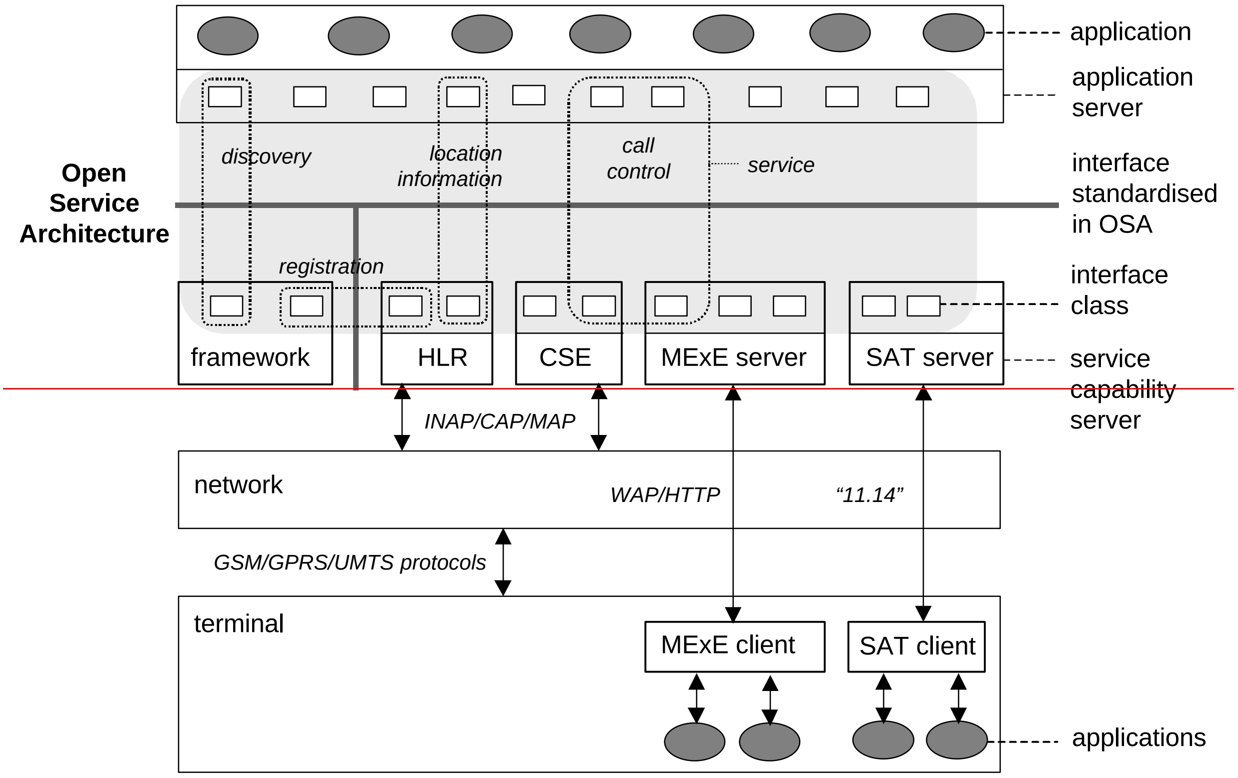
~~-clarification needed on the relation between Application Server and gsmSCF>~~

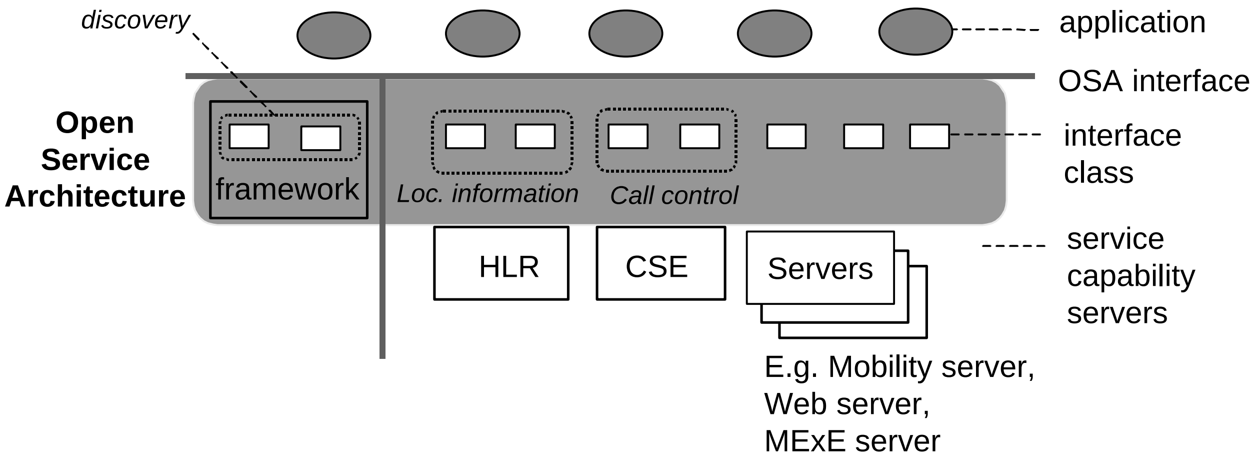
The Open Service Architecture consists of three parts:

- **Applications**, e.g. VPN, conferencing, location based applications. These applications are implemented in one or more Application Servers;
- **Framework**, providing the applications with basic ~~services~~ 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 by 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.

