Oracle® Communications Services Gatekeeper

Patch Release Notes

Release 5.0.0.1

E24004-03

October 2012

Oracle Communications Services Gatekeeper Patch Release Notes, Release 5.0.0.1

E24004-03

Contents

Preface

This book provides the patch release notes for of Oracle® Communications Services Gatekeeper 5.0.0.1.

Audience

This document is intended for anyone who needs an understanding of the contents contained in this Oracle Communications Services Gatekeeper release.

This includes:

- System administrators charged with installing and maintaining Oracle Communications Services Gatekeeper
- Third-party application developers who wish to integrate telephony-based functionality into their products
- Operator-based system developers who wish to extend the functionality of Oracle Communications Services Gatekeeper or to integrate it with Partner Relationship Management (PRM) or Operations Support Systems (OSS) tools
- Managers, support engineers, and sales and marketing personnel for network operators and service providers

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Related Documents

For more information, see the following documents in the Oracle Communications Services Gatekeeper set and patch documentation:

- *Accounts and SLAs Guide*

- *Alarm Handling Guide*
- *Application Developer's Guide*
- *Communication Service Guide*
- *Deployment Guide*
- *Installation Guide*
- *Java API Reference*
- *Licensing Guide*
- *OAM Java API Reference*
- *One API Application Developer's Guide*
- *OAuth Guide*
- *Partner Relationship Management Guide*
- *Platform Development Studio Developer's Guide*
- *Platform Test Environment Guide*
- *RESTful Application Development Guide*
- *SDK User's Guide*
- *Statement of Compliance*
- *System Administrator's Guide*
- *System Backup and Restore Guide*

1 Introduction

These Release Notes summarize the product enhancements, resolved issues, and known issues in Oracle Communications Service Gatekeeper Patch Set 5.0.0.1.

Enhancements

Enhancements in this release include:

- RESTful OneAPI application interfaces

  Services Gatekeeper supports a RESTful application interface that is compatible with the Global System for Mobile Communications (GSM) OneAPI initiative.

  See *REST One API Application Developer's Guide*.

- Enhanced RESTFul Service Support and Platform Development Studio (PDS) REST Wizard

  The Services Gatekeeper RESTFul facade mediates communication between RESTFul clients and both RESTful and custom network endpoints. The PDS contains a new wizard for creating RESTFul communication services from Web Application Description Language (WADL).

  See Chapter 3, "Using Services Gatekeeper with REST Services," for more information.

- XParameter Filter

  For enhanced security, tunneled parameters are filtered to ensure that they are allowed before the request is executed.

  See Chapter 4, "Filtering Tunneled Parameters," for more information.

- Exposure of SMPP and UCP APIs

  The Services Gatekeeper SMPP and UCP APIs are now available for platform developers to create custom plug-ins using these interfaces to connection services used by the standard Services Gatekeeper implementations.

  See Chapter 6, "Using the SMPP API," Chapter 5, "Using the UCP API," and the oracle.ocsg.protocol packages in the *Services Gatekeeper Java API Reference*.

- Composed Service Level Agreement (SLA) Interface

  The SLA Manager in the Platform Test Environment has been enhanced to provide a graphical interface for creating and managing composed SLAs.

  See Chapter 7, "Configuring and Testing Composed Service-Level Agreements," for more information.

- Interceptor Chain Rule Customization by Communication Service.

  See Chapter 8, "Interceptor Chain Customization," for more information.

- Platform Development Studio (PDS) Eclipse wizard for creating custom service interceptors.

  See Chapter 9, "Wizard for Interceptor Creation," for more information.

- WAC API and Operator Endpoint Support

  Services Gatekeeper can function as a Wholesale Applications Community (WAC) Operator Endpoint Server in the Payment API flow using OneAPI 2.1 Payment APIs.

  See Chapter 10, "Wholesale Applications Community Enhancements," for more information.

- Support for OAuth 2.0 Authentication Server and Resource Server functions.

  See Chapter 11, "Services Gatekeeper OAuth 2.0 Authorization and Resource Servers," for more information.

- Additional documentation about how the Parlay X 2.1 Short Messaging/SMPP communication service handles data coding.

  See Chapter 12, "Data Coding," for more information.

Resolved Issues in This Release

Table 1–1 describes known issues from the previous release that are resolved in this release.

Table 1–1 Fixes in this Release

Bug ID / SR ID	Description
13334021 / SR: 3-4723119351	Fixed the order of soap message response for array type elements.
12960582 / SR: 3-4365362351	Added the flag: **-Docsg.payment.px30.ocsgInternalAvp.ignore** to control sending the Internal Services Gatekeeper AVPS in the Credit Control Request (CCR) request. To ignore internal parameters, set the flag to **true**.
13095419 / SR: 3-2889334421	Added support for **schedule_delivery_time** x-parameter for PX21 SMPP traffic path.
11808518 / SR: 3-3048218981, 3-2391892421	Multiple fixes in rest support.
9768407 / SR:3-175439510 1	Added support of **enduseridentifier** parameter in EDR.
11799441 / SR: 3-2395773771	Added support to pass vendor specific error codes from network to applications in Location Traffic path.
11062440 / SR: 3-2221385091	Added support to make **Interval** element configurable in TLRR element in Location traffic path.
9849158 / SR: 3-1856167821	Fixed issue with not picking up configured values for new MLP plugin instances.
11807542 / SR: 3-3005753531	Removed trailing comma in subscriber profile base **DN** entry when entry is blank.

Table 1–1 (Cont.) Fixes in this Release

Bug ID / SR ID	**Description**
13015110 / SR: 3-4471023581	Added support for the optional element **EarliestDeliveryTime** in the MM7 submit request with the new x-parameter **EarliestDeliverytime** for all supported MM7 Release 5 schema versions. The optional element **ExpiryDate** in the MM7 submit request is now supported by x-parameter **ExpiryDate** for all supported MM7 schema versions.
12980832 / SR: 3-4489005781	A delivery receipt containing the message state **ENROUTE** will only result in a **deliverSMResp** response sent back to the SMSC.
12603510 / SR: 3-3519806031, 3-3582264101	Two new MBean attributes are introduced: **SmscGroupId** (empty by default) and **SmscGroupIdEnabled** (disabled by default) The use of these two attributes will solve issues with delivery reports not being handled when Services Gatekeeper is connected to a clustered SMSC with multiple network interfaces. The attribute **SmscGroupIdEnabled** shall be enabled when the Services Gatekeeper system has multiple Px21 SMPP plugin instances that connect to different SMSC network interfaces that are part of one logical SMSC. If the SMSC has several network interfaces and a delivery report for a message sent to *SMSC network interface-1* can be sent out from *SMSC network interface-2.* The attribute **SmscGroupId** should be set to a common value for all plugin instances.
12961020 / SR: 3-4381314411, 3-4381314411	A feature has been implemented to discard the suffix **/TYPE=PLMN***XXXXXX* from the address received in any **addresstype** (**Number**, **RFC2822Address**, **ShortCode**) in an MM7 **DeliveryReportReq** or **DeliverReq**. The feature is controlled by a system property **ocsg.mms.remove_typeplmn**. The feature is enabled by default (suffix is removed). To disable (do not remove suffix), add the following system property on all NT servers: **-Docsg.mms.remove_typeplmn=false**.
12730184 / SR: 3-3804520981	Added a condition not to initialize security providers on AT servers. This will prevent database connections being setup from AT servers.
12709038 / SR: 3-3835994781	**SiteAddress** is stored as a Serialized object instead of a String. This will remove the limitation of 255 characters.
12401601	Added a fix so that AT servers will start correctly if the Domain Wide Administration Port is enabled.
12410443	"**<?xml version='1.0' encoding='utf-8'?>** is added to MM7 SOAP requests.
8176822	The patch script can now patch all artifacts, including the eclipse jar.
12564657 / SR: 3-3543564631	Criteria in notifications are now case insensitive.
9727269 / SR: 3-1716371691	The granularity of start or end dates was reduced to include '**day**' as the least significant part. Also, the date stored as budget configuration internally is now taken from the SLA instead of current time. Note that SLAs already stored need to be updated in order to pick up the new dates. Also, logging of budget configuration that is diffing between two geographically redundant sites has been improved to ease troubleshooting.
11895163	The **Circular Notification Area** is now visible in PTE map.
11904789 / SR: 3-3129449431	The build script was updated with a new target '**generate-mib-forked**' (made default) that forks before calling the generate-mib target. In addition the readme file was updated.

Table 1–1 (Cont.) Fixes in this Release

Bug ID / SR ID	**Description**
11901245 / SR: 3-3074032281	If data coding is provided in an SLA or x-parameter file, it is now used when encoding the message as well. The value of data coding used is taken in the following order: 1. If it exists in SLA, it is used first hand. 2. If it is provided as an x-parameter. 3. If not provided in SLA or x-parameter, the MBean attribute **DefaultDataCoding** is used.
11873021 / SR: 3-3074032281	The characters **^{}\[~]|** and **Euro sign(unicode 20AC)** and form feed from GSM extension table will now be supported when MBean attribute **DefaultDataCoding** is set to **0** (SMSC Default Alphabet).
11797936 / SR: 3-4608875831, 3-2395773771, 3-4676495231	The **commandStatus** in the response from SMSC will be included in the **ParlayX SVC0001 ServiceException** thrown to the application. A new general error code for the ParlayX SVC0001 ServiceException has been added. The new error code is **SMS-000006, "Could not send message. Message was not accepted by the network."** The complete error code in the ParlayX SVC0001 ServiceException will have following format: **SMS-000006:**<*CommandStatus*>:<*Description*>.
11799063 / SR: 3-3228503811	The existing general error code for the ParlayX SVC0001 ServiceException MMS-000005 has been updated to include the **StatusCode** and **StatusText** in the response from MMSC. The complete error code in the ParlayX SVC0001 ServiceException will have the following format: **MMS-000005:**<*StatusCode*>:<*StatusText*>
11857168	Implemented fix to check if the internal queue is null before checking its size. If queue is null (meaning windowing is disabled), **"SendQueueSize = N/A"** will be shown when invoking **listClientConnection** from SMPP server service.
10634032 / SR: 3-2682645161, 3-2692245931	Support for XSD VERSION REL-5-MM7-1-3 in ParlayX MMS service is added.
10326709 / SR: 3-2381491821	A location report is not sent when location data is null.
10203184 / SR: 3-2186379971, 3-2186379931	Fixed an issue that any application could poll for delivery status for any message. Now only the application that sent a specific message can use the polling feature to get the delivery status for that message. If an application attempts to poll for the delivery status for messages sent by a different application a service exception occurs.
10272046 / SR: 3-2103796457	**MM7 Priority** element can now be set using SLA configuration for ParlayX MMS Service.
11827827	Improvements in the SMPP service to properly cleanup and close connections to SMSC when they are manually reset by MBean operation **resetClientConnection** in the SMPP service.
11825557 / SR: 3-3085759721	For concatenated SMS messages consisting of segments, Services Gatekeeper will check all the segments associated with the message and will only use the **deliveredToTerminal** status if all segments have been reported as **delivered**.

Table 1–1 (Cont.) Fixes in this Release

Bug ID / SR ID	Description
10201065 / SR: 3-2186379871	To ensure MMS security, applications can only poll for messages using a **registrationIdentifier**, or correlator, that actually belongs to that specific application instance group. The same check is done when retrieving message attachments. An application can only use a **messageRefIdentifier** for a message that was intended for that application instance to get message content. If an application attempts to poll for messages or get message content for messages for a different application instance, a service exception occurs.
11713571 / 3-2889334421	The configured error response code will not be used when a delivery report is received but the message meta data can not be found in the store. This will prevent report redelivery attempts from the SMSC for a situation that is not temporary/can not be resolved.
11657534 / SR: 3-2744485251	X-parameters contained in the header of an inbound SOAP request are prevented from being echoed back in the response.
11062623	For the Parlay X Payment Service, Services Gatekeeper will validate that a reservation is granted by checking the Granted-Service-Unit in the Credit Control Answer (CCA) and making sure that the granted units are equal to or greater than the reservation request. If this is not the case an exception will be raised (**SVC0001 - payment error 000005**).
11665356	**sourceaddr** and **destaddr** values from the native SMPP **SUBMIT_SM** response are provided in the northbound message.
11669569	The **Deliver Report** CDR in the ParlayX SMS plugin is sent in the northbound direction.

10084247 / SR: 3-2073882931	The SMPP plug-in has been updated to check for x-parameter data that should be sent as optional SMPP TLV parameters. X-parameters can be added from an interceptor.
	The following x-parameters names are supported for tunneling TLV data:
	smpp_optional_int_tlv_param_tags
	smpp_optional_int_tlv_param_values
	smpp_optional_octet_tlv_param_tags
	smpp_optional_octet_tlv_param_values
	Note that tag identifiers must be entered in integer format. For example, a tag with hex value 0x1401 will be set as 5121. Multiple TLV tags/values are separated with a "**,**". The **TLV_OPTIONAL_INT_PARAM_TAGS** list and the **TLV_OPTIONAL_INT_PARAM_VALUES** list must be of the same size. Below is an example pseudo code for using integer TLVs:
	`//Integer parameters will always default to length 4`
	`injectXParam(TLV_OPTIONAL_INT_PARAM_TAGS, "5121,5124");`
	`injectXParam(TLV_OPTIONAL_INT_PARAM_VALUES, "999,1234");`
	Example pseudo code for using octet TLVs:
	`//Use octet type if we need length set to 2`
	`injectXParam(TLV_OPTIONAL_OCTET_PARAM_TAGS, "5121,5124", rctx);`
	`injectXParam(TLV_OPTIONAL_OCTET_PARAM_VALUES, "03e7,04d2", rctx);`
	`private void injectXParam(String name, String value, RequestContext rctx){ rctx.putXParam(name, value); }`

Table 1–1 (Cont.) Fixes in this Release

Bug ID / SR ID	Description
8795030 / SR: 2-5949734	The **enableReceiveMms** method now includes validations for the following: The **shortcode** must be alphanumeric characters with '**tel:**' scheme prefixed. The **appInstance** must really exist.
10421400 / SR: 3-2558532231	Added support for a new "**Bytes**" type in **paymentConfig.xsd**. Data is specified in hex format. For example: `<avpAttributeDefinitions> <avpAttribute code="23" vendorId="10415" name="3GPP-MS-TimeZone" type="Bytes" flag="0"></avpAttribute> </avpAttributeDefinitions> <avpTemplate> <avpValue avpName="3GPP-MS-TimeZone" defaultValue="08000000"/> … </avpTemplate>`
10433900	Byte types containing a null value will not be put into PDU.
10623367 / SR: 3-2632523371	The **startBinarySmsNotification** operation is now available in both console and the Platform Test Environment (PTE).
10430284 / SR: 3-2621104377	When **ReserveAddtionalAmount** is called, the Used Service Unit (USU) in the Credit Control Request (CCR) update will be set as the value of **pl_ payment_reservation_data.usedAmount**.The **pl_payment_reservation_ data.usedAmount** will be reset as zero, and will subtract USU from **pl_ payment_reservation_data.ReuestedAmount**. By default, this fix is disabled. To enable, set system.flag to enable: **-Docsg.payment.px30.reserveAdditionalAmount.usu.sent=true**
10430286 / SR: 3-2621121057	Requested Service Unit (RSU) is passed in **chargeAmount** request in the payment API.
8744222 / SR: 2-5933358	The following validations for the native MMS Plugin have been added: **enableRecieveMmsNotification** operation: includes validations for non-existent, invalid address, validations for non-existent, invalid **applicationInstanceGroupID**, and validations for non-invalid (non-proto based etc) **applicationURI**. **enableStatusReporting** operation: includes validations for non-existent, invalid **applicationInstanceGroupID**, and validations for non-invalid (non-proto based, etc) **applicationURI**.
10262848 / SR: 3-2285175981	Services Gatekeeper validates the length of x-Parameter **smpp_billing_id** in the SMS ParlayX interface. If the length is bigger than 1024, a warning message of the following format is logged: **"The length of XParam smpp_ billing_id is bigger than max size(1024 bytes):** *billingid...*" Services Gatekeeper will continue submitting request to SMSC without the **billing_identification** parameter.
8744324 / SR: 2-6007987	The current date (today) is now accepted in a SLA as a valid end date and will not cause a message sending failure.
9482737 / SR: 3-1306949931	A timer is scheduled to execute once per day. When the timer is executed, the SLAs for all service provider groups and application groups will be checked. If a SLA is about to expire, an EDR will be generated.
10205494 / SR: 3-2201224241	The correct number of CDRs are generated for third-party calls (TPC) to function properly with Diameter. The diameter simulator in the Platform Test Environment (PTE) reports the correct request counter.

Table 1–1 (Cont.) Fixes in this Release

Bug ID / SR ID	Description
10177822 / SR: 3-2176152431	The Third Party Call (TPC) plugin now populates the following fields in CDRs: **CDR@makeCall: +FIELD_CALL_INFO = "START", +FIELD_CDR_START_OF_USAGE, +FIELD_CDR_ORIGINATING_PARTY, +FIELD_CDR_DESTINATION_PARTY, +FIELD_CDR_AMOUNT_OF_USAGE = 1, +FIELD_CALL_SESSION_ID, +FIELD_CORRELATOR CDR@callConnected: +FIELD_CDR_CONNECT_TIME, +FIELD_CORRELATOR CDR@callReleased: +FIELD_CDR_END_OF_USAGE, CDR@endCall: +FIELD_CDR_END_OF_USAGE CDR@getCallInfo** and **CDR@cancelCall No change**. The **FIELD_CALL_SESSION_ID** must be used to group related CDRs to find out all field information.
13256317 / SR: 3-4714564051	Minor improvement by adding debug log that prints "**exception.getMessage()**" string from the exception that caused the "**notifySmsReception**" callback to application endpoint to fail.

Known Problems

Table 1–2 describes known problems in Services Gatekeeper 5.0.0.1.

Table 1–2 Known Problems in this Release

Bug ID	Description
13416402	**requestedAccuracy** is a mandatory parameter defined in the WADL file. When an empty value for **requestedAccuracy** ("**requestedAccuracy=**") in the request is sent to the Services Gatekeeper REST2REST module a **500 Internal Error** is returned. However, if the request doesn't contain this parameter at all, the REST2REST module doesn't throw any exception. The behavior of processing invalid parameters in the REST2REST module should be consistent, deny or pass through the traffic.
13363121	In the clustered environment, an EDR listener occasionally misses the EDRs when connecting to only one server. The cause is that an EDR JMS topic is managed separately by each WebLogic Server instance in the cluster and an EDR listener can receive the message only when it is pinned to the same physical server with the producer. The solution is grouping all JMS servers together using the subdeployment mechanism provided by WebLogic.
12734904	In case of sending SMS to multiple addresses, the delivery receipts (DRs) can be retrieved many times by the query API even if the DRs are notified to clients successfully. ParlayX SMS has the same behavior as OneAPI SMS.
8181940	There is no alarm or event to signal when erroneous authentication events have occurred.
8176540	In the SOAP facade of the Extended Web Services WAP Push/PAP communication service, the error checking of the destination address is faulty. It accepts some poorly-formatted addresses and rejects some valid ones.
8175159	Loading both local and geo-redundant SLAs for the same application group or service provider group having overlapping service contracts will result in undefined behavior as to whether the local or the geo-redundant limits will be enforced. A particular service contract for a particular application group or service provider group may be part of either the local SLA or geo-redundant SLA, never both of them simultaneously. This restriction is not currently enforced by the MBean SLA loading methods

Table 1–2 (Cont.) Known Problems in this Release

Bug ID	Description
8172929	A Services Gatekeeper database improperly sized too small may result in the loss of CDRs and EDRs when the storage is filled. Services Gatekeeper inserts each CDR or EDR to the database by default. If the target table becomes full there will be a delay each time a CDR is inserted into the database. The delay can worsen over time resulting in the internal queue becoming big enough that some EDRs and CDRs are lost. CDR or EDR insertion in a properly configured database does not fail and will not result in the lost of data records.
8152797	A small number of entities are left as artifacts when transactions are rolled back. For example, after a transaction is rolled back there may be CDRs, database or storage entities and network protocol-specific artifacts left in Services Gatekeeper.

2 Installation

This chapter provides information on installing the Oracle Communications Services Gatekeeper patch. Before installing the Services Gatekeeper patch, read *Oracle Communications Services Gatekeeper Installation Guide*.

About This Patch

This patch upgrades Oracle Communications Services Gatekeeper.

The patch contains bug fixes and enhancements. See Chapter 1, "Introduction," and *Oracle Communications Services Gatekeeper OneAPI Developer's Guide* for information on bug fixes, known issues, and enhancements.

Downloading the Patch

Download the Services Gatekeeper patch installer from the Oracle software delivery Web site.

Installing the Patch

The patch includes a complete Services Gatekeeper installer.

To install the patch in a new environment, follow the instructions available in *Oracle Communications Services Gatekeeper Installation Guide*. To upgrade an existing Services Gatekeeper installation, you must have already installed a previous version of Services Gatekeeper. See the chapter on upgrading Services Gatekeeper in *Oracle Communications Services Gatekeeper Installation Guide* for information on upgrading an existing installation.

See *Oracle Communications Services Gatekeeper Installation Guide* for supported hardware configurations.

Installer Changes

There are minor changes to the list of deployed packages in the patch installer. These include:

- Default deployment of modules associated with OneAPI
- Default deployment of modules associated with OAuth 2.0

3 Using Services Gatekeeper with REST Services

This chapter describes how Oracle Communications Services Gatekeeper can be used with existing RESTFul Application Services.

Basic Concepts

Service providers may have existing third-party or proprietary applications or platforms that communicate using REST web services. Services Gatekeeper functionality can be integrated with existing applications that support REST interfaces by creating a RESTFul communication service.

Services Gatekeeper supports two types of RESTFul communication services. A **REST2REST** service exposes an existing REST API allowing communication between RESTFul interfaces. A **REST Exposure** or **empty** service is an application bound, network-facing service used when RESTFul requests are sent to a custom network implementation for translation and processing.

Services Gatekeeper mediates traffic between users and existing REST infrastructure allowing the application of service level agreements, policy enforcement, security, alarms and statistics for more control over communication services.

For more information on communication services, see *Communication Service Guide*.

About the Eclipse Wizard

The Eclipse wizard is a plug-in that enables an Eclipse user to create Services Gatekeeper communication services. The extension projects are created using wizards that customize the project depending on which type of extension is being developed.

The Eclipse wizard generates REST communication services from Web Application Description Language (WADL) files representing RESTFul web application services. Services Gatekeeper then uses the generated service to handle RESTFul communications between two platforms.

The Eclipse wizard is used to create both REST2REST and REST Exposure services.

For more information on using the Eclipse wizard see the configuration procedure in *Platform Development Studio Developer's Guide*.

Generating a REST2REST Communication Service

Generating a REST2REST communication service follows a similar process as other communication services. The Eclipse wizard is used to create a REST2REST communication service.

Generating a Communication Service Project

A REST2REST communication service project is based on a WADL file and a set of attributes given when you run the **Communication Service Project** wizard.

To generate a REST2REST communication service project:

1. In Eclipse, choose **New Project** from the **File** menu.

   This opens the **New Project** window.
2. Expand **OCSG Platform Development Studio**.
3. Select **RESTFul Communication Service Project**.
4. Click **Next** to proceed.

   You may cancel the wizard at any time by clicking **Cancel**. You may go back to a previous window by clicking **Back**.
5. Enter a **Project Name** and choose a location for your project.
6. Click **Next** to continue.

   The **Define the RESTFul communication Service** window is displayed.
7. Click the **Add** button for each WADL file that includes the service definition to be implemented by the new Communication Service.
8. Browse to the WADL file, select it, and click **OK**. To select all WADL files in a directory use ***.wadl**
9. For each WADL file that includes the callback service definition to be used by the new Communication Service in sending information to the service provider's application:

   Click the second **Add** button, browse the callback WADL file, select it and click **OK**.
10. Define the **RESTful Communication Service Properties**:
    - **Company:** Set your company name, to be used in META-INF/MANIFEST.MF.
    - **Version:** Set the version, to be used in META-INF/MANIFEST.MF.
    - **Identifier:** Create an identifier to tie together a collection of Web Services. Will be a part of the names of the generated war and jar files and the service type for the Communication Service:

      communication_service_identifier.war and *communication service identifier_* callback.jar
    - **Service Type:** Set the service type. Used in EDRs, statistics, etc. For example: Rest2RestXsi_Actions
    - **Java Class Package Name:** Set the package names to be used. For example: oracle.ocsg.rest2rest
    - **Web Services Context path:** Set the context path for the Web Service.For example: /xsi_actions

- **REST to REST:** Check this box to generate a Rest2Rest Communication Service.

11. Click **Finish** to create the project.

Adding and Removing Plug-ins to a RESTFul Communication Service Project

For information on managing plug-ins in a RESTFul Communication Service Project see the discussion on adding and removing plug-ins in *Platform Development Studio Developer's Guide*.

Generating a REST Exposure Communication Service

Generating a REST Exposure communication service follows a similar process as REST2REST communication services. Use the Eclipse wizard to create a REST Exposure communication service.

To generate a REST Exposure communication service, follow the procedure in "Generating a Communication Service Project" without checking the **REST to REST** box. An empty plugin can be added to the REST Exposure service during communication service creation.

4 Filtering Tunneled Parameters

This chapter describes an application that filters tunneled parameters. This enhancement enables tighter security by allowing only xparameters that are explicitly allowed.

For general information about xparameters, see the discussion of parameter tunneling in *Application Developer's Guide*.

About the XParameter Filter Application

The filter application blocks requests that contain xparameters that are not configured as allowed xparameters. Filtering is on a global, not application, level.

The application is implemented by the **x-param-filter-interceptor** interceptor. When the application is on and Services Gatekeeper starts up, the application reads a configuration file that lists the allowed xparameters. If the list of allowed xparameters changes, you must update the configuration file and redeploy the filtering application.

The application is deployed as a standalone EAR file: **interceptor_xparam/xparam_interceptors.ear**.

The filter application, named **interceptor_xparam**, is installed with Services Gatekeeper. To turn the application on, configure the customized interceptor chain and enable **interceptor_xparam**. When the filter application is on, you need to configure the xparameters that you want to allow as described in "XParameter Filter Configuration File"; otherwise all requests that contain xparameters will be rejected.

XParameter Filter Configuration File

The xparameter filter configuration file is **xparam_filter_config.xml**. It is located in **interceptor_xparam/xparam_interceptors.ear./APP-INF/classes**.

To allow an xparameter in a request, list its key as an `<xParamKey>` sub element in the `<allowedXParams>` element in **xparam_filter_config.xml**. For example:

```
<allowedXParams>
 <xParamKey>sms.protocol.id</xParamKey>
 <xParamK>sms.service.type<xParamKey>
 . . .
</allowedXParams>
```

The xparameter keys are listed by communication service in the sections on tunneled parameters in the chapters in the *Communication Service Guide*. Some communication services do not support any xparameters.

XParameter Rejection

If an xparameter is not configured in **xparam_filter_config.xml**, a SOAP request that passes it will be rejected.