- **Service Capability Servers**, providing the applications with network services that are abstractions from underlying network functionality. Examples of network services offered by the Service Capability Servers are Call Control, Message Transfer and Location. Network s~~S~~ervices are possibly provided by more than one Service Capability Server. For example, the Call Control service might be provided by CAMEL and MexE. Examples of possible ~~The~~ Service Capability ~~iesy~~ Servers are ~~taken into account for UMTS Release 99~~ are CSE~~AMEL~~, MexE Server, SAT Server and HLR~~[1].~~





~~[1] It shall be noted that the HLR as such is not a Service capability, but the ability to access HLR data is meant here.~~

Figure 22221~~Logical o~~Overview of Open Service Architecture

This specification defines the OSA interface. The way it is implemented (e.g. the programming language used) is outside the scope of this specification and is vendor specific.

Several options exist:

- All interface classes which provide the OSA interface are implemented on one entity outside the core network.
- The interface classes which provide the OSA interface are implemented on different entities outside the core network.
- The interface classes which provide the OSA interface are implemented on different entities outside the core network as well as inside one or more core network entities.

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.

~~Implementation of the service capability server is vendor specific. Several options exist:~~

~~In a separate entity. The interfaces between that entity and the entity implementing the Service capability is then non-standardised~~

~~As a software layer in the entity implementing the Service Capability.~~

~~It is the framework that will provide the applications with an overview of available service capability features and a reference to where they can be found.~~

<Remark to Application Interface:

The network functionality can be accessed via network and mobile based clients, 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>

~~6.2~~5.2 Principles in the Open Service Architecture

Section 5.1~~5.15.16.1~~ introduced the three parts that constitute the Open Service Architecture; these parts are: Applications, Framework and Service Capabilities. This section explains how the three parts are structured and what exactly is standardised in the OSA. Furthermore it explains the principles for application registration, service registration and application invocation.

~~6.2.1~~5.2.1 OSA Interface~~s in OSA~~

The OSA interface is specified in terms of a number of interface classes. The interface classes are divided into two groups:

- **framework interface classes**
- **service interface classes**

<Editor's note: document has to be checked on consistent use of service interface classes rather than service interfaces >

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 descriptional 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.

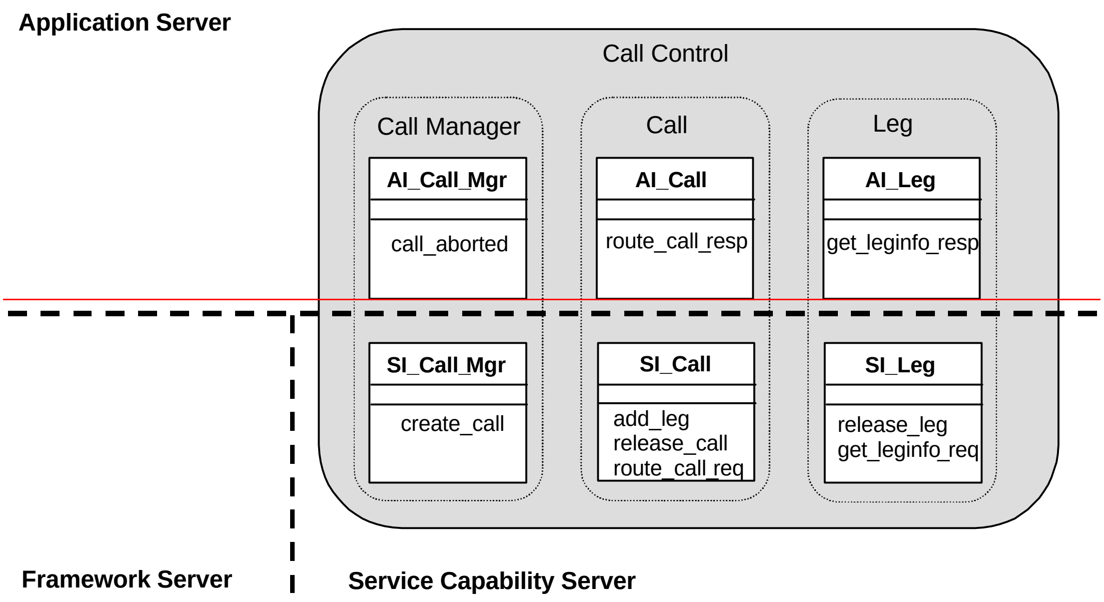
~~All services (both those in the Framework and in the Service Capabilities) are defined in terms of a number of interface classes. This means that only interface classes are specified, not the object implementations in the Application Server, Framework Server and Service Capability Servers that implement these interface classes. The interface classes can be divided into three groups, similar to the way the Open Service Architecture is divided into three parts. These three groups are:~~

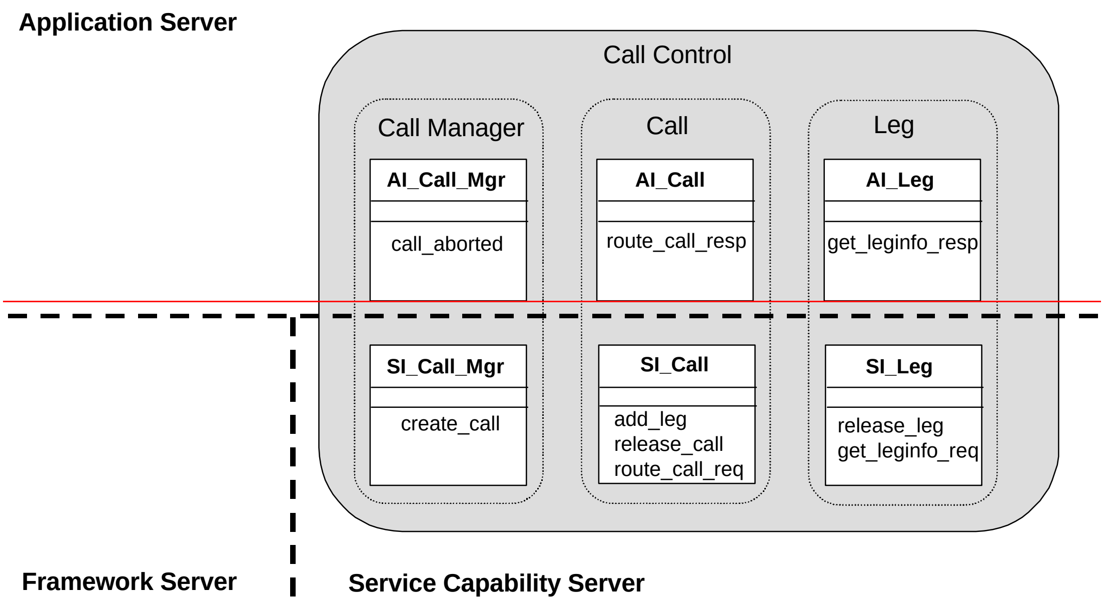
- ~~**application interface classes** (hereafter abbreviated as "application interfaces")~~
- ~~**framework interface classes** (hereafter abbreviated as "framework interfaces")~~
- ~~**service capability interface classes** (hereafter abbreviated as "service interfaces")~~

~~Each service contains application interfaces and/or framework interfaces and/or service interfaces.~~

~~Example:~~

~~The Call Control service is specified as a collection of service interfaces and application interfaces. Examples of the Call Control service interfaces are: SI_Call_Manager, SI_Call and SI_Leg. Examples of Call Control application interfaces are: AI_Call_Manager, AI_Call, AI_Leg. A more precise description can be found in section 8.1. Figure 2 illustrates the example. Each interface is defined in terms of methods that can be invoked on the interface. For example, the SI_Call_Mgr interface might contain the method Create_Call (which realises the Service capability feature 'Initiate and create session' as specified in [reference to stage1]).~~





~~**Figure 32** Example: interface classes that specify the Call Control service~~

~~6.2.2~~5.2.2 Basic mechanisms in OSA

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 ~~Server a~~and Framework ~~Server:~~

- **Establishment of service agreement**. Agreement established either off-line (e.g. by physically passing messages) or on-line.
- **Discovery of framework and service interfaces**. Once a service agreement exists, the application ~~server~~ can obtain ~~all~~ available framework interface classes and allowed service interface classes~~s~~. ~~This mechanism is in general applied before starting the development of a new application or before activation of the application~~The Discovery interface can be used at any time.

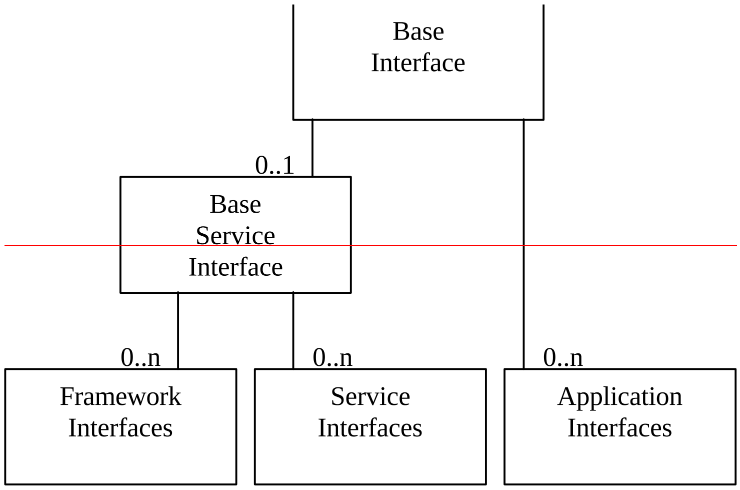
Basic mechanism between Framework ~~Server~~ and Service Capability Server:

- **Registering of service interfaces**. ~~Service i~~Interface classes~~s~~ offered by a Service Capability Server can be registered at the Framework ~~Server~~. In this way the Framework ~~Server~~ can inform the Application ~~Server~~ upon request about available service interface classes (Discovery)~~s~~. 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 a VPN application, the application needs to be invoked when the user makes a call. An event is in this case requested on the Calling Party Number of the user.