The following is an example of a rejection response to a request that passed the dest_ addr_subunit xparameter:

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
 <env:Header/>
 <env:Body>
    <env:Fault>
       <faultcode>env:Server</faultcode>
       <faultstring/>
       <detail>
          <v2:ServiceException
xmlns:v2="http://www.csapi.org/schema/parlayx/common/v2_1">
             <messageId>SVC0001</messageId>
             <text>A service error occurred. Error code is %1</text>
             <variables>Error validating the request, xparam: dest_addr_subunit in
the request data is not allowed.</variables>
          </v2:ServiceException>
       </detail>
    </env:Fault>
 </env:Body>
</env:Envelope>
```

See "Internal XParameters" for information about exceptions for xparameters that are not rejected.

Internal XParameters

It is possible for custom interceptors to insert xparameters on behalf of an application. To prevent these internal xparameters from being rejected by the filter, make sure that any custom interceptor that adds xparameters is executed after the **x-param-filter-interceptor** in the interceptor stack

See the discussion of the service interceptors in *Platform Development Studio Developer's Guide* for information about the order of execution of custom interceptors.

Xparameters added to a request through the `<contextAttribute>` of an SLA are not rejected because the **InjectValuesInRequestContextFromSLA** interceptor executes after **x-param-filter-interceptor**.

5 Using the UCP API

This chapter provides an overview of the Services Gatekeeper Universal Computer Protocol (UCP) API Java interface. It also contains some guidance on how to develop a customized UCP plug-in using the Services Gatekeeper Platform Development Studio and the UCP APIs.

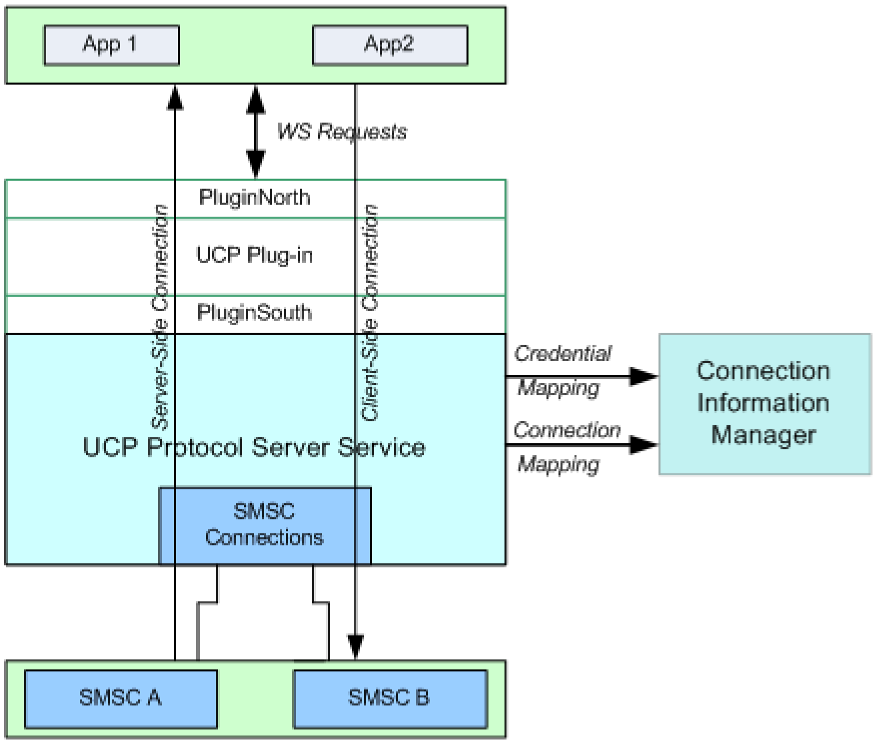
UCP API Overview

The UCP protocol APIs enable platform developers to create custom UCP plug-ins without having to set up and manage connections from Services Gatekeeper to applications and SMSCs.

The UCP Protocol Server Service manages the low-level connectivity details, in conjunction with a configurable Connection Information Manager service, which stores mappings between plug-in instances and the hosts and ports and mappings between application instances and network node credentials.

Using the Protocol Server Service APIs, a plug-in obtains a connection to an application or SMSC and sends a protocol data unit (PDU) or acknowledgement on that connection. The APIs include classes for constructing UCP PDUs.

Figure 5–1 UCP Architecture



A client-side connection is a connection between Services Gatekeeper and the SMSC, since Services Gatekeeper acts as client in this relationship. In the context of this architecture, a server-side connection is a connection between an application and Services Gatekeeper, since Services Gatekeeper acts as server in this relationship.

UCP Protocol Server Service

The UCP Protocol Server Service provides connection services on behalf of UCP plug-ins. It communicates with external applications and SMSCs using UCP over TCP/IP. This service:

- Sends and receives UCP data from the socket.
- Constructs the UCP PDU.
- Associates the current PDU with the correct application instance.
- Calls the plug-in.

All requests from a plug-in instance to the Protocol Server Service contain a plug-in instance ID. The Protocol Server Service performs connection and network credential mapping based on the configuration set up in the Connection Information Manager.

The UCP Protocol Service API defines the interface between the UCP Protocol Server Service and UCP plug-ins. See the oracle.ocsg.protocol.ucp and oracle.ocsg.protocol.ucp.pdu packages in the *Services Gatekeeper Services Gatekeeper Java API ReferenceServices Gatekeeper Java API ReferenceJava API Reference* for documentation of this API.

The Protocol Server Service is a standard Services Gatekeeper WLS service. You can access it from the Administration console as **UCPService** under **Container Services**.

Connection Information Manager

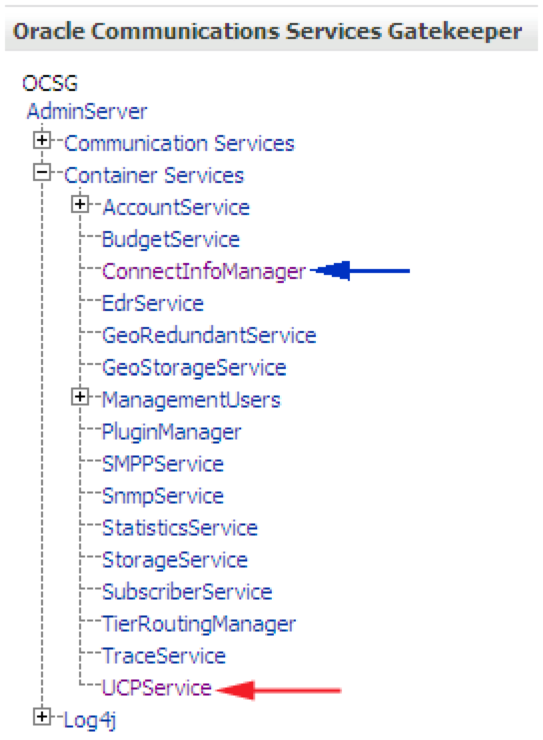
The Connection Information Manager is a standard Services Gatekeeper service, which creates and stores connection and credential mappings that UCP plug-in instances need to connect to network elements and applications.

The UCP Protocol Service uses the Connection Information Manager to map plug-instance IDs to SMSC IP addresses and ports.

You can also optionally configure in the Connection Information Manager the local address and port to bind to when setting up a client-side connection to an SMSC. When Services Gatekeeper connects to the remote network node, it uses the specified local host IP address and port combination to bind the socket on the Services Gatekeeper side of the connection. The Protocol Service uses the specified port as a starting offset and increments the port number by one for each additional connection additional associated with the same plug-in instance ID. If the local host address is not configured, an ephemeral port is used.

You manage connection information settings from the Administration console. See **ConnectInfoManager** under **Container Services**, as shown in Figure 5–2. See the discussion of managing and configuring connection information in *Services Gatekeeper System Administrator's Guide* for information about specific operations.

Figure 5–2 UCP Protocol Server Service and Connection Information Manager in the Administration Console



```
Oracle Communications Services Gatekeeper
OCSG
AdminServer
 ⊞-Communication Services
 ⊟-Container Services
    ⊞-AccountService
     --BudgetService
     --ConnectInfoManager
     --EdrService
     --GeoRedundantService
     --GeoStorageService
    ⊞-ManagementUsers
     --PluginManager
     --SMPPService
     --SnmpService
     --StatisticsService
     --StorageService
     --SubscriberService
     --TierRoutingManager
     --TraceService
     --UCPService
 ⊞-Log4j
```

PluginNorth

A plug-in implements the PluginNorth interface to perform the following tasks on behalf of application-initiated requests:

- Send a mobile-terminated (MT) SMS message

- Open a UCP session
- Send an ACK to the SMSC
- Send a NACK to the SMSC

You would extend and implement this interface to add a new application-facing UCP protocol plug-in.

PluginSouth

A plug-in implements the PluginSouth interface to perform the following tasks on behalf of network-triggered requests:

- Deliver a mobile-originated (MO) SMS message
- Deliver a message delivery notification associated with a previously-sent MT SMS
- Send an ACK to the application
- Send a NACK to the application

You would extend and implement this interface to add a new network-facing UCP protocol plug-in.

Additional Information You Will Need

In addition to the information in this chapter, developers should consult the following documents for information on how to build a UCP plug-in:

- *Services Gatekeeper Java API Reference*

  Of special interest are the following packages, which include the interfaces and classes for the UCP Protocol Server Service:

  – oracle.ocsg.protocol.ucp

  – oracle.ocsg.protocol.ucp.pdu

  – oracle.ocsg.protocol.common

  The following packages include the plug-in interfaces and classes for the Native SMPP plug-in, which is part of the standard Services Gatekeeper Native UCP communication service. They can serve as a reference for developing customized north and south UCP plug-ins.

  – oracle.ocsg.plugin.nativefacade.ucp.north

  – oracle.ocsg.plugin.nativefacade.ucp.south

  In addition, you will need resources from various generic packages such as:

  – com.bea.wlcp.wlng.api.edr

  – com.bea.wlcp.wlng.api.management,

  – com.bea.wlcp.wlng.api.plugin

  – com.bea.wlcp.wlng.api.plugin.common

  – com.bea.wlcp.wlng.api.plugin.context

  – com.bea.wlcp.wlng.api.util

- *Services Gatekeeper Platform Development Studio Developer's Guid*e

This guide explains how to use the Platform Development Studio to create a communication service or plug-in. See the following topics:

  - Understanding communication services
  - Using the Eclipse wizard
  - Description of a generated project

- *Services Gatekeeper Communication Service Guide*

  See the Native UCP chapter. This chapter provides an overview of the Services Gatekeeper Native UCP communication service. It documents the attributes and operations provided to manage the UCP Protocol Server Service. The protocol server service is available for any UCP plug-in to access using the UCP Protocol Server Service APIs.

- *Services Gatekeeper System Administrator's Guide*

  See the connection information chapter. The Connection Information Manager creates and stores connection and credential mappings used by UCP plug-ins.

Procedure for Creating a Customized UCP Plug-in

The following procedure outlines the basic steps to perform to add a custom UCP plug-in.

1. Using the Services Gatekeeper SCE PDS Eclipse wizard, generate a customized network plug-in for the UCP communication service.

   You can also use this wizard to create a custom interceptor, if necessary.

   See the description of a generated project in *Services Gatekeeper Platform Development Studio Developer's Guide*.

2. Create the service type for the customized plug-in by extending the ServiceType class.

   When the plug-in registers itself, an object of this type is passed to the Plug-in Manager.

3. Implement the ManagedPluginService interface. This class activates, deactivates and initializes the plug-in service. It implements the PluginService, PluginServiceLifecycle and PluginInstanceFactory interfaces.

   See the discussions of communication services and generated projects in *Services Gatekeeper Platform Development Studio Developer's Guide*.

4. Implement the ManagedPluginInstance interface.

   This class activates a plug-in instance that has been created with the Plug-in Manager, after which the plug-in should register its MBeans and prepare to accept traffic. The plug-in service that activates this plug-in instance must be in the ACTIVE (ADMIN) or ACTIVE (RUNNING) state when the **activate** method is called.

   This class also initializes and deactivates the plug-in instance and determines whether the plug-in instance is capable of servicing the current request.

   See the discussion of communication services in *Services Gatekeeper Platform Development Studio Developer's Guide*.

5. If you are implementing an application-facing UCP module, extend and implement the PluginNorth interface: **SubmitSm**, **openSession**, **ack** and **nack**.

6. If you are implementing a network-facing UCP module, extend and implement the PluginSouth interface: **deliverSm**, **deliveryNotification**, **ack** and **nack**.
7. Create CDRs and EDRs to trace the message flow, if necessary.
8. From the Administration console, configure the connection and credential mappings in the Connection Information Manager.

   See the discussion of managing and monfiguring connection information in *System Administrator's* Guide.
9. Build the plug-in project and create the EAR package, which will be deployed to Services Gatekeeper.
10. Use the Platform Test Environment (PTE) to test and debug the plug-in.

    See *Services Gatekeeper Platform Test Environment Guide*.

About the UCP Protocol Server Service Interfaces

The packages for the protocol server service are:

- oracle.ocsg.protocol.common
- oracle.ocsg.protocol.ucp
- oracle.ocsg.protocol.ucp.pdu

Using the UCP Protocol Server Service API, you can develop a custom UCP plug-in without having to implement the low-level connection functionality. The API provides a wrapper to bind, send, and receive messages and allows customization of PDUs.

oracle.ocsg.protocol.common

The oracle.ocsg.protocol.common.package provides four basic interfaces from which the UCP Protocol Server Service APIs are derived:

- AbstractProtocolService

  This is the base class for the **getProtocolServiceNorth** and **getProtocolServiceSouth** methods. The UCPNetworkingServiceImpl class inherits from AbstractProtocolService to implement these methods.
- ProtocolServiceProxyFactory

  Gets references to the **getProtocolServiceNorth** and **getProtocolServiceSouth** methods for use by the plug-in.
- ProtocolServiceNorth

  This is the base interface for creating server-side connections.
- ProtocolServiceSouth

  This is the base interface for creating client-side connections.

oracle.ocsg.protocol.ucp

The main protocol server service interfaces used by a UCP plug-in are:

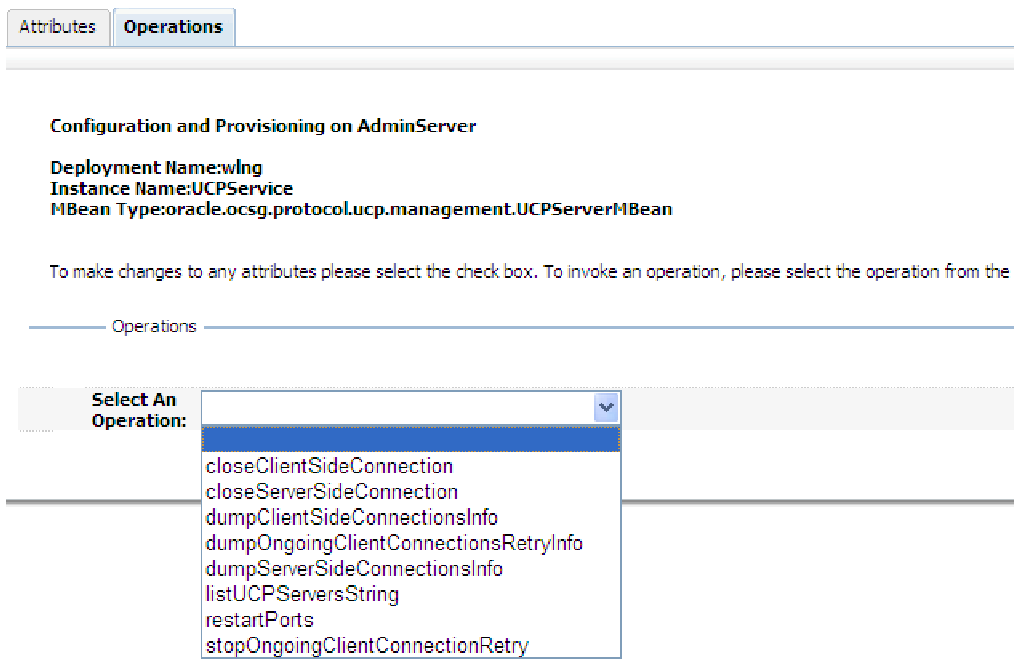
- UCPNetworkingService

  The UCPNetworkingService interface provides methods to add, remove, list and otherwise manage server-side and client-side connections. See the

UCPNetworkingService interface in the oracle.ocsg.protocol.ucp package in the *Services Gatekeeper Java API Reference* for a list of the methods in this interface.

In addition to accessing these methods programmatically, a System Administrator can also access most of the methods in the UCPNetworkingService interface as OAM operations from the **UCPService** pane of the Administration console.

Figure 5–3 Protocol Service Operations in Administration Console



Configuration and Provisioning on AdminServer

Deployment Name:wlng
Instance Name:UCPService
MBean Type:oracle.ocsg.protocol.ucp.management.UCPServerMBean

To make changes to any attributes please select the check box. To invoke an operation, please select the operation from the

Operations
Select An Operation:
closeClientSideConnection
closeServerSideConnection
dumpClientSideConnectionsInfo
dumpOngoingClientConnectionsRetryInfo
dumpServerSideConnectionsInfo
listUCPServersString
restartPorts
stopOngoingClientConnectionRetry

- UCPNetworkingServiceClient

  This interface implements methods for sending PDUs, ACKs, and NACKs on a client-side connection. It extends the Services Gatekeeper oracle.ocsg.protocol.common.ProtocolServiceSouth interface.

  The plug-in uses this interface's **sendPDUOnClientConnection** method to send the plug-in instance ID and the PDU. The method returns a connection ID that identifies the connection to the SMSC on which the request was sent.

- UCPNetworkingServiceServer

  This interface implements methods for sending PDUs, ACKs, and NACKs on a server-side connection. It extends the Services Gatekeeper oracle.ocsg.protocol.common.ProtocolServiceNorth interface.

- The plug-in uses the **sendPDUOnServerConnection** method to send the connection ID and the PDU. The method returns a connection ID that identifies the connection to the application on which the request was sent.

oracle.ocsg.protocol.ucp.pdu

This package provides utility classes for building UCP PDUs for the supported UCP operations. This package provides classes for all of the supported UCP abstract data types (ADTs), as well as a generic UCP ADT, UCP constants, headers, and parameters

Connection Mapping

The Protocol Server Service uses mappings between application instances to network nodes configured in the Connection Information Manager to set up the connections that are used by the plug-ins.

At a minimum you need to configure the credential map, host address, and user password using these operations:

- **createOrUpdateCredentialMap**
- **createOrUpdateRemoteHostAddress**

  or

  createOrUpdateLocalHostAddress
- **createOrUpdateListenAddress**
- **createOrUpdateUserPasswordCredentialEntry**

There are various possible mapping logics; for example:

- One connection to the SMSC for all Services Gatekeeper applications
- One connection to the SMSC for a group of Services Gatekeeper applications
- One connection to the SMSC for each Services Gatekeeper application

The simplest scenario is to configure a plug-in always to use the same application instance for all UCP requests. This requires only one connection to the SMSC. You would create the application instance in Services Gatekeeper and dedicate it to UCP southbound requests in the Connection Information Manager. Before making the call to the UCP Protocol Server Service, the plug-in can switch context to the UCP-dedicated application instance

Another scenario would configure the plug-in to use the current application instance to send requests through the service. This results in multiple connections to the SMSC, at least one per application instance. In this case, you must configure the Connection Information Manager with connection credentials and SMSC address and port mappings for all application instances.

There is no single correct solution. The mapping logic that you choose depends on the demands of your situation.

OAM Attributes Affecting UCP Network Connectivity

The **UCPProtocol** read-only attribute contains the UCP protocol string. This value is set to the listen address defined by the **createOrUpdateListenAddress** operation in the Connection Information Manager.

In the Administration console, you can configure two attributes that control how the Protocol Server Service handles reconnection attempts:

- **MaxReconnectAttempts**: Specifies the maximum number of reconnection attempts permitted. Set to -1 for no maximum, 0 for no reconnection attempts, or a positive integer indicating the maximum number of reconnections to attempt.
- **TimeBetweenReconnectAttempts**: Specifies the time in milliseconds between recconnection attempts.

Using the APIs

The first three examples in this section provide some guidance related to common tasks using the UCP APIs that would be performed by a custom application-facing UCP plug-in that implements and extends PluginNorth. The examples are based on a prototype for a ParlayX2.1 SMS plug-in. The last example is for a PluginSouth implementation processing a **DELIVER_SM** request.

The tasks include:

- Sending a submitSm Request to the SMSC
- Creating a UCP PDU
- Sending an openSession Request to the SMSC
- Sending a DeliverSm to an Application

Sending a submitSm Request to the SMSC

When a plug-in receives a **SUBMIT_SM** request from an application, the PluginNorth implementing class processes the parameters in the request, constructs the PDU, and sends it to the SMSC using the UCPNetworkingServiceClient APIs.

The plug-in:

1. Gets the UCP NetworkingService object. For example:

```
UCPNetworkingService ucpService =
PX21UCPPluginInstanceImpl.getUCPNetworkingService();
```

2. Gets any outstanding standing **SUBMIT_SM** requests.
3. Creates the submit PDU, using the classes in the oracle.ocsg.protocol.ucp.pdu package. See "Creating a UCP PDU".
4. Gets the source connection ID.
5. Gets the UCP NetworkingService protocol interface for sending data on a client connection. For example:

```
UCPNetworkingServiceClient client =
ucpService.getProtocolServiceSouth(UCPNetworkingServiceClient.class);
```

6. Sends the PDU on the client connection, using the UCPNetworkingServiceClient **sendPDUOnClientConnection** method. For example:

```
clientConnectionID = client.sendPDUOnClientConnection
(
    px21UCPPluginInstanceImpl.getPluginInstanceId(),
    px21UCPPluginInstanceImpl.getSourceServerPort(),
    submitSMPDU,
    outstandingSubmitSMRequests,
    sourceConnectionID
);
```

Creating a UCP PDU

To create a UCP PDU, you can use the classes in the oracle.ocsg.protocol.ucp.pdu package.

The following method creates the submitSM PDU used in "Sending a submitSm Request to the SMSC". It uses the using the UcpHeader, UcpParameter, GenericUcpAdt classes defined in the pdu package.

```
private UcpPDU createSubmitSMPDU(SendSms parameters) {
        UcpHeader ucpHeader = new UcpHeader();
        UcpParameter orParam = new UcpParameter("O");
        ucpHeader.setParameter(UcpHeader.PARAM_OR, orParam);
        UcpParameter otParam = new UcpParameter(UcpConstants.OT_SUBMIT_SHORT_
MESSAGE);
        ucpHeader.setParameter(UcpHeader.PARAM_OT, otParam);
        UcpParameter trnParam = new UcpParameter("01");
        ucpHeader.setParameter(UcpHeader.PARAM_TRN, trnParam);
        UcpParameter lenParam = new UcpParameter("00000");
        ucpHeader.setParameter(UcpHeader.PARAM_LEN, lenParam);

        GenericUcpAdt data = new GenericUcpAdt(33);

        //ADC
        URI[] destAddresses = parameters.getAddresses();
        String uriStringDestAddress = destAddresses[0].toASCIIString();

        //Strip "tel:"
        String destAddressString = stripURIPrefix(uriStringDestAddress);
        UcpParameter adcParam = new UcpParameter(destAddressString);
        data.setParameter(Ucp50Adt.PARAM_ADC, adcParam);

        //OADC
        String senderName = parameters.getSenderName();
        UcpParameter oadcParam = new UcpParameter(senderName);
        data.setParameter(Ucp50Adt.PARAM_OADC, oadcParam);

        //NRQ and NT
        SimpleReference simpleRef = parameters.getReceiptRequest();
        String nrq = "";
        String nt = "";
        if(simpleRef != null){
            nrq = "0"; //0 == NADC not used
            nt = "7"; // 7 == all
        }
        UcpParameter nrqParam = new UcpParameter(nrq);
        data.setParameter(Ucp50Adt.PARAM_NRQ, nrqParam);
        UcpParameter ntParam = new UcpParameter(nt);
        data.setParameter(Ucp50Adt.PARAM_NT, ntParam);

        //If LRq is empty, the contents of LRAd and LPID are ignored

        //Message type 3 == "Alphanumeric message encoded into IRA characters."
        UcpParameter mtParam = new UcpParameter("3");
        data.setParameter(Ucp50Adt.PARAM_MT, mtParam);

        String message = parameters.getMessage();
        String iraEncodedMessage = iraEncodeMessage(message);
        UcpParameter msgParam = new UcpParameter(iraEncodedMessage);
        data.setParameter(Ucp50Adt.PARAM_MSG, msgParam);

        return new UcpPDU(ucpHeader, data);
    }
```

Sending an openSession Request to the SMSC

A connection from the UCP plug-in to the SMSC is implicitly established on receipt of the openSession request. Upon receiving the openSession request, the Protocol Server Service uses the current context as a key to determine the connection and credential mapping to use for the new connection that it is creating. The user and plug-in instance ID must therefore be configured in the Connection Information Manager before the openSession request is sent; otherwise the openSession request will fail.

The APIs do not provide a specific open session method.

To create a new session to the SMSC, create an openSession PDU using the pdu package and use the **sendPDUOnClientConnection** with a that openSessionPDU as the PDU parameter:

```
UCPNetworkingServiceClient client =
ucpService.getProtocolServiceSouth(UCPNetworkingServiceClient.class);

String connectionID = client.sendPDUOnClientConnection
    (myUCPPluginInstanceImpl.getPluginInstanceId(),
     sourceServerPort,
     openSessionPDU,
     outstandingOpenSessionRequests,
     sourceConnectionId);
```

A UCP plug-in uses the Protocol Server Service API after it receives an openSession PDU. The UCP Protocol Server Service creates a new socket connection for each session management operation of subtype openSession that is sent. The created connections are later used for sending **SUBMIT_SM** requests.

Sending a DeliverSm to an Application

When a plug-in receives a **DELIVER_SM** request from the SMSC, the PluginSouth implementing class processes the parameters in the request.

If the plug-in is communicating with a web services-based application, it typically analyzes the parameters in the request to find the correct application callback reference (URL) to which the mobile-originated SMS message should be sent and then sends it.

If the plug-in is communicating with a UCP-based application, it typically constructs a DELIVER_SM PDU, which it sends to the application-facing UCP NetworkingServerService APIs.

After notifying the application of the message, the plug-in should send an ACK or NACK to the SMSC to report whether the notification was successful.

The following process flow is for a plug-in communicating with a web services-based application:

1. Gets the UCP NetworkingService object. For example:

   ```
   UCPNetworkingService ucpService =
   PX21UCPPluginInstanceImpl.getUCPNetworkingService();
   ```

2. Processes the incoming deliverSM PDU to get the source and destination addresses. This implementation uses the UCP50Adt class to extract the data from the PDU:

   ```
   String destinationAddress = deliverSMPDU.getData().getParameter(Ucp50Adt.PARAM_
   ADC).getValueAsString();
   String originatingAddress = deliverSMPDU.getData().getParameter(Ucp50Adt.PARAM_
   OADC).getValueAsString();
   ```

3. Gets the notification callback references.
4. Implements support for using criteria and storing the mobile-originated message.
5. Creates the deliverSM PDU.

6. Send the deliverSM notification PDU. For example:

```
boolean notificationOK = sendDeliverSMNotification(callbackRef,
destinationAddress, originatingAddress, deliverSMPDU);
```

7. Send ACK or NACK to the SMSC depending on the outcome of the notification. For example:

```
if(notificationOK){
    sendAck(ucpService, connectionId, deliverSMPDU);
}else{
    sendNack(ucpService, connectionId,deliverSMPDU, UcpConstants.ERROR_CODE_
SYNTAX_ERROR);
            }
```

6
Using the SMPP API

This chapter provides an overview of the Services Gatekeeper Short Messaging Peer to Peer Protocol (SMPP) API Java interface. It also contains some guidance on how to develop a custom SMPP plug-in using the Services Gatekeeper Platform Development Studio and the SMPP APIs.

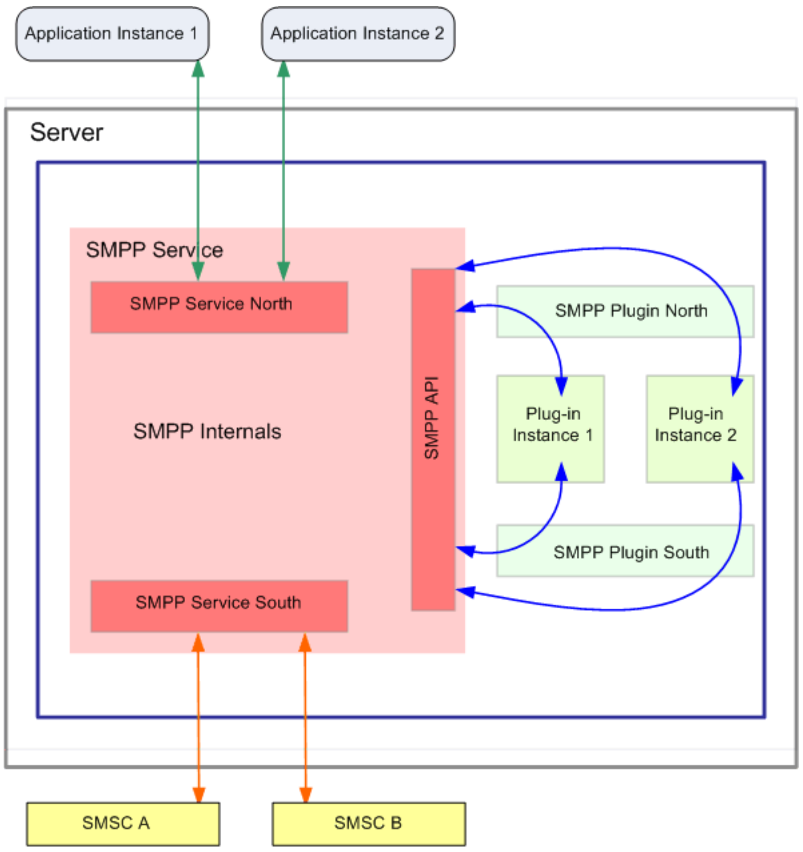
SMPP Overview

The Services Gatekeeper SMPP implementation depends on a core module, the SMPP Service, which provides connectivity services for SMPP plug-ins. The SMPP API defines the interfaces between the plug-ins and the SMPP Service.

Using this API, platform developers can create SMPP plug-ins without having to manage the low-level tasks of connecting from Services Gatekeeper to applications and to SMSCs.

Figure 6–1 illustrates the basic Services Gatekeeper SMPP architecture.

Figure 6–1 SMPP Architecture



SMPP Service Interfaces

The SMPP Service performs connection services on behalf of the standard SMPP plug-ins – Native SMPP and ParlayX 2.1 SMPP – as well as any custom SMPP plug-ins.

The SMPP Service handles the following tasks:

- Receives SMPP data from the socket.
- Constructs the SMPP protocol data unit (PDU).
- Associates the current PDU with the correct application instance.
- Invokes the plug-in.
- Manages connections between Services Gatekeeper and applications.
- Manages connections between Services Gatekeeper and Short Message Service Centers (SMSCs).

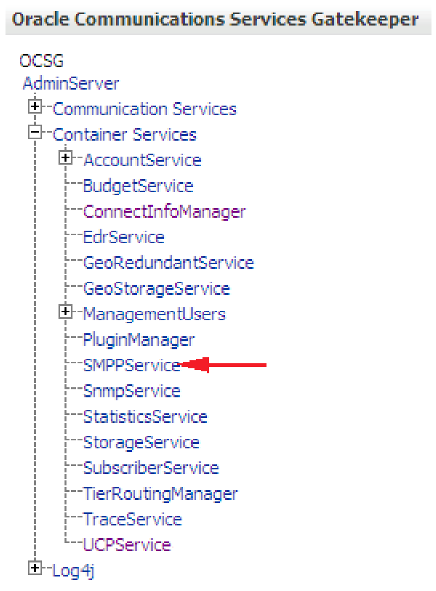
See the oracle.ocsg.protocol.smpp.service package in the *Services Gatekeeper Java API Reference* for documentation of the SMPP Services interfaces.

The SMPPServiceNorth interface processes requests received from an application-facing plug-in and sends them to the application.

The SMPPServiceSouth interface processes requests received from a network-facing plug-in and sends them to the SMSC.

The SMPP Service is a standard Services Gatekeeper WebLogic Server (WLS) service. You can access its Operations, Administration, and Maintenance (OAM) functions from the Administration console as **SMPPService** under **Container Services**.

Figure 6–2 SMPP Service in the Administration Console



SMPPPluginSouth

The SMPPluginSouth interface processes network-triggered operations received from SMPPServiceSouth and sends them to SMPPServiceNorth. You would extend and implement this interface to add a new network-facing SMPP protocol.

SMPPPluginNorth

The SMPPluginNorth interface processes requests received from SMPPServiceNorth and sends them to SMPPServiceSouth. You would extend and implement this interface to add a new application-facing SMPP protocol.

Additional Information You will Need

In addition to the information in this chapter, developers should consult the following documents for information on how to build an SMPP plug-in:

- *Services Gatekeeper Java API Reference*

  Of special interest are the following packages, which include the interfaces and classes for the SMPP service and plug-ins:

  – oracle.ocsg.protocol.smpp.service
  – oracle.ocsg.protocol.smpp.plugin
  – oracle.ocsg.protocol.smpp.event
  – oracle.ocsg.protocol.smpp.common
  – oracle.ocsg.protocol.common

In addition, you will need resources from various generic packages such as:

- com.bea.wlcp.wlng.api.edr
- com.bea.wlcp.wlng.api.management
- com.bea.wlcp.wlng.api.plugin

- *Services Gatekeeper Platform Development Studio Developer's Guide*

  This guide explains how to use the Platform Development Studio to create a communication service or plug-in. See the following topics:

  - Understanding communication services
  - Using the Eclipse wizard
  - Description of a generated project

- *Services Gatekeeper Communication Service Guide*

  See the Native SMPP chapter. This chapter provides an overview of the Services Gatekeeper Native SMPP communication service, which uses the SMPP Service. This chapter includes general information about how the SMPP Service handles connectivity and documents the configurable attributes and operations of the SMPP Service.

Procedure for Creating a Custom SMPP Plug-in

The most common task is to add a custom network-facing SMPP plug-in using the south interfaces. It is also possible to create a custom application-facing SMPP module using the north interfaces. The following procedures cover both scenarios.

The basic steps for creating a custom SMPP plug-in are as follows:

1. Using the Services Gatekeeper SCE PDS Eclipse wizard, generate a customized network plug-in for the SMPP communication service.

   You can also use this wizard to create a custom interceptor, if necessary.

   See the description of a generated project in *Platform Development Studio Developer's Guide*.

2. Create the service type for the customized plug-in by extending the ServiceType class.

   When the plug-in registers itself, an object of this type is passed to the Plug-in Manager.

3. Implement the ManagedPluginService interface.

   This class activates, deactivates and initializes the plug-in service. It implements the PluginService, PluginServiceLifecycle and PluginInstanceFactory interfaces.

   See the discussions of communication services and generated projects in *Services Gatekeeper Platform Development Studio Developer's Guide*.

4. Implement the ManagedPluginInstance interface.

   This class activates a plug-in instance that has been created with the Plug-in Manager, after which the plug-in should register its MBeans and prepare to accept traffic. The plug-in service that activates this plug-in instance must be in the ACTIVE (ADMIN) or ACTIVE (RUNNING) state when the **activate** method is called.

This class also initializes and deactivates the plug-in instance, determines whether the plug-in instance is capable of servicing the current request, and sets up the session information cache.

See the discussion of communication services in *Services Gatekeeper Platform Development Studio Developer's Guide*.

5. Extend and implement the SMPPPluginMBean interface and register the MBean using the SMPP API.
6. If you are implementing a network-facing SMPP module, extend and implement the SMPPPluginSouth interface to process network-triggered events received from SMPPServiceSouth. See the oracle.ocsg.protocol.smpp.plugin package in *Services Gatekeeper Java API Reference* for the list of methods in this interface. See also Using the SMPP APIs.
7. Send the processed requests and responses to the application using the SMPPServiceNorth interface.
8. If you are implementing an application-facing SMPP module, extend and implement the SMPPPluginNorth interface to process application-initiated events received from SMPPServiceNorth. See the oracle.ocsg.protocol.smpp.plugin package in *Services Gatekeeper Java API Reference* for the list of methods in this interface. See also Using the SMPP APIs.
9. Send the processed requests and responses to the SMSC using the SMPPServiceSouth interface.
10. Maintain a session information class to cache session values such as client and server connection IDs, source and destination addresses, whether a delivery notification is required, and so on.
11. Create CDRs and EDRs to trace the message flow, if necessary.

    See the discussion of annotations, EDRs, alarms, and CDRs in *Services Gatekeeper Platform Development Studio Developer's Guide*.
12. Build the plug-in project and create EAR package, which will be deployed to Services Gatekeeper.

    Make sure that the **smpp_api.jar** is in the build class path; for example:

    ```
    <path path="${target.dir}/protocol/modules/smpp_
    api/oracle.ocsg.protocol.smpp_api_5.0.0.0.jar"/>
    ```
13. Use the Platform Test Environment (PTE) to test and debug the plug-in.

    See *Services Gatekeeper Platform Test Environment Guide*.

Configuration Settings Affecting SMPP Connections

The System Administrator can configure several attributes that control how the SMPP Service manages connections.

The System Administrator can also set some parameters on how the SMPP Service behaves on a per application basis, such as whether certain operations are allowed after sending a short message or whether network-triggered notification is enabled. These parameters are set using the **addApplicationSpecificSettings** operation.

These settings can affect how requests and responses should be processed before they are sent. The SMPP Service API provides methods for querying some of these settings. See "SMPPService" for more information.

For a complete list of the SMPP Service attributes and operations, see the reference material for the SMPP server service in the Native SMPP chapter in *Services Gatekeeper Communication Service Guide*.

About the SMPP Interfaces

The packages for developing an SMPP plug-in are:

- oracle.ocsg.protocol.common
- oracle.ocsg.protocol.smpp.service
- oracle.ocsg.protocol.smpp.plugin
- oracle.ocsg.protocol.smpp.common
- oracle.ocsg.protocol.smpp.event

oracle.ocsg.protocol.common

The oracle.ocsg.protocol.common package includes the ProtocolServiceProxyFactory interface, which is derived from the AbstractProtocolService class. This is the base class for the **getProtocolServiceNorth** and **getProtocolServiceSouth** methods.

The SMPP plug-in implementations use the **getProtocolServiceNorth** method to get a reference to the interface used to send PDUs to applications on server connections and the **getProtocolServiceSouth** method to get a reference to the interface used to send PDUs to SMSCs on client connections.

This package also includes the ProtocolServiceNorth and ProtocolServiceSouth interfaces from which the SMPPServiceNorth and SMPPServiceSouth interfaces are derived.

oracle.ocsg.protocol.smpp.service

The oracle.ocsg.protocol.smpp.service package includes the interfaces for the SMPP Service:

- SMPPService
- SMPPServiceNorth
- SMPPPluginSouth

SMPPService

This interface provides methods for generic SMPP Service tasks. These include checking whether available or active client connections exist for a plug-in instance, registering the SMPP work manager, and registering the plug-in MBean object, which exposes configurable attributes and operations to the SMPP Service.

It provides methods for querying the following SMPP Service configuration settings:

- **ConnectionBasedRouting**: an attribute in the SMPP service
- **LooseBinding**: an attribute in the SMPP service
- **notificationEnabled**: an application-specific setting in the SMPP Service
- **subsequentOperationsAllowed**: an application-specific setting in the SMPP Service

For details about these settings, see s the reference material for the SMPP server service in the Native SMPP chapter in *Communication Service Guide*.

SMPPServiceNorth

The SMPPServiceNorth interface maintains a server connection pool that provides connections between Services Gatekeeper and applications. Services Gatekeeper is a server in this relationship.

When the application sends a successful **BIND** request to Services Gatekeeper, the plug-in obtains a server connection from the server connection pool and uses the implementation of the SMPPServiceNorth interface to send messages to the application.

The server connection:

- Receives messages from the application.
- Invokes the SMPPPluginNorth interface through a proxy.
- Sends messages to the application.
- Manages SMPP timers and windowing toward the application.
- Stores transaction mapping information in cache.

This interface provides the following methods to send northbound requests and responses submitted by the plug-in: **cancelSmResponse**, **dataSm**, **dataSmResponse**, **deliverSm**, **querySmResponse**, **replaceSmResponse**, **submitSmMultiResponse**, **submitSmResponse**.

SMPPServiceSouth

The SMPPServiceSouth interface maintains a client connection pool that provides connections between Services Gatekeeper and Short Message Service Centers (SMSCs). Services Gatekeeper is a client in this relationship.

The service processes **BIND** and **UNBIND** requests from the plug-in and obtains client connections on which to perform SMPP operations toward the SMSC.

The client connection:

- Receives messages from the SMSC.
- Invokes the SMPPPluginSouth interface through a proxy.
- Sends messages to the SMSC.
- Manages SMPP timers and windowing toward the SMSC.
- Stores transaction mapping information in cache.

This interface provides the following methods to send southbound requests and responses submitted by the plug-in: **bind**, **cancelSm, dataSm**, **dataSmResponse**, **deliverSmResponse**, **querySm**, **replaceSm**, **submitSm**, **submitSmMulti**, **unbind**.

oracle.ocsg.protocol.smpp.plugin

The oracle.ocsg.protocol.smpp.plug-in package defines the interfaces between the SMPP service and the SMPP plug-ins:

- SMPPServiceNorth
- SMPPServiceSouth
- SMPPPluginMBean

The plug-in developer extends and implements these interfaces for a custom plug-in.

SMPPPluginNorth

A plug-in extends and implements the SMPPluginNorth interface to process the following supported application-initiated operations:

- BIND
- CANCEL_SM
- DATA_SM
- DATA_SM_RESPONSE
- DELIVER_SM_RESPONSE
- QUERY_SM
- REPLACE_SM
- SUBMIT_SM
- SUBMIT_SM_MULTI

The SMPPPluginNorth implementation uses the SMPPServiceSouth interface to send these operations to the SMSC.

SMPPPluginSouth

The plug-in extends and implements the SMPPluginSouth interface to process supported network-triggered operations, such as:

- CANCEL_SM_RESPONSE
- DATA_SM
- DATA_SM_RESPONSE
- DELIVER_SM
- QUERY_SM_RESPONSE
- REPLACE_SM_RESPONSE
- SUBMIT_SM_MULTI_RESPONSE
- SUBMIT_SM_RESPONSE
- UNBIND

The SMPPPluginSouth implementation uses the SMPPServiceNorth interface to send these operations to the application.

SMPPPluginMBean

This interface defines the network-facing connection attributes of the plug-in. A custom plug-in extends and implements this interface to provide the facilities to manage and query the plug-in.

The SMPPPluginNorth and SMPPPluginSouth implementations use this interface to query values in the plug-in while processing requests and responses.

oracle.ocsg.protocol.smpp.common

This package provides the SMPPException class.

oracle.ocsg.protocol.smpp.event

This package provides classes for SMPP events.

Using the SMPP APIs

The basic procedure for processing and sending an incoming request or response through the SMPP Service is as follows:

1. Get the SMPPService object.
2. Process the fields in the incoming request or response.

   Depending on the particular request or response typical processing may involve setting various fields in the request or response. For a response, you may need to process event data from the original request.

   It may be necessary to query some SMPP Service configuration settings using the SMPPService methods. See "SMPPService" for more information.
3. Get the SMPPService object's protocol interface for sending data.

   For sending data to the SMSC, you need the interface for SMPPServiceSouth to get a client-side connection. For sending data to the application, you need the interface for SMPPServiceNorth to get a server-side connection.
4. Send the request or response using the methods provided by the SMPPServiceNorth or SMPPServiceSouth.

The following sections illustrate how the SMPP Server APIs and settings are used in processing some requests and responses. They focus on sample tasks involving the SMPP API. Logging, exception handling, session information management, alarm creation, and other tasks not using the SMPP APIs are not considered.

- Processing a BIND Request from an Application
- Processing a SUBMIT_SM Request from an Application
- Processing a SUBMIT_SM Response from the SMSC
- Processing a DELIVER_SM Request from the SMSC
- Processing a DELIVER_SM Response from an Application

These are among the tasks performed in custom SMPPPluginNorth and SMPPPluginSouth implementations.

Processing a BIND Request from an Application

When the plug-in receives a **BIND** request from an application, the SMPPPluginNorth class processes the request and sends it to the SMSC.

The SMPPPluginNorth **bind** method:

1. Gets the plug-in instance id and sets it in the request.
2. Gets the SMPP Service object.
3. Gets the service object's protocol interface for sending data on a client connection.
4. Sends the request using the SMPPServiceSouth's **bind** method.

For example:

```
public BindResponse bind(Bind request) {
    BindResponse bindResp = null;
    // Set the plug-in instance id
    request.setPluginInstanceId(plugin.getPluginInstanceId());

    // Get the SMPP service object
    SMPPService smppService = plugin.getSMPPService();

    // Get the interface for sending data on a client-side connection
    SMPPServiceSouth serviceSouth =
smppService.getProtocolServiceSouth(SMPPServiceSouth.class);
    // Send the request
      bindResp = serviceSouth.bind(request);

      return bindResp;
}
```

Processing a SUBMIT_SM Request from an Application

When a plug-in receives a **SUBMIT_SM** request from an application, the SMPPPluginNorth class processes the request and sends it to the SMSC.

The SMPPPluginNorth **submitSm** method:

1. Gets the plug-in instance and application instance IDs and sets them in the request.
2. Queries the SMPP Service's application-specific **notificationEnabled** setting and sets the **registeredDelivery** field in the request accordingly.

   ```
   if (request.getRegisteredDelivery() != 0 &&
   !plugin.isNotificationAllowed(aigId)) {
         request.setRegisteredDelivery(0);
   }
   ```

3. Gets the SMPP Service object.

   ```
   SMPPService smppService = plugin.getSMPPService();
   ```

4. Gets the service object's protocol interface for sending data on a client connection.

   ```
   SMPPServiceSouth serviceSouth =
   smppService.getProtocolServiceSouth(SMPPServiceSouth.class)
   ```

5. Process any extra parameters (xparams) in the request.
6. Sends the request using the SMPPServiceSouth's **submitSm** method.

   ```
   serviceSouth.submitSm(request);
   ```

Processing a SUBMIT_SM Response from the SMSC

When a plug-in receives a **SUBMIT_SM_RESPONSE** from the SMSC, the SMPPPluginSouth class processes the response and sends it to the application.

The SMPPPluginSouth **submitSmResponse** method:

1. Gets the SMPP Service object.

   ```
   SMPPService smppService = plugin.getSMPPService();
   ```

2. Gets the plug-in message ID and sets it in the response.

3. Gets and processes the request event data from the original request to which this is the response.
4. Queries the SMPP Service and application-specific settings to determine whether a delivery receipt will be provided. For example, the following example checks the **notificationEnabled** and **isSubsequentOperationsAllowed** application-specific settings and the **ConnectionBasedRouting** SMPP Service attribute.

```
boolean needDR = plugin.isNotificationAllowed(aigId) &&
originalRequest.getRegisteredDelivery() != 0 &&
        !plugin.isConnectionBasedRoutingEnabled();
      if (plugin.isSubsequentOperationsAllowed(aigId) || needDR) {
          // Set the session information accordingly . . .
}
```

5. Gets the SMPP Service object's interface for sending data on a server-side connection.

```
SMPPServiceNorth serviceNorth =
smppService.getProtocolServiceNorth(SMPPServiceNorth.class);
```

6. Sends the response on that connection using SMPPService North's **submitSmResponse** method.

```
serviceNorth.submitSmResponse(response);
```

Processing a DELIVER_SM Request from the SMSC

A **DELIVER_SM** request from the SMSC can be a simple SMS message from the network, or it can be the SMSC sending a delivery receipt for a previously submitted **SUBMIT_SM** request.

When a plug-in receives a **DELIVER_SM** request from the SMSC, the SMPPPluginSouth **deliverSm** method first examines the isDeliveryReceipt field in the request to determine whether the request is for a delivery receipt or a network-triggered SMS message. For example:

```
public void deliverSm(final DeliverSm request)  {
    request.setPluginInstanceId(plugin.getPluginInstanceId());
    final boolean isDeliverReceipt = request.isDeliverReceipt();

    if (isDeliverReceipt) {
      deliverSmForDeliveryReceipt(request);
    } else {
      deliverSmForMO(request);
    }
  }
```

If the **DELIVER_SM** request is not for a delivery receipt, the processing is simple. The SMPPPluginSouth's method for processing the request:

1. Gets the SMPP Service object.
2. Gets the SMPP Service object's interface for sending data on a server-side connection.

```
SMPPServiceNorth serviceNorth =
smppService.getProtocolServiceNorth(SMPPServiceNorth.class);
```

3. Sends the request using the SMPPServiceNorth's **deliverSm** method.

```
serviceNorth.deliverSm(request)
```

If the request requires a delivery receipt, the SMPPPluginSouth method for processing the request performs some additional tasks before sending the request:

1. Gets and sets the receipted message ID in the request.

   ```
   String msgId = createPluginMessageId(request.getReceiptedMessageId());
   request.setReceiptedMessageId(msgId);
   ```

2. Using the plug-in's implementation of the SMPPPluginMBean, gets the response command status.

   ```
   failureCommandStatus =
   plugin.getManagement().getMySMPPPluginMBean().getDeliverSmRespCommandStatus();
   ```

   You would implement the **getDeliverSmCommandStatus** method in your SMPPPluginMBean class to get the outcome of the **DELIVER_SM** request. The status should indicate whether the application was reached.

3. Uses the SMPPService **isConnectionBasedRouting** method to establish whether connection-based routing is enabled in the SMPP Service and processes the request accordingly.

   If connection-based routing is enabled, the operator can send a delivery receipt to a site other than the one through which the original message was submitted. See the discussion of connection-based routing in *Services Gatekeeper Communication Service Guide* for information about how connection-based routing works in combination with other configuration settings.

4. Queries any additional relevant configuration settings for the plug-in using the custom management methods implemented by the plug-in in the SMPPPluginMBean and processes accordingly. For example, you may want to query whether to delete SMPP session information after the delivery receipt is received.

5. Uses the SMPPService **isSubsequentOperationsAllowed** method to query whether subsequent operations are allowed for the application instance and sets the session information accordingly.

6. Gets an SMPP Service object.

7. Gets the SMPPService object's interface for sending data on a server-side connection.

8. Sends the request using the SMPPServiceNorth's **deliverSm** method.

Processing a DELIVER_SM Response from an Application

A **DELIVER_SM** response from an application can be the response for the receipt of an mobile-originated SMS message or of a delivery receipt.

The SMPPPluginNorth **deliverSmReponse** method gets the original request event associated with the response, determines whether the response if for a delivery receipt, and passes the request as well as the response to the method that will process and send the response.

```
public void deliverSmResponse(DeliverSmResponse response) {
    DeliverSm originalRequest = (DeliverSm)response.getRequestEvent();
    if (originalRequest != null && !originalRequest.isDeliverReceipt()) {
      deliverSmResponseForMO(response, originalRequest);
    } else {
      deliverSmResponseForDeliveryReceipt(response, originalRequest);
```

```
        }
    }
```

The appropriate **deliverSmResponse** method processes any information needed from the response and its associated request.

A method that processes a response for a mobile-originated SMS message may need to construct EDR data before sending the response to the SMSC using the SMPPServiceSouth **deliverSmResponse** method.

7 Configuring and Testing Composed Service-Level Agreements

This chapter describes how to use the SLA Manager in the Platform Test Environment (PTE) to edit a composed service contract in Oracle Communications Services Gatekeeper.

The Platform Test Environment has been enhanced to provide a graphical interface to the SLA editor for creating and managing composed SLAs. When a user modifies a composed SLA using this editor the corresponding XML file is updated immediately.

The editor provides browsers that provide access to information needed to create and manage SLAs, such as the available identifiers, communication service names, and methods.

For information about composed service contracts, see the discussion of the structure of a composed service contract in *Accounts and SLAs Guide*.

For information about starting and using the SLA Manager in the PTE, see the discussion of configuring and testing service-level agreements in *Platform Test Environment Guide*.

Starting the Composed SLA Editor

To start the Composed SLA Editor:

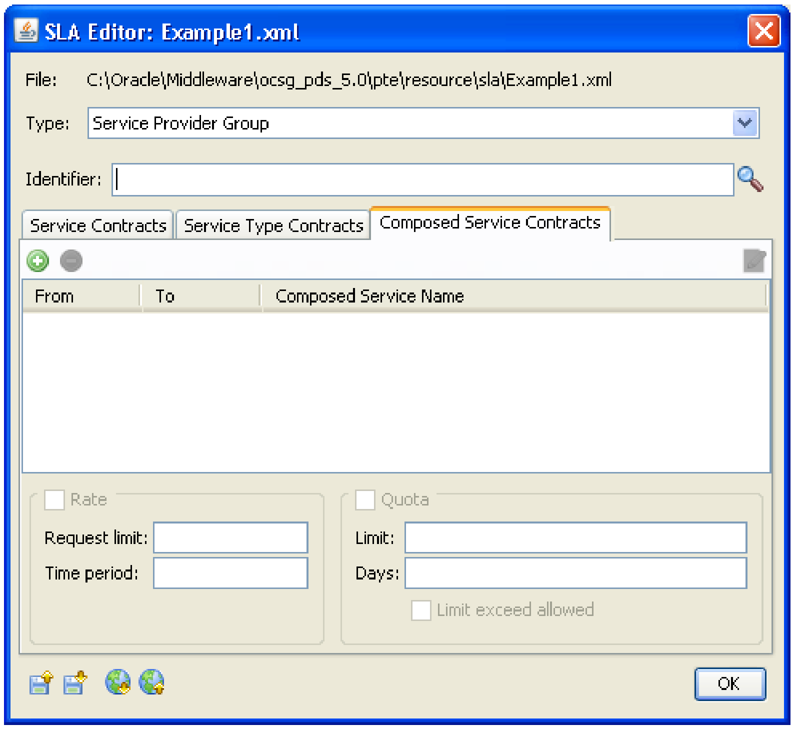
1. In the PTE, connect to Services Gatekeeper and start the SLA Manager.

   For instructions on how to do this, see the discussion of configuring and testing service-level agreements in *Platform Test Environment Guide*.

2. In the SLA Manager, select the **Composed Service Contracts** tab.

   The Composed SLA Editor appears.

Figure 7–1 Composed Service Editor



SLA Editor: Example1.xml

File: C:\Oracle\Middleware\ocsg_pds_5.0\pte\resource\sla\Example1.xml

Type: Service Provider Group

Identifier:

Downloading an Existing Composed SLA

If you have an existing Composed SLA that you want to edit, you can download it from the Services Gatekeeper server.

To download an existing SLA:

1. With the **Composed Service Contract** tab selected, select the type of SLA from the **Type** menu.
2. Enter the identifier for the SLA in the **Identifier** field. You can click the search icon to browse for the identifier:
3. Select the SLA to download.
4. Click download icon at the bottom of the editor to download the SLA:

   The **Selector** box appears.
5. Select the contract that you want to download from the **Selector** box.
6. Click **OK**.

Creating a New Composed SLA

To create a new Composed SLA:

1. With the **Composed Service Contract** tab selected, select the type of SLA that you want to create from the **Type** menu.
2. Assign a unique identifier for the composed SLA in the **Identifier** field.
3. Click the add icon to create a new SLA:

   ⊕

   The new SLA appears in the editor.
4. Add the start and end dates and the composed service name.

   See Composed Service Name and Start/End Dates for instructions on how to assign a name and dates to the composed service contract.
5. Click **OK**.

Editing an SLA

You can edit the following values in a new or previously existing contract:

- Composed Service Name and Start/End Dates
- Rates and Quotas
- Communication Services and Methods

For details about what these values represent, see the discussion of defining service provider group and application group SLAs in *Accounts and SLAs Guide*. The following sections describe only the mechanics of modifying these values in the editor.

Composed Service Name and Start/End Dates

To change the **Composed Service Name**, start date (**From** field), or end date (**To** field):

1. In the composed service editor with the **Composed Service Contract** tab selected, click the field that you want to modify.
2. Over-write the existing value with the new value.

   If you are editing the **Composed Service Name** field, make sure to enter a unique name for the composed service.
3. Click **OK**.

Rates and Quotas

To edit the rate or quota at the bottom of the editor, with the SLA selected:

1. Check the check box for the rate or quota to be enforced by the SLA.
2. Enter the values for the rate or quota to be applied.
3. Click **OK**.

Communication Services and Methods

This section describes how to add communication services to the composed service and how to specify the methods that the composed service contract applies.

The methods portion is optional. If no methods are specified, the enforcements apply to all of the communication services's methods. If at least one method is specified, only the specified method or methods participate in the composed service.

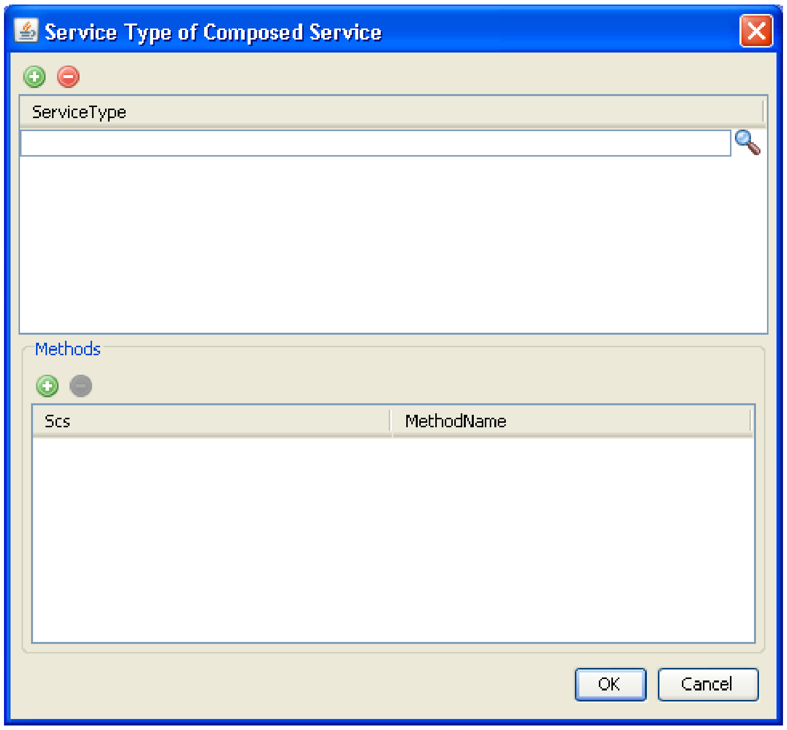
Adding Communication Services

To add the communication services that define the composed service contract:

1. Select the Composed SLA in the editor.
2. Click the edit icon:

   The **Service Type of Composed Service** box appears. Use the upper pane to add communication services.

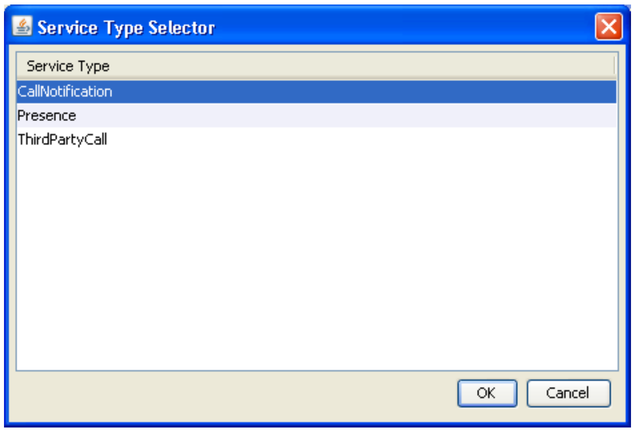
Figure 7–2 Service Type of Composed Service Box



3. For every communication service that you want to add to the composed service, do the following:
   a. In the **Service Type of Composed Service** pane, click the add button to add a service.
   b. Click the search button to load the list of available communication services.

      The **Service Type Selector** list appears.

Figure 7–3 Service Type Selector



c. In the **Service Type Selector** box, select a service type.

d. Click **OK**.

Adding Methods

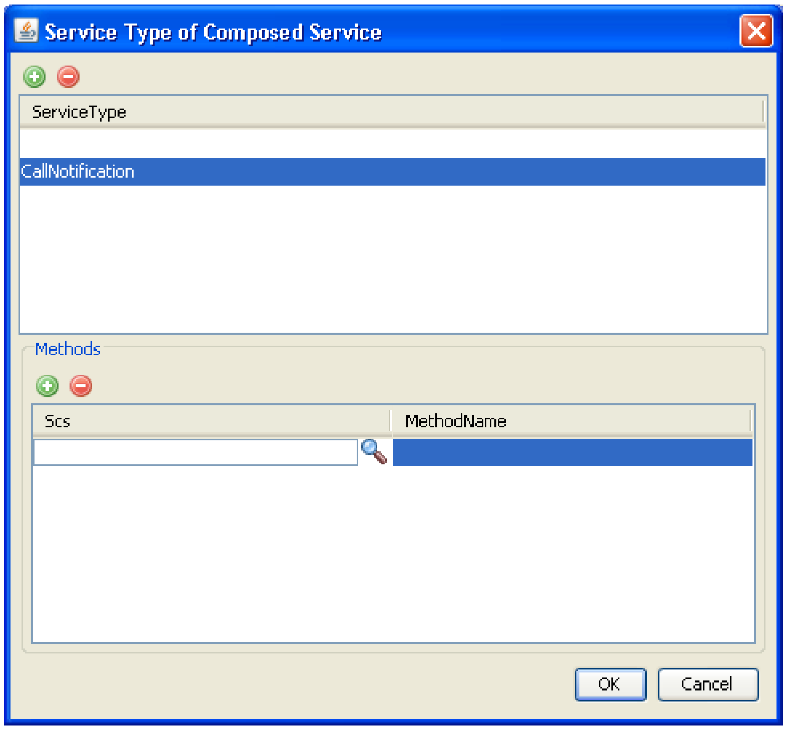
If you want to limit SLA enforcement to specific methods of a communication service, use the lower pane of the **Service Type of Composed Service** box.

To specify a method to which the composed service contract enforcements are applied:

1. With the communication service selected in the upper pane of the **Service Type of Composed Service** box, click the add button in the lower pane.

   A row for the method appears in the lower **Scs** pane.

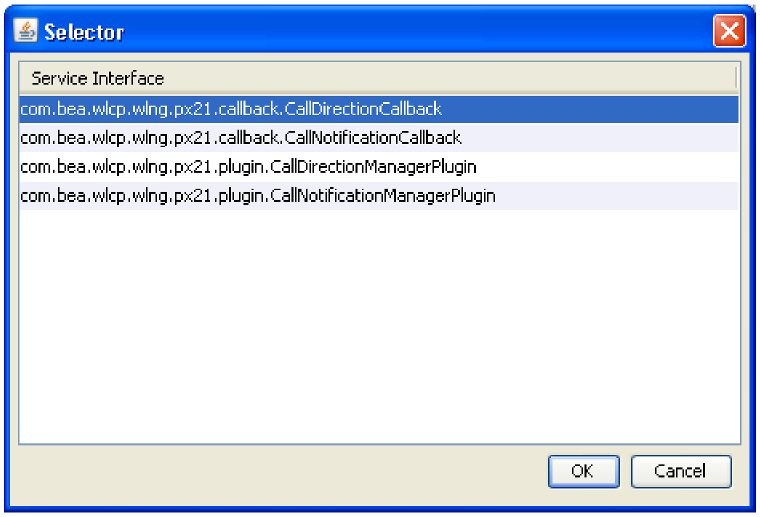
Figure 7–4 Service Type of Composed Service Box with SCS Search Button



2. Click the search button.

   The Scs **Selector** box appears.

Figure 7–5 Scs Selector



com.bea.wlcp.wlng.px21.callback.CallDirectionCallback
com.bea.wlcp.wlng.px21.callback.CallNotificationCallback
com.bea.wlcp.wlng.px21.plugin.CallDirectionManagerPlugin
com.bea.wlcp.wlng.px21.plugin.CallNotificationManagerPlugin

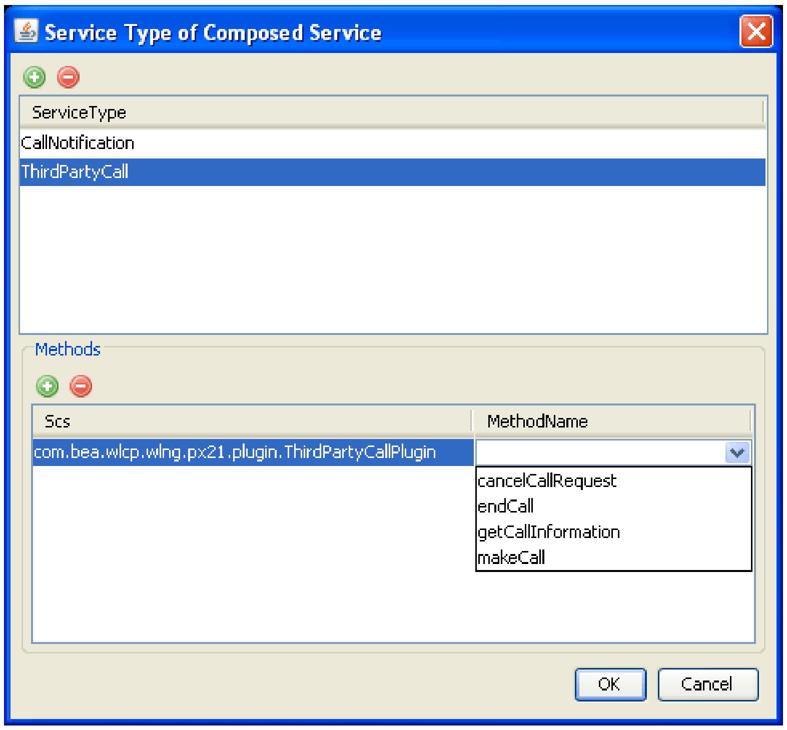
3. Select the service interface.
4. Click **OK**.

   The service interface appears in the lower pane of the Service Type of Composed Service box.

5. Click the Method Name area next to the selected Scs to display the available methods for the Scs.

   A menu of the available methods appears.

Figure 7–6 Service Type of Composed Service Box with Method Menu



6. Select a method name from the menu.
7. Click **OK**

Repeat this procedure for every method that you want to add to the SLA enforcement.

Upload an Edited Composed SLA

To upload the edited SLA:

1. Select the SLA in the editor and click the upload icon:

   An alert asks whether the composed SLA is geo-redundant or local.

2. Respond to the alert by clicking the **Geo-redundant** or **Local** button.

   A local SLA is enforced locally in the network tier cluster into which it is loaded. A geo-redundant SLA is loaded to and enforced across all network tier clusters in a geo-redundant configuration.

   The composed SLA is uploaded.

8 Interceptor Chain Customization

This chapter describes how interceptor chains can be customized for a specific communication service. Interceptor rules can be used to define which interceptors are used based on communication service.

For general information on service interceptors, see the Service Interceptors chapter in the *Platform Development Studio Developer's Guide*.

Overview

The Services Gatekeeper Plugin Manager retrieves a list of all eligible interceptors when an initial service request is received. The Plugin Manager references an internal interceptor rule configuration and removes disabled interceptors for subsequent requests. The customized interceptor chain is then stored in cache so future requests to the same communication service are handed off automatically to the custom interceptor chain.

The interceptor rule configuration is stored in the Services Gatekeeper database and loaded into cache at initial startup. Changes to the configuration will result in a flush of the cached interceptor chains. Subsequent requests then trigger Services Gatekeeper to refresh the cached rule configuration from the database.

Available interceptors in Services Gatekeeper are determined by the **config.xml** file located in the **interceptors.ear** file. For information on configuring available interceptors, including custom interceptors, see the *Platform Development Studio Developer's Guide*.

Managing Custom Interceptor Filter Rules

To create a custom interceptor rule, create an XML file based on the **interceptorRule.xsd** file located in the **$MIDDLEWARE_HOME/ocsg_5.X/modules/com.bea.wlcp.wlng.plugin.mngr_5.0.0.1.jar**. The Plugin Manager MBean is used to create, edit and delete interceptor rule configuration.

The MBean can be accessed from a variety of interfaces including the WebLogic Administration Console, the Platform Test Environment (PTE) or by using the WebLogic Scripting Tool (WLST).

For information on using the WebLogic Administration Console and WSLT, see the Operation and Maintenance chapter in *System Administrator's Guide*.

For information on using the PTE with the Plugin Manager Mbean, see the discussion on Configuring Communication Services by Changing MBean Attributes and Operations in *Platform Test Environment Guide*.

Interceptor Rule Parameters

The default interceptor configuration is provided in the **interceptorRule.xml** file located in the **com.bea.wlcp.wlng.plugin.mngr_5.0.0.1.jar**. When a new Services Gatekeeper domain is created this configuration is used. Interceptors not included in the configuration file are enabled by default.

Use the Mbean operations available in the Administrator Console to edit the configuration. See "Summary of Tasks Related to Interceptors" for more information.

Interceptor rules contain the elements listed in Table 8–1.

Table 8–1 Interceptor Rule Elements

Name	Type	Description
packageName	string	The plug-in for the communication service for which the rule is to be valid for. Regular expressions can be used in the package name to specify more than one package.
methodName	string	The method for which the rule is to be valid for. Regular expressions can be used in the method name to specify more than one method.
interceptorPoint	tns:InterceptorPoint	The topological system location in Services Gatekeeper where the interceptor chain is applied (**MT_NORTH**, **MT_SOUTH**, **MO_NORTH** or **MO_SOUTH**.
interceptorNam e	string	The interceptor for which the rule applies. Multiple interceptors can be included if each is enclosed in the **<interceptorName>** tag.
enable	boolean	Boolean indicating if the interceptor(s) listed should be **enabled** or **disabled** in the rule.

Example 8–1 contains an interceptor rule configuration file that performs the following:

- Applies the rule configuration to all parlayrest plug-ins by using a wildcard:
  - **packageName** is set to **..*$**
- Enables the standard Services Gatekeeper interceptors for the MT_NORTH interceptor for all parlayrest plug-ins:
  - **interceptorPoint** is set to **MT_NORTH**
  - **interceptorName** lists the standard interceptors included in the rule
  - **enable** is set to **true** allowing all the listed interceptors to run
- Disables the MT_NORTH OAuth 2.0 interceptor using a second rule

Example 8–1 Sample interceptorRule.xml Configuration File

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:interceptorConfig xmlns:tns="http://ocsg.oracle/plugin/xsd/interceptorRule"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ocsg.oracle/plugin/xsd/interceptorRule
interceptorRule.xsd ">
  <!-- following are retrieved from ServiceType.java
oracle.ocsg.parlayrest.plugin.MmsPlugin
oracle.ocsg.parlayrest.callback.MessageNotificationCallback
oracle.ocsg.parlayrest.plugin.PaymentPlugin
oracle.ocsg.parlayrest.plugin.ParlayRestSmsPlugin
oracle.ocsg.parlayrest.callback.ClientSmsNotificationCallback
oracle.ocsg.parlayrest.plugin.TerminalLocationPlugin
  -->
  <tns:interceptorRule>
    <tns:packageName>^oracle\.ocsg\.parlayrest\.plugin\..*$</tns:packageName>
    <tns:methodName>^.*$</tns:methodName>
    <tns:interceptorPoint>MT_NORTH</tns:interceptorPoint>

<tns:interceptorName>com.bea.wlcp.wlng.interceptor.EnforceApplicationState</tns:in
terceptorName>

<tns:interceptorName>com.bea.wlcp.wlng.interceptor.EnforceSpAppBudget</tns:interce
ptorName>

<tns:interceptorName>com.bea.wlcp.wlng.interceptor.EnforceComposedBudget</tns:inte
rceptorName>

<tns:interceptorName>com.bea.wlcp.wlng.interceptor.FindAndValidateSLAContract</tns
:interceptorName>

<tns:interceptorName>com.bea.wlcp.wlng.interceptor.CheckMethodParametersFromSLA</t
ns:interceptorName>

<tns:interceptorName>com.bea.wlcp.wlng.interceptor.EnforceBlacklistedMethodFromSLA
</tns:interceptorName>

<tns:interceptorName>com.bea.wlcp.wlng.interceptor.InjectValuesInRequestContextFro
mSLA</tns:interceptorName>

<tns:interceptorName>com.bea.wlcp.wlng.interceptor.EnforceNodeBudget</tns:intercep
torName>

<tns:interceptorName>com.bea.wlcp.wlng.interceptor.EnforceSubscriberBudget</tns:in
terceptorName>
    <tns:enable>true</tns:enable>
  </tns:interceptorRule>

  <tns:interceptorRule>

  <!-- Enable/disable OAuth2 interceptor -->
  <tns:interceptorRule>
    <tns:packageName>^.*$</tns:packageName>
    <tns:methodName>^.*$</tns:methodName>
    <tns:interceptorPoint>MT_NORTH</tns:interceptorPoint>

<tns:interceptorName>oracle.ocsg.oauth2.interceptor.OAuth2Interceptor</tns:interce
ptorName>
    <tns:enable>false</tns:enable>
  </tns:interceptorRule>
</tns:interceptorConfig>
```

Summary of Tasks Related to Interceptors

The following is a summary of tasks related to Interceptor Rules.

Interceptor Rules

Table 8–2 lists the tasks related to application accounts and the operations you use to perform those tasks.

Table 8–2 Tasks Related to Application Accounts

Task	Operation to Use
List the enabled interceptors loaded in Services Gatekeeper	listInterceptors
Retrieve the current interceptor rule configuration file	retrieveInterceptorConfiguration
Update the interceptor rule configuration file	updateInterceptorConfiguration

Reference: Attributes and Operations for Interceptor Rules

Managed object: Container Services > PluginManager

MBean: com.bea.wlcp.wlng.plugin.PluginManagerMBean

Following is a list of operations for configuration and maintenance.

- listInterceptors
- retrieveInterceptorConfiguration
- updateInterceptorConfiguration

listInterceptors

The **listInterceptors** operation retrieves a list of all enabled interceptors in Services Gatekeeper.

Scope

Domain

Signature

```
listInterceptors(InterceptionPoint: String)
```

Parameters

InterceptionPoint
The Services Gatekeeper interface where the interceptor(s) are applied (MT_NORTH, MT_SOUTH, MO_NORTH or MO_SOUTH).

retrieveInterceptorConfiguration

The **retrieveIneterceptorConfiguraiton** operation retrieves the active interceptor rule configuration in Services Gatekeeper.

Scope

Domain

Signature

```
retrieveInterceptorConfiguration()
```

updateInterceptorConfiguration

The **updateInterceptorConfiguration** operation updates the Services Gatekeeper interceptor rule configuration.

Scope

Domain

Signature

```
updateInterceptorConfiguration(Sla : String)
```

Parameters

Sla
The contents on an **interceptorRule.xml** file. See Example 8–1

9 Wizard for Interceptor Creation

This chapter describes how to use the Platform Development Studio (PDS) Eclipse wizard to create custom service interceptors.

For general information on service interceptors and on using the PDS Eclipse wizards, see *Platform Development Studio Developer's Guide*.

Artifacts for a Custom Interceptor Module

When you use the wizard to create the skeleton of an interceptor module, the wizard generates the following artifacts:

- **build.xml**:

  This is the build file used to build all the interceptor modules and to package them into a single EAR file for deployment.

- **build.properties**

  This is the properties file required by the ant process to build the module.

- **common.xml**

  This file defines the common properties used by the module, such as environment variables, WebLogic library path, and so on.

- **CustomizedApplicationLifecycleListener.java**

  This is an implementation of the WebLogic ApplicationLifecycleListener used to manage the module life cycle.

- **InterceptorXXX.java**

  This is your interceptor implementation, where *XXX* is the name that you assign to the interceptor in the wizard.

Generating a Custom Interceptor Module

Make sure that Eclipse is configured to create Services Gatekeeper modules. For more information, see the discussion on configuring Eclipse in *Platform Development Studio Developer's Guide*.

To generate a custom interceptor module:

1. In Eclipse, choose **New Project** from the **File** menu.

   The **Select a wizard** window appears.

2. Select the **Interceptor Module** item under **OCSG Platform Development Studio**.

3. Click **Next**.

   The **Generate Interceptor modules** window appears.

4. In the **Project name** field, enter the name of the Eclipse project for the interceptor module.

5. Either enter the location of the project in the **Location** field or accept the default location by checking the **Use default location** box.

6. In the **Package Name** field, enter the package name to be used for the interceptor package.

7. In the **Application Lifestyle Listener** field enter the ApplicationLifecycleListener implementation class name. If this value is not specified in the wizard, the default value "CustomizedApplicationLifecycleListener" is used.

   For information about the Application Lifestyle Listener, see the discussion of creating a custom listener in Platform Development Studio Developer's Guide and the description of the ApplicationLifecycleListener class in the *WebLogic Server 10.3 API Reference*.

8. Click the **Add** button to add an interceptor module.

   The **Add Interceptor** dialog box appears. In the **Add Interceptor** dialog box, do the following:

   a. In the **Name** field, enter the fully qualified name of the custom interceptor.

   b. In the **Point** menu, choose the position element that describes the type of requests for which the interceptor is valid. Choices are **MO North**, **MT North**, **MO South**, and **MT South**.

   c. In the **Index** field, enter the index in the interceptor stack at which this interceptor will be invoked relative to other interceptors within the same position element. The order is ascending. The index value must be unique within a position element.

      See Chapter 8, "Interceptor Chain Customization," for information about interceptor configuration in Services Gatekeeper, including listing interceptors, to determine the correct Index value.

   d. Click **OK**.

      The interceptor module appears in the list of modules.

   Repeat this step for every interceptor module that you want to add to the EAR file.

9. Click **Finish** to generate the custom interceptor artifacts.

Deploying Custom Interceptors

For information on packaging and deploying custom interceptors see the chapter on Service Interceptors in *Platform Development Studio Developer's Guide*.

10 Wholesale Applications Community Enhancements

This chapter describes how to use Oracle Communications Services Gatekeeper as a Wholesale Applications Community (WAC) Operator Endpoint server in the Payment API flow.

Services providers can expose Diameter-based network charging capability using Service Gatekeeper's OneAPI V2.1 Payment Server functionality. Policy configuration allows the enforcement of request rate and quotas assuring that charging systems are protected from denial of service (DoS) attacks.

About WAC

The WAC is an industry alliance of communications service providers, device manufacturers and software vendors formed to standardize the development and distribution of mobile applications.

For information about WAC, see:

`http://www.wacapps.net/`

Oracle is a member of the WAC.

Supported WAC Functions

Services Gatekeeper supports the following functions used in the WAC baseline architecture for the Payment API Flow:

- OneAPI V2.1 Payment Services

For information about the WAC Payment API Flow, see:

`https://members.wholesaleappcommunity.com/redmine/projects/napigw0-2/wiki/Network_APIs_Platform_Beta_02_Technical_Specification`

OneAPI V2.1 Payment Services

Services Gatekeeper supports the OneAPI RESTful Payment API V2.1 allowing service providers to accept WAC payment requests as part of the Payment API Flow. Services Gatekeeper handles WAC OneAPI payment transactions by forwarding requests to a Diameter plug-in for processing in the network tier.

The WAC Payment API flow currently supports the following transactions:

- Charge a user for a one-time payment

- Check status of previous transaction
- Display a list of all transactions made with an application

The Services Gatekeeper implementation is based on the specifications provided by the Global System for Mobile Communications (GSM).

For information on using the OneAPI V2.1 Payment interface with Services Gatekeeper, see *OneAPI Application Developer's Guide*.

11 Services Gatekeeper OAuth 2.0 Authorization and Resource Servers

This chapter has been deprecated in favor of the stand-alone *Oracle Communciations Services Gatekeeper OAuth Guide*.

12 Data Coding

This chapter provides additional information about how the Parlay X 2.1 Short Messaging/SMPP communication service handles data coding. It does not describe any new or updated features.

Character Set Encoding

The SMPP protocol expects the sender name value in ASCII characters. The use of non-ASCII characters can cause the request to become garbled or even to be removed at the SMSC.

The maximum size of an SMS message is 140 bytes, regardless of the type of data coding used. If the content exceeds 140 bytes, Service Gatekeeper sends it as multiple SMS messages.

Standard and Extended GSM Alphabets

The standard GSM 03.38 alphabet uses 7 bits per character, allowing for 128 different characters with hexadecimal values 0x00 to 0x7F.

If all the characters in an SMS message are from the standard GSM alphabet, it is possible to send 160 of these 7-bit encoded characters in one SMS message of 140 bytes. This is because 140 bytes equals 1120 bits and if each character uses 7-bits, 160 (1120/7) characters fit into the message.

There is also an extended GSM alphabet that defines an additional 10 characters along with the original 128. These characters are sent as two 7-bit encoded characters, starting with the 7-bit encoded escape character (0x1B) from the standard alphabet. For example, if a message contains the character **{** from the extended alphabet, this character is encoded as 1B 28 where 1B is the escape character and 28 is the **{** extended character.

Each extended character requires two 7-bit encoded characters (escape character + extended character). Therefore, an SMS message containing a combination of characters from the standard GSM alphabet and characters from the extended GSM alphabet will hold fewer than 160 characters. The exact number depends on the particular mix of standard and extended characters.

For a list of the characters defined in the GSM standard and extended alphabets see:

`http://www.csoft.co.uk/sms/character_sets/gsm.htm`

To indicate that only SMS messages in which all the characters are from the standard or extended GSM alphabet, the **DefaultDataCoding** attribute should be set to **0**. This is the default. setting. If the **DefaultDataCoding** attribute is set to **0** and the SMS message contains characters that are not in the standard or extended GSM alphabets, Services Gatekeeper rejects the message and throws an exception.

Other Alphabets

It is possible to send characters that are not in the standard or extended GSM alphabets if the **DefaultDataCoding** attribute is configured appropriately.

In addition to the standard and extended GSM alphabets (called the "SMSC Default Alphabet" in the SMPP v3.4 specification), two other common character sets are the IA5/ASCII character set and the UCS2 character set.

In the IA5/ASCII alphabet, the characters are 8-bit encoded, in other words one byte per character, so it is possible to send 140 of these 8-bit encoded characters in one SMS message that uses this coding scheme. If you are using the IA5/ASCII alphabet, set the **DefaultDataCoding** attribute for the plug-in to **1**.

Characters in the UCS2 alphabet are 16-bit encoded, requiring two bytes per character, so it is possible to send only 70 of these characters in a single SMS message. If you are using the UCS2 alphabet, set the **DefaultDataCoding** attribute for the plug-in to **8**.

For a complete list of supported character set values, see the "data_coding" section in the SMPP v3.4 specification.

Overriding the DefaultDataCoding Attribute

You can override the **DefaultDataCoding** attribute in requests using an xparameter or an SLA setting. This makes it possible, for example, to use the standard 7-bit GMS alphabet as the default but to send specific SMS messages using a different character set.

Use the **data_coding** xparameter for parameter tunneling in the header of the request or the **com.bea.wlcp.wlng.plugin.sms.DataCoding** parameter for defining the coding scheme in the `<requestContext>` element of an SLA.

For example, although the **DefaultDataCoding** parameter may be set to **0** for a plug-in instance, the following SOAP header sets the data coding scheme for its SMS message to **8**, stipulating that the UCS2 character set should be used for encoding the SMS message in this particular request:

```
<soapenv: Header>
. . .
    <xparams>
        <param key="data_coding" value="8" />
    <xparams>
. . .
</soapenv:Header>
```

In the next example, the `<requestContext>` element in an SLA sets the data coding scheme to **1**, stipulating that the IA5/ASCII character set should be used for encoding SMS messages initiated by the application associated with this particular SLA:

```
<requestContext>
  <contextAttribute>
    <attributeName>ccom.bea.wlcp.wlng.plugin.sms.DataCoding</attributeName>
    <attributeValue>1</attributeValue>
  </contextAttribute>
</requestContext>
```