5.3 Base interface classes



~~Figure 4 Interface class hierarchy~~

~~6.3.1~~5.3.1 Base Interface Class

<Editor's note: some text is needed to explain the concept of Base Interface Class >

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

Name	Base_Interface
Method	
Parameters	
Returns	
Errors	

5.3.2 Base Service Interface class

This class provides the base for ALL service and framework interfaces described in the following chapters. It allows an application~~s~~ to set a reference to the application, which is used by the OSA interface~~service and framework interfaces when initiating an~~ 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 to. This reference can be provided by the application 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

-

~~7~~6 Framework service capability features

~~7.1~~6.1 Authentication

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.

~~7.1.1~~6.1.1 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 client application and Authentication Framework authenticating each other. This uses an authentication mechanism chosen by the Authentication Framework. ~~After authentication the client application and Authentication Framework negotiate a service agreement which will involve each party digitally signing the agreement.~~

- ~~A client application sends an initial message to the Authentication Framework - this will include the authentication capabilities of the client application. The Authentication Framework will then choose an authentication mechanism based on information about the authentication capabilities of the framework, client application and the service requested. If the client 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 client application. The Authentication Framework will instruct the client to perform the agreed mechanism.~~

- ~~The client 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 client 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 client 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.~~

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

~~7.1.1~~6.1.2 Authentication interface class~~ - framework interface~~

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

- A client application sends an initial message to the Authentication Framework - this will include the authentication capabilities of the client application. The Authentication Framework will then choose an authentication mechanism based on information about the authentication capabilities of the framework, client application and the service requested. If the client 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 client application. The Authentication Framework will instruct the client to perform the agreed mechanism.
- The client 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 client 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 client 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.

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

Method **`authenticateClient()`**

This method is used by the framework to authenticate the client. The client 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 client. The challenge will be encrypted with the mechanism prescribed by the `initiateClientAuthentication()`.

Returns response

This is the response of the client 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

Direction~~Name~~ Framework to application

Method **`terminateAppClientAuthentication()`**

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

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 client uses this to check `terminationText`. If a match is made, the authentication is terminated, otherwise an error is returned.

Returns

Errors

~~Name~~Direction Framework to application

Method **`signAppServiceAgreement()`**

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

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 client.

agreementText

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

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 client.

Errors

~~Name~~Direction Framework to application

Method **`terminateAppServiceAgreement()`**

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

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 client.

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 client uses this to check the `terminationText`. If a match is made, the service agreement is terminated, otherwise an error is returned.

Returns

Errors

~~7.1.2Authentication - service interface~~

Direction~~Name~~ Application to framework

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 client capability. If a mechanism within the client's capability cannot be found, the framework must return an error.

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 client. 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.

Direction~~Name~~ Framework to application

Method

authenticateFramework()

This method is used by the client to authenticate the framework. The framework must respond with the correct responses to the challenges presented by the client. 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).

Parameters

challenge

The challenge presented by the client to be responded to by the framework. The challenge mechanism used will be in accordance with 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 client in the current sequence. The response will be the challenge, decrypted with the mechanism prescribed by the initiateClientAuthentication().

Errors

~~Name~~Direction Framework to application

Method

terminateClientAuthentication()

This method is used by the client 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).

Parameters

terminationTex

This is the termination text that is signed by the client using the private key of the client 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

~~Name~~Direction Application to framework

Method **SelectService()**

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

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

~~Name~~Direction Application to framework

Method **SignServiceAgreement()**

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

Parameters ServiceToken

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

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.

~~Name~~Direction Application to framework

Method **`terminateServiceAgreement()`**

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

Parameters

serviceToken

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

terminationText

This is the termination text that is to be digitally signed by the client. 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 `terminationText`. If a match is made, the service agreement is terminated, otherwise an error is returned.

Returns

Errors

~~Name~~Direction Application to framework

Method **`obtainFrameworkInterface()`**

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

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

~~7.2~~6.2 Authorisation

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

~~7.2.1Authorisation - framework interface~~

~~7.2.2Authorisation - service interface~~

~~7.3~~6.3 Event Notification

~~7.3.1Event Notification – framework interface~~

Direction~~Name~~ Application to framework

Method **`enableNotification()`**

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

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

~~Name~~Direction Application to framework

Method **`disableNotification()`**

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

~~7.3.2Event Notification – application interface~~

~~Name~~Direction Framework to application

Method `eventNotify()`

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

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

~~Name~~Direction Framework to application

Method `notificationTerminated()`

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

Parameters

Returns

Errors

~~7.4~~6.4 Registration

~~<Editor's note: it has to be decided in contributions whether registration of Service Capability Servers at the Framework is performed at "service level" (e.g. "Call Control", Location Management, Messaging etc.) or even at a more detailled level (e.g. create call, get location etc.)>~~

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

~~7.4.1Registration – framework interface~~

~~7.4.2Registration – service interface~~

~~7.5~~6.5 Discovery

~~7.5.1Discovery – framework interface~~

Direction~~Name~~ Application to framework

Method `discoverService()`

This method returns the service identity associated with a service.

Parameters serviceProperties

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

Returns serviceID

This is the unique identity of the service.

Errors

~~7.5.2Discovery – application interface~~

~~-~~

~~8~~7 Non-Framework service capability features

The service capability features provided to the application by service capabilities ~~such as CAMEL, MExE, SAT, HLR~~servers to enable access to network resources.~~. These services enable the application to make use of the “real” network functionality, i.e. the functionality needed in the applications.~~

<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 .

~~8.1~~7.1 Call Control

~~*<Editor's note: in TS22.121 the term "session control" is used instead of "call control". It depends on future contributions whether the scope of OSA in 3GPP Release99 will include "session control" or only limited "call control" support>*~~

The Call control network service~~Service Capability Features~~ consist of three interface classes:

1. C~~A c~~all manager, containing management function for call related issues
2. C~~A c~~all ~~interface~~
3. ~~A l~~Leg ~~interface~~

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.

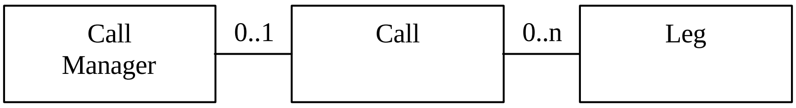


Figure 5~~555~~ Call control class hierarchy

~~8.1.1~~7.1.1 Call Manager

~~8.1.1.1Call Manager – service interface~~

~~This interface is the 'service manager' interface for the Generic Call Capability Features.~~

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 set the call gap rate, to create call objects and to enable or disable call-related event notifications.

~~Name~~Direction Application to network

Method **`SetCallLoadControl()`**

This method imposes or removes load control on calls made to a particular address range within the generic call control service.

Parameters AddressRange

Specifies the address range to which the overload control should be applied or removed.

Duration

Specifies the duration for which the load control should be set. If the duration is zero, then the load control is removed.

Mechanism

Specifies the load control mechanism to use (for example, admit one call per interval), and any necessary parameters, such as the call admission rate. The contents of this parameter are ignored if the load control duration is set to zero.

Treatment

Specifies the treatment of calls that are not admitted. The contents of this parameter are ignored if the load control duration is set to zero.

Returns

Errors

~~Name~~Direction Application to network

Method **`CreateCall()`**

This method is used to create a new call object.

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

~~Name~~Direction Application to network

Method **`EnableCallNotification()`**

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

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

~~Name~~Direction Application to network

Method **`DisableCallNotification()`**

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

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.

~~8.1.1.2Call Manager – application interface~~

~~The generic call manager application interface provides the application call management functions to the generic call service.~~

~~Name~~Direction Network to application

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.

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

~~Name~~Direction Network to application

Method **callOverloadEncountered()**

This method indicates that the network has detected overload and may have automatically imposed load control on calls requested to a particular address range or calls made to a particular destination within the generic call control service.

Parameters AddressRange

Specifies the address range within which the overload has been encountered.

overloadType

Specifies the type of overload encountered.

treatment

Specifies the treatment of calls that are not admitted due to the load control imposed.

Returns

Errors

~~Name~~Direction Network to application

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.

Parameters AddressRange

Specifies the address range within which the overload has ceased.

overloadType

Specifies the type of overload that has ceased.

Returns

Errors

~~Name~~Direction Network to application

Method `callFaultDetected()`

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

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

~~Name~~Direction Network to application

Method `callEventNotify()`

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

Parameters

call

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

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.

appInterface

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

Returns

Errors

~~Name~~Direction Network to application

Method `callNotificationTerminated()`

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

Parameters

Returns

Errors

7.1.2 Call

~~Call – service interface~~

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.

~~Name~~Direction Application to network

Method **routeCallToDestination_Req()**

This asynchronous method requests routing of the call (and inherently attached parties) to the destination party, via a passive call leg (which is implicitly created).

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.

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

~~Name~~Direction Application to network

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). The call object must already have been created

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

~~Name~~Direction Application to network

Method `releaseCall()`

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

Parameters callSessionID

Specifies the call session ID of the call.

cause

Specifies the cause of the release.

Returns

Errors

~~Name~~Direction Application to network

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.

Parameters callSessionID

Specifies the call session ID of the call.

Returns

Errors

~~Name~~Direction Application to network

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.

Parameters callSessionID

Specifies the call session ID of the call.

callInfoRequested

Specifies the call information that is requested.

Returns

Errors

~~Name~~Direction Application to network

Method **`setCallChargePlan()`**

Set an operator specific charge plan for the call. The charge plan must be set before the call is routed to a target address.

Parameters callSessionID

Specifies the call session ID of the call.

callChargePlan

Specifies the charge plan to use.

Returns

Errors

~~Name~~Direction Application to network

Method **`getCallState()`**

This method requests the current state of the call.

Parameters callSessionID

Specifies the call session ID of the call.

Returns State

Specifies the current state of the call.

Errors

~~Name~~Direction Application to network

Method `getCallLegs()`

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

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

~~Name~~Direction Application to network

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).

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

~~Name~~Direction Application to network

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.

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

~~Name~~Direction Application to network

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.

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

~~Name~~Direction Application to network

Method `getControlLeg()`

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

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

7.1.2.1.1 State Diagram

Figure 6~~Figure 6Figure 6Figure 3~~ shows the state model for the generic call interface. 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 request arrived.

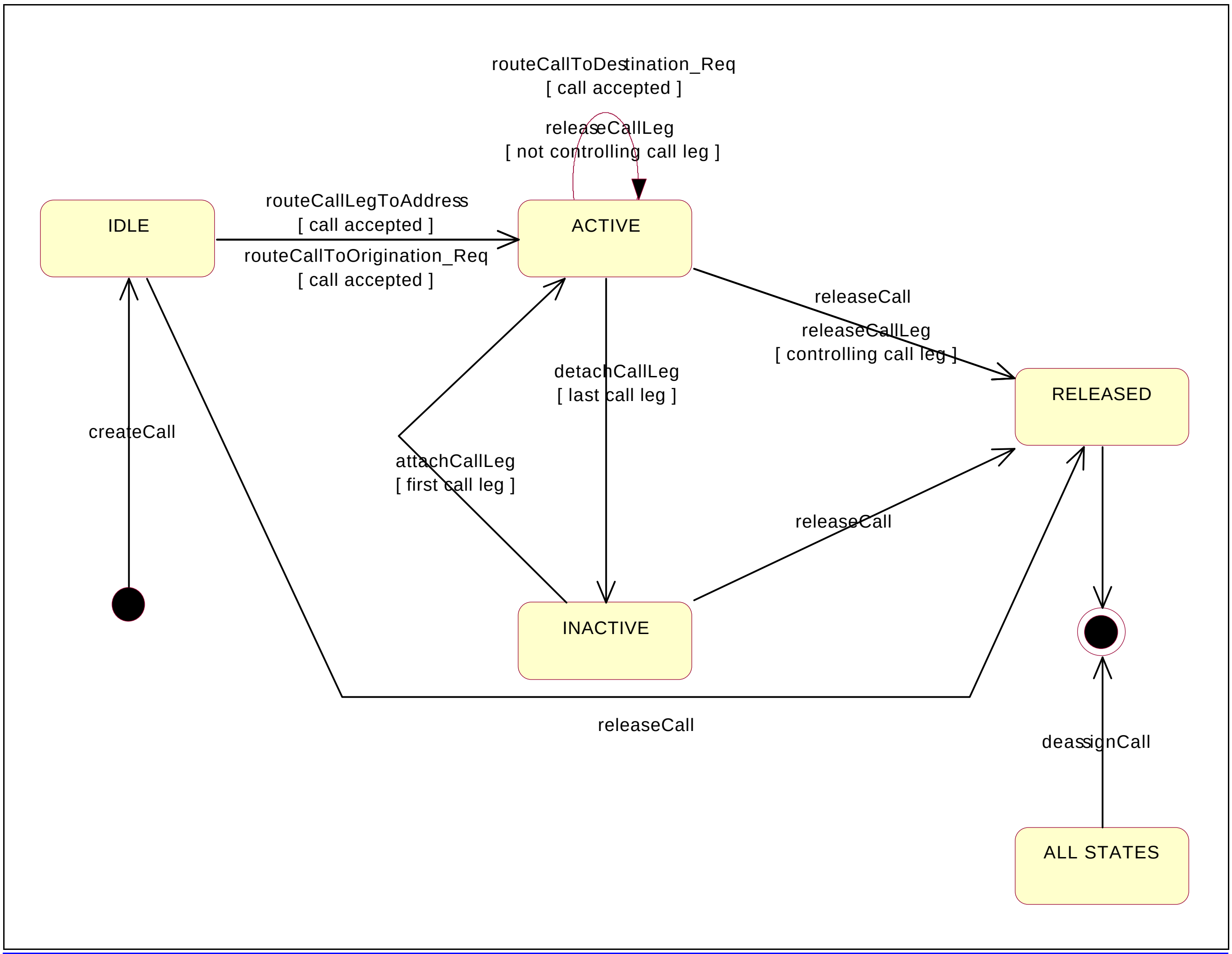


Figure 6~~6663~~ - State diagram for the `Call` interface

~~8.1.2.2Call – application interface~~

~~The generic application call interface is implemented by the client application developer and is used to handle call request responses and state reports.~~

~~Name~~Direction Network to application

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.

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

~~Name~~Direction Network to application

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.).

Parameters

callSessionID

Specifies the call session ID of the call.

error

Specifies the error which led to the original request failing.

Returns

Errors

~~Name~~Direction Network to application

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.

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

~~Name~~Direction Network to application

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.).

Parameters callSessionID

Specifies the call session ID of the call.

error

Specifies the error which led to the original request failing.

Returns

Errors

~~Name~~Direction Network to application

Method **getCallInfo_Res()**

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

Parameters callSessionID

Specifies the call session ID of the call.

callInfoReport

Specifies the call information requested.

Returns

Errors

~~Name~~Direction Network to application

ion

Method getCallInfo_Err()

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

Parameters callSessionID

Specifies the call session ID of the call.

error

Specifies the error which led to the original request failing.

Returns

Errors

7.1.3 Call Leg

~~Call Leg – Service interface~~

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.

~~Name~~Direction Application to network

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.`

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

~~Name~~Direction Application to network

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.

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

~~Name~~Direction Application to network

Method `getCallLegState()`

This method requests the current state of the call leg.

Parameters callLegSessionID

Specifies the call leg session ID of the call leg.

Returns state

Specifies the current state of the call leg.

Errors

~~Name~~Direction Application to network

Method `releaseCallLeg()`

This method requests the release of the call leg. If successful, the associated address (party) will be released from the call, and the call leg deleted.

Parameters callLegSessionID

Specifies the call leg session ID of the call leg.

cause

Specifies the cause of the release.

Returns

Errors

~~Name~~Direction Application to network

Method **getAddresses()**

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

Parameters callLegSessionID

Specifies the call leg session ID of the call leg.

Returns addressList

Specifies the addresses associated with the call leg.

Errors

Direction~~Name~~ Application to network

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.

Parameters callLegSessionID

Specifies the call leg session ID of the call leg.

callLegInfoRequested

Specifies the call leg information that is requested.

Returns

Errors

Direction~~Name~~ Application to network

Method **getCallLegType()**

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

Parameters callLegSessionID

Specifies the call leg session ID of the call leg.

Returns callLegType

Specifies the call leg type.

Errors

Direction~~Name~~ Application to network

Method `getCall()`

This method requests the call associated with this call leg.

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

7.1.3.1.1 State Diagram

Figure 7~~Figure 8Figure 7Figure 4~~ shows the state model for the generic call leg interface. 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.

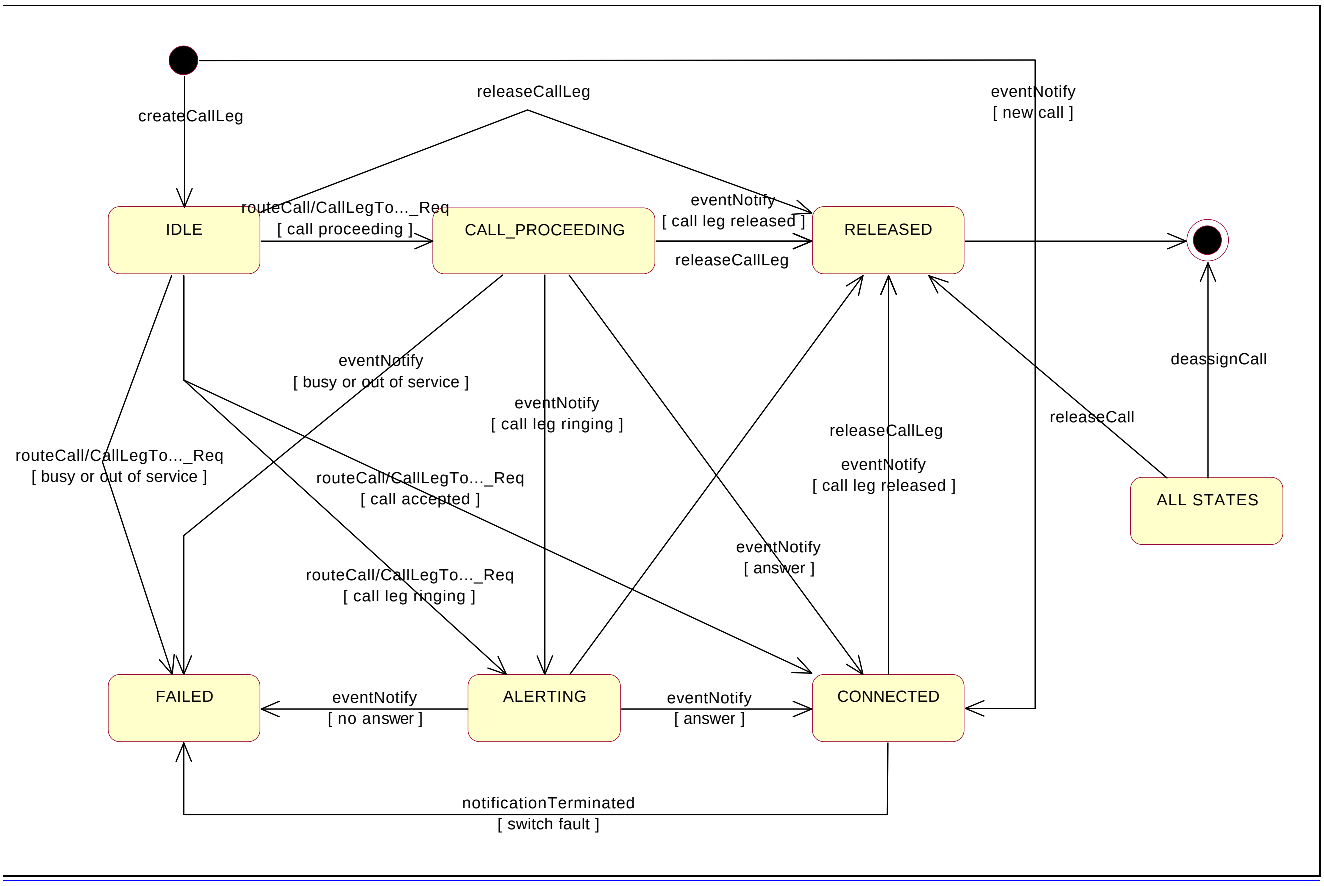


Figure 7~~8774~~ - State diagram for the `CallLeg` interface

~~Call Leg – Application interface~~

The generic application call leg interface is implemented by the client application developer and is used to handle responses and errors associated with requests on the call leg.

Direction~~Name~~ Network to application

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.).

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

Direction~~Name~~ Network to application

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.).

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

Direction~~Name~~ Network to application

Method `GetCallLegInfo_Res()`

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

Parameters CallLegSessionID

Specifies the call leg session ID of the call leg.

CallLegInfoReport

Specifies the call leg information requested.

Returns

Errors

Direction~~Name~~ Network to application

Method **`GetCallLegInfo_Err()`**

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

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

~~8.2~~7.2 Security/privacy

~~8.3~~7.3 Address Translation

~~8.4~~User Location

~~8.4.1User Location – Service Interface~~

Direction~~Name~~ Application to network

Method **`EnableLocationNotification()`**

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

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

Direction~~Na~~ Application to network

~~me~~

Method **`disableLocationNotification()`**

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

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.

Direction~~Name~~ Application to network

Method **`getUserLocation()`**

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

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

~~User Location – Application Interface~~

Direction~~Name~~ Network to application

Method **locationReport()**

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

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

Direction~~Name~~ Network to application

Method **locationNotificationTerminated()**

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

Parameters

Returns

Errors

~~8.5~~7.4 User Status

~~8.5.1User Status – Service Interface~~

Direction~~Name~~ Application to network

Method **enableStatusNotification()**

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

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

Direction~~Name~~ Application to network

Method **`disableStatusNotification()`**

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

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.

Direction~~Name~~ Application to network

Method **`GetUserStatus()`**

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

~~User Status – Application Interface~~

Direction~~Name~~ Network to application

Method **`statusEventNotify()`**

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

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

~~Name~~Direction Network to application

Method **`statusNotificationTerminated()`**

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

Parameters

Returns

Errors

~~8.6~~7.5 Terminal Capabilities

~~8.7~~7.6 Message Transfer

~~8.8~~7.7 Data Download

~~8.9~~7.8 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>

~~8.10~~7.9 Charging

CAMEL Call L~~eg - Service interface~~

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

Direction~~Name~~ Application to network

Method **`setChargeInfo()`**

The application calls this method to insert charging information in the cal data records (CDR) generated by the network

Parameters

callLegSessionID

Specifies the call leg session ID of the call leg.

chargeInfo

Specifies the charging information. The format is application specific.

Returns

Errors

Direction~~Name~~ Application to network

Method `setAdviceOfCharge()`

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

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

Direction~~Name~~ Application to network

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.

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

~~CAMEL Call Leg – Application Interface~~

Direction~~Name~~ Network to application

Method **superviseCall_Res()**

This asynchronous method reports a call supervision event to the 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

Direction~~Name~~ Network to application

Method **superviseCall_Err()**

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

Parameters callSessionID
Specifies the call session ID of the call.

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

Returns

Errors

~~9~~8 Relation between interface classes and service capability servers

<Editor's notes:

- *the purpose of this chapter is to define the mapping between interface classes (and methods) in the OSA Application Interface and the underlying protocol operations and parameters >*
- *section 9.1 is an example of how the mapping is defined. Other sections are needed, e.g. to define the mapping between User Location Management and HLR etc.>*

~~9.1~~8.1 Relation between Call Control and CAMEL

- *Do we also define mapping from application interface to SCSes (e.g. making subscription to registered user results in creation of / addition to O-CSI, T-CSI etc.?*

  Question: who sets correct O-CSI, T-CSI, etc. in HLR: application provider or operator? (in first case, these management operations need to be standardised)

9 Annex - Example of use of OSA (informative)

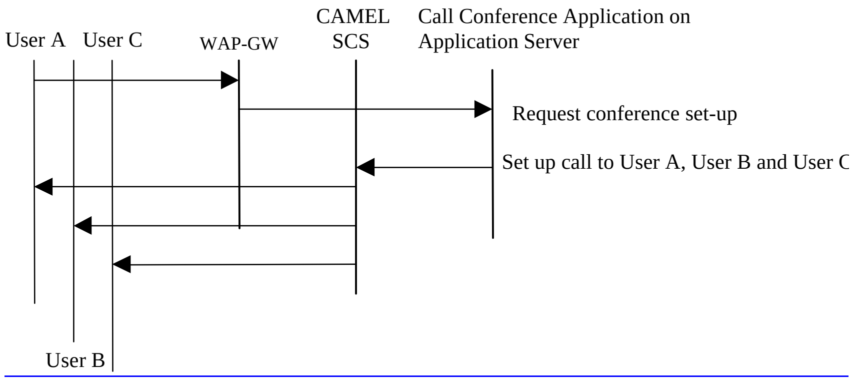
The following example shows how the OSA, described on a high level, could be used to execute an application. Note that OSA enables the use of various Service Capability Servers by an application.

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.


User A
User C
WAP-GW
CAMEL SCS
Call Conference Application on Application Server
Request conference set-up
Set up call to User A, User B and User C
User B

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"

10 History

Date	Version	Comment
July 1999	0.1~~0~~.0	Initial Draft produced in ~~at~~ 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
Rapporteur:		
Email:		Telephone: