BEA Extension SDK for BEA WebLogic Network Gatekeeper™

Introduction and Developer Guide

Version 2.2
Revised: July 31, 2006
## Contents

1. **Introduction to the Extension SDK**
   - Document Scope and Audience ........................................ 1-1
   - Prerequisites ................................................................... 1-2
   - Guide to this Document .................................................. 1-2
   - Terminology ..................................................................... 1-3
   - WebLogic Network Gatekeeper Documentation ...................... 1-7

2. **The Traffic Path Map**
   - Overview .......................................................................... 2-1
   - High Level Traffic Flow .................................................... 2-4
   - Extending WebLogic Network Gatekeeper ............................. 2-5

3. **Traffic Path Flow**
   - Asynchronous application-initiated ................................. 3-2
   - Synchronous application-initiated ..................................... 3-6
   - Network-triggered .............................................................. 3-10
   - Registering the listener ..................................................... 3-12
   - Handling incoming events ................................................ 3-13

4. **The SLEE and Interacting with SLEE Services and Utilities**
   - SLEE Basics ..................................................................... 4-2
   - SLEE States ....................................................................... 4-2
   - SLEE service states .......................................................... 4-3
Interacting with SLEEs ................................................................. 4-4
Implementing SLEE Service Behavior ........................................ 4-5
  ServiceDeployable ................................................................. 4-6
  ServiceDeployableExt ............................................................ 4-6
  ServiceAccessible ................................................................. 4-6
  ServiceManageable ............................................................... 4-7
The ServiceContext object and accessing SLEE utilities ............... 4-7
  Overview ................................................................................. 4-7
  The SLEE Context ................................................................. 4-8
  SLEE utilities accessed through the ServiceContext and SLEEContext interfaces. 4-9
    Services fetched from the Service Context ................................ 4-10
    Services fetched from the SLEE Context .................................... 4-11
Operation, Administration, and Management (OAM) .................... 4-19
  Implementing OAM access control ............................................ 4-20
Using the database ................................................................. 4-20
Using the Configuration Store Service ....................................... 4-27
Using the Global Store Service .................................................. 4-30
  Example .................................................................................. 4-31
Using the Alarm Service ......................................................... 4-34
Using the PolicyManager ........................................................... 4-36
Using the Event Log Service ....................................................... 4-40
Using the Event Data Record Service ......................................... 4-43
  Creating an EDR ..................................................................... 4-44
  Receiving an EDR from the EDR service ..................................... 4-46
Using the Charging Service ....................................................... 4-47
Using Parameter Tunneling .......................................................... 4-51
Using the Time Service ............................................................. 4-53
Using the Trace Service .............................................................. 4-59
5. The Traffic Path Framework

Web Services layer to SESPA .................................................. 5-2
  Registering the class ...................................................... 5-2
  Using the Common Loader to access a SESPA object ............... 5-4
  Using the SESPA object .................................................. 5-5
  Removing an object from the SLEE Common Loader ................ 5-6

SESPA to ESPA ................................................................. 5-7

ESPA to Plug-in Manager .................................................. 5-9
  Accessing the Plug-in Manager ......................................... 5-9
  Plug-in Manager interfaces ............................................. 5-10
    SLEEResourceManager ............................................... 5-11
    Resource ............................................................. 5-12
    ResourceListener ..................................................... 5-13
  Registration and De-registration .................................... 5-13
  Typical Traffic Path for Application-Initiated Requests ........ 5-14

Plug-in Manager to Network Plug-in ................................... 5-16
  Service Capability Interaction ...................................... 5-16
  Plug-in States ........................................................ 5-19
    Suspending a plug-in ................................................ 5-20
    Using plug-in states for high availability ...................... 5-20
  Best practices for the plug-in interfaces ......................... 5-20
    The session id parameter ......................................... 5-20
    Threading .......................................................... 5-21
  Helper classes for network plug-ins ............................... 5-21

Network Plug-in to Service Capability Manager ....................... 5-23
Accessing the SC Manager ......................................................... 5-23
Interfaces ................................................................. 5-24
SLEESCSManager ........................................................... 5-26
SLEESCSRegistrationOperations ................................. 5-26
SLEESCSDiscoveryOperations ................................. 5-26
Network-triggered events and the SC Manager ................. 5-27

6. High availability
   Introduction ................................................................. 6-1
   High Availability Between Network Plug-ins and ESPA SCs .... 6-2
   Application-Initiated ..................................................... 6-2
   Network-Triggered ........................................................ 6-2
   Plug-ins using the Service Capability Manager ................. 6-3
   Failure on notification reporting .................................. 6-3
   Plug-ins not using the Service Capability Manager .......... 6-4
   High Availability between SESPA and ESPA .................. 6-4

7. A Closer Look at One Layer: the Network Plug-in

8. Call Control
   Call Control Interfaces ...................................................... 8-1
   Implemented by the Call Control SC ................................. 8-3
     IrCallControlPlugInListener ........................................ 8-3
     IrAppCall ............................................................. 8-3
     IrAppCallLeg .......................................................... 8-4
   Implemented by the Network Plug-in .............................. 8-5
     CallControlResource .................................................. 8-5
     IrCallControlManager - Deprecated - .......................... 8-5
     IrCallControlManagerExt ............................................. 8-6
9. Call User Interaction

Call User Interaction interfaces ................................................. 9-1

Implemented by the Call User Interaction SC ............................. 9-2

IrAppUI ................................................................. 9-3
IrAppUI.Call .......................................................... 9-3

Implemented by the Network Plug-in ....................................... 9-4

UserInteraction.CallResource ................................................. 9-4
IrUI ................................................................. 9-5
IrUI.Call .......................................................... 9-5
IrUICallManager ....................................................... 9-5
IrUICallManagerExt .................................................. 9-6

Use case for Call User Interaction ......................................... 9-6

Application-initiated Call User Interaction Session .................... 9-6

10. SMS and MMS Messaging

Messaging Interfaces .......................................................... 10-1

SMS interfaces ............................................................ 10-2

Implemented by the Messaging Service Capability ....................... 10-3

MessageListener ......................................................... 10-3
MessageListenerExt ..................................................... 10-4
AsyncSmsResourceListener ............................................. 10-4
AsyncSmsResourceNotification ........................................ 10-5
Implemented by the SMS Network Plug-in ........................................ 10-5
  MessagingResource ......................................................... 10-6
  MessagingResourceExt ................................................. 10-6
  AsyncSmsResource .......................................................... 10-7
  AsyncSmsResourceNotificationListener .................................. 10-7

MMS interfaces ........................................................................ 10-8
  Implemented by the Messaging Service Capability ...................... 10-9
  MmsListener ................................................................. 10-9
  MmsListenerExt ............................................................. 10-9
  MmsResourceExt2 ............................................................ 10-10
  AsyncMmsResourceNotification .......................................... 10-10
  AsyncMmsResourceListener .............................................. 10-11

  Implemented by the MMS Network Plug-in ............................... 10-11
  MmsResource ............................................................. 10-12
  MmsResourceExt ........................................................... 10-12
  MmsResourceExt2 ............................................................ 10-13
  AsyncMmsResource ......................................................... 10-13
  AsyncMmsResourceNotificationListener .............................. 10-14

Use cases for SMS .................................................................. 10-14
  Registering the Service Capability ...................................... 10-14
  Application-initiated send message .................................... 10-15
  Network-initiated messages ............................................. 10-17

11. Message Sender
  Message Sender Interfaces ..................................................... 11-1
  Implemented by the Message Sender service capability ............ 11-2
    IrStatusNotificationReqListener ........................................ 11-2
    IrMessageResListener .................................................. 11-3
12. Content Based Charging

Content Based Charging Interfaces ....................................................... 12-1

Implemented by the Content Based Charging SC .......................... 12-2

IrAppChargingSession ................................................................. 12-3

Implemented by the Content Based Charging plug-in .................. 12-4

ChargingResource .................................................................. 12-5

IrChargingSession ................................................................. 12-5

Use case ................................................................. 12-6

13. Subscriber Profile

Subscriber Profile Interfaces .......................................................... 13-1

Implemented by the Subscriber Profile SC .............................. 13-2

IrAppSubscriberProfile ......................................................... 13-3

Implemented by the Subscriber Profile plug-in ...................... 13-3

IrSubscriberProfileResource ................................................. 13-4

IrSubscriberProfile ............................................................. 13-4

IrSubscriberProfileSubscriptionExt ..................................... 13-4

Implemented by a customized non-SC client of the Subscriber Profile plug-in . . . 13-5

IrAppSubscriberProfileSubscriptionExt ................................ 13-5

Use case ................................................................. 13-5

14. User Location

User Location Interfaces ............................................................. 14-1

Implemented by the User Location SC .................................... 14-2
14. Use cases

14.1 Application-initiated User Location Request

14.2 Network-triggered user location request

15. Plug-ins that execute as both a SLEE service and a web application

15.1 Overview

15.2 The Stages of Interaction

15.2.1 Interface class registration

15.2.2 Processing incoming requests

15.2.3 SLEE Service part

15.2.4 Web Application part

15.2.5 Processing outgoing requests

15.2.6 Web Application part

15.2.7 SLEE service part

16. Policy rules and Policy Utilities

16.1 PolicyRequest data

16.2 Adding a new rule to Policy Decision Point

16.2.1 Getting data defined in the PolicyRequest

16.3 Extending Service Level Agreements

16.3.1 Update the SLA Schema

16.3.2 Load the new SLA schema into the Policy Service
Update and reload the rule files ........................................... 16-9
Update the appropriate SLAs ........................................... 16-10
Load the updated SLAs ................................................. 16-10
Using a Policy Utility object ........................................... 16-10
Defining a Policy Utility class ....................................... 16-11
An Example of a Policy Utility ....................................... 16-12

17. Hints and Tips

General SLEE Managers ............................................... 17-1
Using the SLEE Task Manager ........................................ 17-2
Using the SLEE ID Managers .......................................... 17-2
Synchronous Interface/Asynchronous Implementation ............ 17-2
Time-outs .................................................................... 17-4
The Tracing and Alarm Services ..................................... 17-5
Tracing ........................................................................ 17-5
Alarms ......................................................................... 17-6
SLEE Service and Web Archive (WS layer) interaction using the SLEE common loader ................................................. 17-6
Version Handling and the SLEE Common Loader .......... 17-8
Web Services and WSDD/WSDL .................................... 17-8
Networking .................................................................. 17-10
Locator Classes ............................................................ 17-10
CORBA Object Activation ............................................. 17-10
CORBA Object Deactivation ......................................... 17-12
Introduction to the Extension SDK

The worlds of Internet applications and telephony networks continue to converge. But the relationship between them is often overly complex and difficult to manage. What is needed is a powerful, flexible, secure interface providing, on the one hand, a simple way for application developers to include telephony-based functionality in their software applications (like text and multimedia messaging) and, on the other, features that guarantee the security, stability, and performance required by network operators and demanded by their subscribers. BEA’s WebLogic Network Gatekeeper is designed to do exactly this.

By design Network Gatekeeper is highly modular and extensible, so that it can be customized to support the wide range of technologies that make up this environment. This document provides an overview of WebLogic Network Gatekeeper which should serve as a background for those developers who wish to extend its functionality in this way.

This chapter introduces the overall structure and scope of the document.

- Document Scope and Audience
- Guide to this Document
- Terminology
- WebLogic Network Gatekeeper Documentation

**Document Scope and Audience**

This document provides a high level overview of WebLogic Gatekeeper from various perspectives, including:
Introduction to the Extension SDK

- The Traffic Path Map and an overview of traffic flow
- The SLEE Map
  - Utilities
  - Management
- The mechanisms for interaction between layers
- Provisions for high availability
- A closer look at one layer, network plug-ins
  - Call Control
  - Call User Interaction
  - Messaging
  - Content Based Charging
  - Subscriber Profile
  - User Location
  - Plug-ins that use HTTP
- SLAs, Policy, and Policy Utilities

The intended audience of this document consists of system integrators and field engineers who need to extend the out-of-the-box functionality of WebLogic Network Gatekeeper.

Prerequisites

WebLogic Network Gatekeeper is implemented in Java and uses CORBA extensively. Substantial knowledge of both of these technologies is required for successful use of the Extension SDK. As well, because the Network Gatekeeper is a fairly complex system, a solid grasp of its architecture, preferably including hand-on experience, is essential. Finally, because most application facing interfaces are Web Services based (either Parlay X or other), familiarity with the relevant standards is of use.

Guide to this Document

The document contains the following chapters:

- Introduction to the Extension SDK: This chapter
Terminology

The following terms and acronyms are used in this document:

- **Accessible**—A SLEE service that can be reached using a CORBA connection
- **Account**—A registered application or service provider, associated with an SLA
Introduction to the Extension SDK

- Account group—Multiple registered service providers or services which share a common SLA
- Administrative User—Someone who has privileges on the Network Gatekeeper management tool. This person has an administrative user name and password
- Alarm—The result of an unexpected event in the system, often requiring corrective action
- API—Application Programming Interface
- Application—A TCP/IP based, telecom-enabled program accessed from either a telephony terminal or a computer
- Application Service Provider—An organization offering application services to end users through a telephony network
- AS—Application Server
- Application User—An Application Service Provider from the perspective of internal Network Gatekeeper administration. An Application User has a user name and password
- ATE—WebLogic Network Gatekeeper’s Application Test Environment
- CBC—Content Based Charging
- CORBA—Common Object Request Broker Architecture. WebLogic Network Gatekeeper uses CORBA for its internal communication
- Deployable—The minimum condition required of services that run in a SLEE
- End User—The ultimate consumer of the services that an application provides. An end user can be the same as the network subscriber, as in the case of a prepaid service or they can be a non-subscriber, as in the case of an automated mail-ordering application where the subscriber is the mail-order company and the end user is a customer to this company
- ESPA—The core layer of the Service Capabilities module
- Event—A trackable, expected occurrence in the system, of interest to the operator
- HA—High Availability
- HTML—Hypertext Markup Language
- IDL—CORBA’s Interface Definition Language.
- IIOP—Internet Inter-ORB Protocol
Terminology

- IN—Intelligent Network
- INAP—Intelligent Network Application Part
- IOR—Interoperable Object Reference
- IP—Internet Protocol
- JDBC—Java Database Connectivity, the Java API for database access
- Location Uncertainty Shape—A geometric shape surrounding a base point specified in terms of latitude and longitude. It is used in terminal location
- Manageable—A deployable SLEE service that can be managed using the Management Tool
- MAP—Mobile Application Part
- Mated Pair—Two physically distributed installations of WebLogic Network Gatekeeper nodes sharing a subset of data allowing for high availability between the nodes
- MM7—A multimedia messaging protocol specified by 3GPP
- MPP—Mobile Positioning Protocol
- Network Plug-in—The WebLogic Network Gatekeeper module that implements the interface to a network node or OSA/Parlay SCS through a specific protocol
- Northbound Interface—The Application Services Provider facing interface
- NS—Network Simulator
- OAM—Operation, Administration, and Maintenance
- Operator—The party that manages the Network Gatekeeper. Usually the network operator
- ORB—The CORBA Object Request Broker
- OSA—Open Service Access
- PAP—Push Access Protocol
- Plug-in—See Network Plug-in
- Plug-in Manager—The Network Gatekeeper module charged with routing an application-initiated request from a service capability module to a network plug-in
Introduction to the Extension SDK

- **Policy Engine**—The Network Gatekeeper module charged with evaluating whether a particular request is acceptable under the rules
- **Quota**—Access rule based on an aggregated number of invocations. See also Rates
- **Rate**—Access rule based on allowable invocations per time period. See also Quotas
- **Rules**—The customizable set of criteria - based on SLAs, subscriber profiles, and operator-desired additions - according to which requests are evaluated
- **SC**—See Service Capability
- **SCF**—Service Capability Function or Service Control Function, in the OSA/Parlay sense.
- **SCS**—Service Capability Server, in the OSA/Parlay sense. WebLogic Network Gatekeeper can interact with these on its southbound interface
- **Service Capability**—Support for a specific kind of traffic within WebLogic Network Gatekeeper. For example, the Messaging Service Capability processes SMS and MMS messages
- **Service Capability Manager**—The Network Gatekeeper module charged with routing a network-triggered event from a network plug-in to an appropriate service capability module
- **Service Provider**—See Application Service Provider
- **SESPA**—The top layer of the Service Capabilities module
- **SIP**—Session Initiation Protocol
- **SLA**—Service Level Agreement
- **SLEE**—Service Logic Execution Environment. The core modules in WebLogic Network Gatekeeper run in SLEE-based environments
- **SLEE Service**—A software module that is designed to execute in the SLEE
- **SMPP**—Short Message Peer-to-Peer Protocol
- **SMS**—Short Message Service
- **SMSC**—Short Message Service Centre
- **SNMP**—Simple Network Management Protocol
- **SOAP**—Simple Object Access Protocol
Southbound Interface—The network-node facing interface. It may also interact with an OSA/Parlay gateway

SPA—Service Provider APIs

SS7—Signalling System 7

Subscriber—A person or organization that signs up for access to an application. The subscriber is charged for the application service usage. See End User

SQL—Structured Query Language

TCP—Transmission Control Protocol

Traffic Path—The data flow of a particular request through WebLogic Network Gatekeeper

USSD—Unstructured Supplementary Service Data

VAS—Value Added Service

VLAN—Virtual Local Area Network

VPN—Virtual Private Network

WSDL—Web Services Definition Language

XML—Extended Markup Language

WebLogic Network Gatekeeper Documentation

The WebLogic Network Gatekeeper documentation set includes:

- Architectural Overview- WebLogic Network Gatekeeper
  A high level description of the functionality and architecture of the WebLogic Network Gatekeeper.

- System Administrator’s Guide- WebLogic Network Gatekeeper
  A detailed guide to administering WebLogic Network Gatekeeper.

- Installation Guide- WebLogic Network Gatekeeper
  A guide to installing WebLogic Network Gatekeeper and the Network Gatekeeper Management Tool.
Introduction to the Extension SDK

- Application Developer's Guide - Parlay X for WebLogic Network Gatekeeper
  The developer's guide describes how to design and implement applications using the Parlay X 1.0 Web Services exposed by WebLogic Network Gatekeeper.

- API Descriptions - Parlay X for WebLogic Network Gatekeeper
  The API descriptions describe WebLogic Network Gatekeeper Parlay X 1.0 APIs available for developers and applications.

- Application Developer’s Guide - Extended Web Services for WebLogic Network Gatekeeper
  A guide to designing and implementing applications using the Extended Web Services API.

- API Description - Extended Web Services for WebLogic Network Gatekeeper
  The Extended Web Services API reference for WebLogic Network Gatekeeper.

- Integration Guidelines for Partner Relationship Management - WebLogic Network Gatekeeper
  A guide to using the WebLogic Network Gatekeeper PRM interfaces.

- Web Service API Description for Partner Relationship Management - WebLogic Network Gatekeeper
  The PRM Web Services API reference for WebLogic Network Gatekeeper.
The Traffic Path Map

The following sections provide an overview of WebLogic Network Gatekeeper in terms of the components that make up the path that data traffic follows:

- Overview
- High Level Traffic Flow
- Extending WebLogic Network Gatekeeper

Overview

One way of understanding the structure of WebLogic Network Gatekeeper is in terms of the traffic path, the set of components through which data flows as it traverses the system. A traffic path consists of a northbound interface (that connects with an external application), a service capability module, and a network plug-in.
Applications are Internet based applications that wish to include text and multimedia messaging, user location and presence, call control, or other telephony based features. An individual Application is part of a larger entity called an Application Service Provider. Policy decisions are based on Service Level Agreements at both the Application and the Service Provider levels. For a more complete description of the Application Service Provider model, see “Managing Application Service Providers” in WebLogic Network Gatekeeper Architectural Overview, a separate document in the Network Gatekeeper set.

Northbound Interfaces offer high level APIs that allow Application Service Providers to access particular telecom network functionality in a simple-to-develop-and-use way. Most, but not all, of these interfaces are Web Services based. Out of the box, Network Gatekeeper supports the following northbound interfaces:

- Parlay X 1.0
- Parlay X 2.1 (partial)
- A Network Gatekeeper developed set called the Extended Web Services
- A number of legacy APIs, including SMPP, MM7, and PAP
Overview

Service Capabilities are abstractions of the kinds of functionality supported by the Network Gatekeeper. It is in the service capabilities layer that most policy decisions are made. Differing northbound interfaces may use the same service capability, and that service capability, in turn, may be linked to different network plug-ins, depending on the structure of the underlying network. Some service capabilities may be supported by a wide variety of northbound interface types, some not. And an interface may support some but not all of the features of a particular service capability. Out of the box, Network Gatekeeper supports:

- Access
- Messaging
- Message Sender
- Charging
- Call Control
- Subscriber Profile (requires customization on the plug-in side to access an external database)
- User Interaction
- User Location
- User Status

Network Plug-ins take the request which has been processed by the service capability and transform it into a form suitable for the targeted underlying network node or OSA gateway. Out of the box, Network Gatekeeper supports the following network protocols (not all protocols are supported by all service capabilities):

- Parlay 3.3 (OSA rel 4)
  - Generic Call Control
  - Multiparty Call Control
  - Call User Interaction
  - Generic User Interaction
  - User Location
  - User Status
  - Parlay 3.3 (OSA rel 4) SMS and MMS
The Traffic Path Map

- CIMD2
- Ericsson MM7 R2.5
- Ericsson MM7 R2.0
- Ericsson MM7 1.0
- MM7 rel 5
- Nokia EAIF
- SMPP version 3.4
- Parlay 5.0 Multimedia Messaging SMS
- Parlay 5.0 Multimedia Messaging MMS
- PAP 2.0
- Parlay 5.0 Multimedia Messaging WAP Push
- MLP (LIF) 3.2.0

High Level Traffic Flow

A typical sequence for an application-initiated request (flowing from an application service provider, through the Network Gatekeeper, to the network) using this map would look like this:

1. The application logs into the appropriate northbound interface with a username, password, and operator-supplied IDs, using a SOAP-based Web Services request. If heightened security is desired, the request may travel over SSL or through a VPN. The structure of the request is documented in a published WSDL file and the request is created using any toolset the developer wishes.

2. The Network Gatekeeper verifies that the maximum number of sessions specified in the Service Level Agreement (SLA) for this application and its service provider has not been exceeded.

3. A login ticket is issued establishing an application session. This ticket is included in all future incoming SOAP requests from the application service provider.
4. A request for a particular operation enters at the northbound interface and is pre-processed. The SOAP envelope is removed and the request is transformed into a system object. Because Web Services based requests are stateless, a stateless adapter layer (known as SESPA) converts the object to a stateful form, which is required by the current internals of the Network Gatekeeper.

5. This stateful request is passed on to the service capability module (known as an ESPA SC), where it is bundled and sent to the policy engine to be evaluated according to the rules. This evaluation determines whether or not the request will be accepted. The rules enforce a set of SLAs (service provider and application specific). They may also include any additional criteria the operator chooses to employ. Data from subscriber profiles can also incorporated in the rules.

6. If the policy engine accepts the request, the service capability module contacts the plug-in manager to find a suitable network plug-in. The plug-in manager evaluates the needs of the request and chooses an appropriate network plug-in.

7. The policy engine is again contacted and evaluates the request in terms of traffic SLAs (including allowed overall traffic from WebLogic Network Gatekeeper to the network).

8. The request is passed on to the network plug-in supplied through the plug-in manager. This plug-in transforms the request into the protocol used by the network node to which the plug-in is connected.

9. The request is safely and efficiently passed to the network.

10. Once the request has been accepted by the network node, all pertinent charging data about the request are recorded.

Note: The path for the northbound interface may differ slightly depending on the type of interface used.

Extending WebLogic Network Gatekeeper

Out of the box, WebLogic Network Gatekeeper provides a robust, secure interface between Internet applications and telecom networks by supporting available standards. But in many of the environments in which Network Gatekeeper is deployed, elements that employ customized or completely nonstandard interfaces are common. The WebLogic Network Gatekeeper Extension SDK is designed to assist developers who are charged with modifying Network Gatekeeper to support these custom or nonstandard elements. In general there are three scenarios in which extending Network Gatekeeper plays a role:
- **New Network Plug-in**: The functionality made available by the northbound interfaces and supported by the service capabilities is sufficient, but the underlying network element relies on a protocol not supported by any of the standard plug-ins. In this case, the creation of a new network plug-in, which supports the new protocol but can be mapped back to the standard service capability interface, is all that is necessary. This is, in fact, the most common scenario for extending the Network Gatekeeper.

- **New Northbound Interface**: The functionality made available by the service capability layer is sufficient, and the packaged network plug-ins support the protocols required by the underlying network elements, but the application service provider requires a non-standard interface to access this functionality. In this case, the implementation of a new northbound interface to allow the service provider access may be all that is required. Such interfaces can then be mapped to the appropriate stateless adapter so that application requests can be handed off to the service capability.

There are some limitations to this option. The internal mechanism used for almost all packaged northbound interfaces assumes a Web Services or at least http based implementation, so this method would not work for a non-WS/http based northbound interface. In addition, all charging records (CDRs) generated by such a traffic path are associated with the pre-existing service capability, and the SLA data and rules that apply to those requests are those associated with that particular service capability. This may not be appropriate in some cases.

- **New Traffic Path**: It’s possible that the particular Network Gatekeeper deployment situation requires the creation of an entirely new traffic path. This is, by definition, the most ambitious of the three scenarios, but the Extension SDK does offer templates to help construct an entirely new path. In such a case, the developers must create the northbound interfaces, service capabilities, and network plug-ins in a way that functions effectively in the SLEE-based execution environment of the Network Gatekeeper. For more information on this environment, see “The SLEE and Interacting with SLEE Services and Utilities” on page 4-1. The new path must also follow the basic architecture of the Network Gatekeeper, including the use of the plug-in manager and its counterpart the service capability manager. For more information on this basic architecture see “Traffic Path Flow” on page 3-1. To better understand the ways the system communicates internally, see “The Traffic Path Framework” on page 5-1. The WebLogic Network Gatekeeper Architectural Overview, a separate document in the Network Gatekeeper set, may also be of use.
Traffic Path Flow

The description in “High Level Traffic Flow” on page 2-4 is a very high level overview of the path of an application request as it travels through WebLogic Network Gatekeeper. But there are, in fact, three basic patterns of traffic flow within the Network Gatekeeper and the following chapter describes these patterns in a slightly more detailed way, so that you can have a better grasp of the types of traffic that Network Gatekeeper typically handles.

The patterns are presented as a set of three generalized sequence flow diagrams following an arbitrary method request through Network Gatekeeper. These sequences are presented as an introduction to typical traffic flow patterns. The specifics of how all this occurs are covered in a more detailed way in “The Traffic Path Framework” on page 5-1. The three types of flow include:

- **Asynchronous application-initiated**
  
  A request that is asynchronous from the point of view of the originating service provider application.

- **Synchronous application-initiated**
  
  A request that is synchronous from the point of view of the originating service provider application.

- **Network-triggered**
  
  A request or notification from the network to the service provider application. The application must have indicated that it is interested in receiving such requests. Examples might include an incoming call that is to be routed through a call control application or a notification that a particular terminal/user has entered a geographic area.
Generally speaking, asynchronous interfaces are preferred, as synchronous interfaces block until the response is available, which ties up a thread for an extended period of time. The number of threads available in Network Gatekeeper at any one time is a finite resource, defined in the thread pool size, so using and releasing a thread in the shortest time possible is best for the system as a whole. If it is possible to choose between an asynchronous or a synchronous interface, the asynchronous should be used.

Asynchronous application-initiated

The first sequence follows an application request that does not expect an immediate result (other than the standard http response) from Network Gatekeeper. The initial http response in this sequence simply indicates only that the request has been scheduled to be evaluated by policy and then sent on to a plug-in if it is accepted. All subsequent results - including a rejection by the policy engine - are returned to the application via a callback that, depending on the interface, is either set up via the Network Gatekeeper’s Management Tool or sent as a part of the initial request. Internal to Network Gatekeeper, the request is also executed in an asynchronous manner, for performance reasons.
Figure 3-1  Asynchronous application-initiated sequence
1. The application sends a SOAP-encoded request (SOAP) to the *Web Services layer* implementation (WS layer), which is deployed in the Embedded Tomcat server.

   **Note:** You may notice the code templates associated with the WS layer belong to the `com.acompany.wespa.mysactype` package. The WS layer was called the WESPA layer in earlier versions of the Extension-SDK.

2. The WS layer removes the SOAP envelope and transforms the request into a normal Java object. It is then passed to the *stateless adapter layer* (SESPA SC) by means of a registration mechanism facilitated by the SLEE Common Loader. Because the internals of the Network Gatekeeper are stateful, traffic that flows through it must also be stateful, even though the initial Web Services requests are, by their nature, stateless. If parameters not supported by standard internal interfaces need to be transported through the traffic path, or tunneled, they should be added here, see “Using Parameter Tunneling” on page 4-51.

   **Note:** All calls across layers from this point are CORBA based.

3. SESPA SC has a listener class which serves as a callback object for returning the response to the request from the *service capability layer* (ESPA SC) to SESPA SC. Only one listener is used and different request/response pairs are matched together using transaction IDs.

4. SESPA SC acquires the application’s session object and an ESPA Manager object using the login ticket that was in the header of the SOAP envelope and then uses the ESPA Manager object to make the appropriate call on the ESPA SC. The call, `xMethodReq`, has, as arguments, SESPA's listener object and a transaction ID. The transaction ID is used to match the request/response pairs.

   **Note:** Timeout timers should be used to monitor the response. If a response arrives successfully, the timer should be cancelled.

5. The ESPA SC creates a new thread using the SLEE Task Manager.

6. In the new thread, the ESPA SC bundles the application’s request into a Policy Request object, which is then sent on to the Policy Service, where an evaluation (`eval`) based on the *rules* is performed. In the ESPA SC, this is called a Policy Enforcement Point (PEP). There are two possible outcomes as a result of this evaluation:

   a. If the application’s request is denied by the Policy Service, an error method is called on the SESPA listener object. A new thread is created using the SLEE Task Manager, and the error method is propagated to the WS layer and finally is returned to the application. The error method contains the information that the request was denied by the Policy Service.

   b. If the application’s request is accepted by the Policy Service, the Policy Service examines the original request parameters and may, in some situations, update them. These
Asynchronous application-initiated

parameters are then returned to the ESPA SC. The updated parameters are what will be used from this point forward.

7. A resource task (a request for both a reference to an appropriate network plug-in and a separate thread on which the call to the plug-in will take place) is scheduled by calling scheduleResourceTask on the Plug-in Manager. A callback object (the Task) is supplied in the request. (The Task object is not shown in the diagram).

8. Because the call to the plug-in takes place on a separate thread, the scheduleResourceTask call returns immediately.

9. When the Plug-in Manager has found a suitable plug-in and a free thread is available from the thread pool, doTask is called on the Task object, which returns a reference for an appropriate plug-in to the ESPA SC.

10. The ESPA SC holds a listener object that is designed to listen for the response from the network to the initial request. The request/response pairs are matched using transaction IDs.

   Note: Timeout timers should be used to monitor the response. If a response arrives successfully, the timer should be cancelled.

11. The request is propagated to the plug-in, with the listener object sent as an argument. If tunneled parameters have been used, they are retrieved here; see “Using Parameter Tunneling” on page 4-51 for more information on parameter tunneling.

12. The plug-in creates a new thread using the SLEE Task Manager and translates the request according to appropriate network protocols. The operation shown in the diagram is asynchronous to the network, so the initial call to the network returns immediately to the plug-in, but the actual result of the operation is received later and returned using the listener object.

13. When the response to the request reaches the plug-in, a new thread is created using the SLEE Task Manager and the plug-in sends the request back to the ESPA SC using the listener.

14. When the ESPA SC has received the response, it creates a new thread using the SLEE Task Manager and creates a CDR using the SLEE Charging service.

15. The response is propagated from the ESPA SC listener to the SESPA SC listener, where a new thread is created using the SLEE Task Manager.

16. The SESPA SC listener propagates the response to the WS layer, which sends it on to the application via a mechanism which is dependent on the interface itself.

   Note: In the 2.2 version of the Extension SDK, SLEE services, including SESPA, have access to the AXIS classes necessary for creating SOAP messages directly, so it is
also possible for SESPA to send the response directly, without using the WS layer as an intermediary. This is now the default practice.

**Synchronous application-initiated**

The second sequence follows an application request that expects a synchronous result from Network Gatekeeper, and will wait until it receives it. Internal to Network Gatekeeper, however, the request is executed in an asynchronous manner, for performance reasons.
Figure 3-2  Synchronous application-initiated sequence
Traffic Path Flow

1. The application sends a SOAP-encoded request (SOAP) to the *Web Services layer* implementation (WS layer), which is deployed in the Embedded Tomcat server.

2. The WS layer removes the SOAP envelope and transforms the request into a normal Java object. It is then passed to the *stateless adapter layer* (SESPA SC) by means of a registration mechanism facilitated by the SLEE Common Loader. Because the internals of the Network Gatekeeper are stateful, traffic that flows through it must also be stateful, even though the initial Web Services requests are, by their nature, stateless.

   **Note:** All calls across layers from this point are CORBA based.

3. SESPA SC has a listener class which serves as a callback object for returning the response to the request from the *service capability layer* (ESPA SC) to SESPA SC. Only one listener is used and different request/response pairs are matched together using transaction IDs.

4. SESPA SC acquires the application’s session object and an ESPA Manager object using the login ticket that was in the header of the SOAP envelope and then uses the Manager object to make the appropriate call on the service capability (ESPA SC). The call, $xMethodReq$, has, as arguments, SESPA’s listener object and a transaction ID. The transaction ID is used to match the request/response pairs.

   **Note:** Timeout timers should be used to monitor the response. If a response arrives successfully, the timer should be cancelled.

5. A new thread is created using the SLEE Task Manager, the request returns and the SESPA SC Listener thread is set into wait state.

6. In the new thread, the ESPA SC bundles the application’s request into a Policy Request object, which is then sent on to the Policy Service, where an evaluation ($eval$) based on the *rules* is performed. In the ESPA SC, this is called a Policy Enforcement Point (PEP). There are two possible outcomes as a result of this evaluation:

   a. If the application’s request is denied by the Policy Service, an error response, $xMethodErr$, including an error code indicating that the request was denied, is sent from the ESPA SC to the SESPA SC. The SESPA listener wakes up from the wait state by calling notify and throws an a PolicyException to the WS layer and finally the Exception is propagated to the application.

   b. If the application’s request is accepted by the Policy Service, the Policy Service examines the original request parameters and may, in some situations, update them. These parameters are then returned to the ESPA SC. The updated parameters are what will be used from this point forward.
7. A resource task (including a request for an appropriate network plug-in) is scheduled by calling `scheduleResourceTask` on the Plug-in Manager. A callback object (the Task) is supplied in the request. (The Task object is not shown in the diagram).

8. Because the call to the plug-in will take place on a separate thread, the `scheduleResourceTask` call returns immediately and the execution thread returns back to the SESPA SC.

9. When the Plug-in Manager has found a suitable plug-in and a free thread is available from the thread pool, `doTask` is called on the Task object, which returns a reference for an appropriate plug-in to the ESPA SC.

10. The request is propagated to the plug-in. A listener object is provided in the request. Only one listener is used and different request/response pairs are matched together using transaction IDs.

   **Note:** Timeout timers should be used to monitor the response. If a response arrives successfully, the timer should be cancelled.

11. The plug-in translates the request according to appropriate network protocols. The operation shown in the diagram is asynchronous to the network, so the call to the network returns immediately. The actual result of the operation will be received later and returned using the listener callback object provided by the ESPA SC.

12. When the response to the request reaches the plug-in, it uses the listener object to propagate the response to the ESPA SC.

13. When the ESPA SC receives the response, a new thread is created using the SLEE Task Manager and returns.

14. In the new thread, a CDR is created using the SLEE Charging service.

15. The response is propagated from the ESPA SC listener to the SESPA SC listener.

16. The SESPA SC listener performs a notify on itself, releasing itself from the wait. The response is propagated through the SESPA SC to the WS layer and is then returned to the application.

   **Note:** In the 2.2 version of the Extension SDK, SLEE services, including SESPA, have access to the AXIS classes necessary for creating SOAP messages directly, so it is also possible for SESPA to send the response directly, without using the WS layer as an intermediary. This is now the default practice.
Network-triggered

The third sequence follows a request to an application triggered from the network, as, for
example, when the network notices that a particular terminal/user has entered a designated
location. The scenario described below is an example of an synchronous call from the network.
There are two parts to processing network-triggered requests:

1. The application must register its interest in a particular event and provide the Network
   Gatekeeper with an end-point (a URL at which the application has set up a Web Service) to
   which the notification can be sent.

2. At some later point, the network notifies the Network Gatekeeper that the event has occurred,
   which is then forwarded on to the registered end-point.
Figure 3-3  Network-triggered sequence
Registering the listener

1. The application sends a SOAP-encoded request (SOAP) to the Web Services layer implementation (WS layer), which is deployed in the Embedded Tomcat server. The request (addListener) provides a URL to an end-point at which the application has implemented a Web Service to receive notifications about network-initiated events.

2. The WS layer removes the SOAP envelope and transforms the request to add the listener into a normal Java object. It is then passed to the stateless adapter layer (SESPA SC) by means of a registration mechanism facilitated by the SLEE Common Loader. Because the internals of the Network Gatekeeper are stateful, traffic that flows through it must be stateful, even though Web Services requests are, by their nature, stateless.

   Note: All calls across layers from this point are CORBA based.

3. SESPA SC instantiates a listener class which will serve as a callback object when the network-triggered request is propagated through the service capability layer (ESPA SC) to SESPA SC.

4. The ESPA SC bundles the application’s request (addListener) into a Policy Request object, which is sent to the Policy Service, where an evaluation (eval) based on the rules is performed. In the ESPA SC, this is called a Policy Enforcement Point (PEP). There are two possible outcomes as a result of this evaluation:

   a. If the application’s request is denied by the Policy Service, a PolicyException is sent from the ESPA SC to the SESPA SC. This exception is propagated to the WS layer and finally returned to the application.

   b. If the application’s request is accepted by the Policy Service, the Policy Service examines the original request parameters and may, in some situations, update them. These parameters are then returned to the ESPA SC. The updated parameters are what will be used from this point forward.

5. The ESPA SC calls getResourceCtx on the ResourceDiscovery interface of the Plug-in Manager. The request includes the type of plug-in necessary as well as a property that queries whether the plug-in is capable of handling network-initiated traffic. A plug-in is returned to the ESPA SC.

6. To support high availability, Network Gatekeeper usually operates in a clustered environment, so the ESPA SC adds the reference to the database and distributes the reference to other ESPA SCs.

7. The ESPA SC also returns an ID for the listener to SESPA SC, which again, because of the clustered environment, updates the database with the reference and distributes the reference
to other SESPA SCs. All other SESPA instances should also register listeners when the reference is distributed.

8. The ID is returned to the originating application. This ID is used if the application ever decides to remove the listener.

9. Once a plug-in has been chosen, the ESPA SC invokes `addListener` on it and the plug-in starts to listen for the designated network-initiated events.

   As a way of reducing the number of objects, it is possible to arrange programmatically for a single listener to be registered in the ESPA SC from its start-up (technically, when it is put in the activated state). In this alternate pattern, the plug-in then distributes all incoming events to this one listener.

**Handling incoming events**

1. A network-initiated event is routed to the plug-in.

2. The plug-in asks the SC manager which ESPA SC has registered for this type of event, using the method `getSCSCtx` on the `SCSDiscovery` interface. (The SC manager serves the same routing function for network-triggered requests that the plug-in manager serves for application-initiated requests.) The parameters provided are matched to the parameters given when the listener was created.

3. A list of matching ESPA SCs is returned, and the plug-in invokes the appropriate “notify” method on the ESPA SC. The specific method depends on the service capability.

4. A new thread is created using the SLEE Task Manager. The ESPA SC finds all registered listeners in SESPA. It also performs a policy evaluation of the incoming request. (Not shown in the diagram.) The calling thread is set into wait state.

5. The method request is propagated to the SESPA SC listener, where a new thread is created using the SLEE Task Manager.

6. The method request is propagated to the WS layer, which then forwards it on to the application at the end-point specified when the application registered for the notification.

**Note:** In the 2.2 version of the Extension SDK, SLEE services, including SESPA, have access to the AXIS classes necessary for creating SOAP messages directly, so it is also possible for SESPA to send the response directly, without using the WS layer as an intermediary. This is now the default practice.

7. When the response returns from the application, it is sent through the layers. A new thread is created using the SLEE Task Manager.
Traffic Path Flow

8. When it reaches the ESPA layer, whatever charging functions are necessary are performed and a response method is called on the plug-in.

9. The plug-in awakes from its wait state by calling notify.

10. The plug-in returns the call from the network.
Thinking about WebLogic Network Gatekeeper in terms of the Traffic Path Map emphasizes through and through data flow. But there is another useful way of thinking about the Network Gatekeeper: as a set of services that execute (mostly) in WebLogic Network Gatekeeper Service Logic Execution Environments (SLEEs). The SLEE is a kind of container and handles many basic functions (life cycle, service supervision, and database supervision) for the services that run inside it. All the functionality of the Web Services, SESPA SC, ESPA SC, and network plug-in layers all run as services inside these CORBA based SLEEs.
If you are extending the functionality of one of those services, or creating your own, you need to understand what the SLEEs offer and how your software should interact with them. This section will give you a high level understanding of how that works.

**SLEE Basics**

This is a very quick overview of SLEE functioning. For more information, see the Architectural Overview and System Administrator’s Guide from the WebLogic Network Gatekeeper documentation set.

**SLEE States**

SLEEs are the containers in which SLEE services (most of the modules that make up WebLogic Network Gatekeeper) run. SLEES have one of three states: SHUTDOWN, RUNNING, and SUSPENDED. When a SLEE process is started up, it is in the SHUTDOWN state, as in Figure 4-2.
Before services can begin executing, the SLEE’s state must be changed to RUNNING. Doing this will cause all autostarted services installed in the SLEE to be automatically started and activated. If the SLEE state is changed from RUNNING to SHUTDOWN, all services executing in the SLEE will be stopped. The SUSPENDED state temporarily stops all requests into and out of the SLEE without actually stopping the started or activated SLEE services.

**SLEE service states**

All software modules installed and run in the SLEE are SLEE services. A SLEE service can have one of five states (see also Figure 4-3):

- **Installed**
  The service software is installed in the SLEE.

- **Started**
  The service is started and available in the Network Gatekeeper Management Tool but cannot send and receive CORBA requests.

- **Activated**
  The service is running and can send and receive CORBA requests.

- **Suspended**
  The service is activated but cannot receive new service requests. Used for graceful service shutdown.
The SLEE and Interacting with SLEE Services and Utilities

- Error
  The service has raised too many critical alarms and has been taken out of service by the SLEE. The allowed number of critical alarms is configured at service installation.

![SLEE service states diagram](image)

**Figure 4-3** SLEE service states

**Interacting with SLEEs**

The modules that you create in extending the Network Gatekeeper’s functionality will run as services in a SLEE. The rest of this chapter describes the mechanisms available to allow your services to interact with SLEEs, both basic SLEE processes and SLEE utility capabilities and other SLEE services.

**Note:** All of these interactions can only be performed by modules executing internally in the SLEE.

The following sections describe the interfaces and classes that allow your module(s) to interact with the SLEE:

- Implementing SLEE Service Behavior
Implementing SLEE Service Behavior

SLEE services have three levels of behavior in relation to the SLEE and other SLEE services:

- All services that are to be installed and run in a SLEE must be at least **deployable**.

- Services that need to be able to communicate with the SLEE and other SLEE services using CORBA must also be **accessible**.

- Services that need to be managed at runtime through the WebLogic Network Gatekeeper Management Tool must also be **manageable**.

In general, the modules you will be creating will need to be at least deployable and accessible. If there are any features that may need to be set at runtime, the module will also need to be manageable. To do this, your modules must implement the following interfaces:

- SLEE utilities accessed through the ServiceContext and SLEEContext interfaces
- Operation, Administration, and Management (OAM)
- Using the database
- Using the Configuration Store Service
- Using the Global Store Service
- Using the Alarm Service
- Using the Event Log Service
- Using the Event Data Record Service
- Using the Charging Service
- Using Parameter Tunneling
- Using the Time Service
- Using the Trace Service
  - Context Trace
ServiceDeployable

In order to deploy your service in the SLEE at all, you must implement ServiceDeployable. This interface allows the SLEE to notify the service of its deployment status, or state. See “SLEE service states” on page 4-3 for more information on states.

The SLEE uses the object implementing this interface to call the following methods on your service:

- `started()` - Called by the SLEE when the service has been started
- `activated()` - Called by the SLEE when the service has been activated (is in normal running state)
- `deactivated()` - Called by the SLEE when the service is to be deactivated.
- `stopped()` - Called by the SLEE when a service is to be stopped.
- `setServiceContext(...)` - Called by the SLEE to set the Service Context. An object representing the Service Context is passed in. This `ServiceContext` object is used to access all other SLEE services; for more information see “The ServiceContext object and accessing SLEE utilities” on page 4-7.

ServiceDeployableExt

If your service supports the suspend/resume states, you must also implement the ServiceDeployableExt - the Service Deployable Extension - interface.

The interface extends ServiceDeployable by adding following operations:

- `getNumberOfActiveSessions()` - Returns the number of active sessions the service holds.
- `resume()` - Causes the service to change its suspended state back to active state
- `suspend()` - Suspends the service

ServiceAccessible

To allow your service to communicate with the SLEE and other SLEE services, you must implement the ServiceAccessible interface. The ServiceAccessible interface represents the object that is installed in the SLEE service ORB, allowing your service to be accessed by other SLEE services. The class implementing this interface MUST have an empty public constructor.
The CORBA POA is supplied by the SLEE using the method `setPOA(...)` and the SLEE service must implement and return this same POA using the method `public org.omg.PortableServer.POA default_POA()`. Your SLEE service uses this POA, or a child POA, when creating new CORBA objects.

The SLEE calls the following method on the object implementing this interface:

- `setServiceContext(...)` - Called by the SLEE to set the Service Context. An object representing the Service Context is passed in. This `ServiceContext` object is used to access all other SLEE services; for more information see “The ServiceContext object and accessing SLEE utilities” on page 4-7.

**ServiceManageable**

If your service exposes methods to allow the Network Gatekeeper Management Tool to interact with the service at runtime, it must implement the `ServiceManageable` interface. This interface represents the object that is installed in the name service as the OAM (Operation Administration and Management) object for the service. Note that the `ServiceManageable` interface extends `ServiceCORBAServant`. See “Operation, Administration, and Management (OAM)” on page 4-19.

The SLEE calls the following method on the object implementing this interface:

- `setServiceContext(...)` - Called by the SLEE to set the Service Context. An object representing the Service Context is passed in. This `ServiceContext` object is used to access all other SLEE services; for more information see “The ServiceContext object and accessing SLEE utilities” on page 4-7.

**The ServiceContext object and accessing SLEE utilities**

The `Service<name>` interfaces provide a method for the SLEE to pass in a `ServiceContext` to your modules. This object allows access to a wide range of SLEE services and utilities.

**Overview**

The `ServiceContext` object represents the context of a service; that is, it a way of accessing handles to other SLEE services. All service behavior interfaces have a `setServiceContext` method that is called from the SLEE before calling the method `started()` on the service. This method passes in the `ServiceContext` object.

Using this interface your service can get a handle to the following SLEE services:
The SLEE and Interacting with SLEE Services and Utilities

- The charging service: see “Charging Service” on page 4-10.
- The policy service: see “Policy Manager” on page 4-10.
- The SLEE Event Channel service: see “SLEE Event Channel” on page 4-10.
- The TraceLogService service: see “Trace Log Service” on page 4-11.
- ServiceInstanceHandler: see “Service Instance Handler” on page 4-11.
- The Configuration Store: see “Configuration Store” on page 4-11.

Your service can also use the Service Context object to get references to any objects in the SLEE implementing the following interfaces:

- A ServiceAccessible object implemented by the SLEE service: see “ServiceAccessible” on page 4-6.
- A ServiceDeployable object implemented by the SLEE service: see “ServiceDeployable” on page 4-6.
- A ServiceManageable object implemented by the SLEE service, see “ServiceManageable” on page 4-7.

The ServiceContext object also provides access to a handle to the SLEE Context service, which is used to get other SLEE and utility services. See “Services fetched from the SLEE Context” on page 4-11.

Using the ServiceContext object it is also possible to:

- Get the name of the jar file in which this service was installed
- Get the name of the service
- Get the services’s state

The SLEE Context

The SLEEContext object represents the SLEE context for a service; that is, it provides an initial object for retrieving a number of SLEE and SLEE utility services. A SLEEContext object is accessed by your service via the ServiceContext object provided by the SLEE. Your service can use the SLEE Context to access a number of SLEE and other utility services.

Note: There is also a ClusterContext object that provides access to context information across the multiple servers of a cluster. See “Context Trace” on page 4-66 and “Using Parameter Tunneling” on page 4-51 for more information on its use.
Using this interface your service can access the following SLEE and other utility services:

- The Alarm Service: see “Using the Alarm Service” on page 4-34.
- The Cyclic ID manager: see “SLEE Cyclic ID Manager” on page 4-16.
- The Database Manager: see “SLEE DB Manager” on page 4-12.
- The Event Log: see “Event Log Service” on page 4-16.
- The Event Data Record Service: see “Event Data Record Service” on page 4-16.
- The Global Counter and the Global Counter Manager: see SLEE Global Counter and “SLEE Global Counter Manager” on page 4-17.
- The ID Manager: see “SLEE ID Manager” on page 4-17.
- The Load Manager: see “SLEE Load Manager” on page 4-12.
- The Plug-in Manager: see “SLEE Resource Manager” on page 4-13
- The Service Capability Manager: see “Network Plug-in to Service Capability Manager” on page 5-23.
- The Statistics Manager: see “SLEE Statistics Manager” on page 4-13.
- The Task Manager: see “SLEE Task Manager” on page 4-14.
- The Time Manager: see “SLEE Time Manager” on page 4-15.
- The Supervised List service: see “Supervised List” on page 4-17.

**SLEE utilities accessed through the ServiceContext and SLEEContext interfaces**

This section describes in a bit more detail the SLEE services and utilities that your service can access using the ServiceContext object passed into your service at start-up or the SLEEContext object that you can access through the ServiceContext object. Some of the services are fairly simple, and examples are offered in this section. Some of the services require more space and they will be summarized here and presented at greater length elsewhere. For the complete description of all the SLEE utility classes and methods, see the SLEE JavaDoc.
Services fetched from the Service Context

Charging Service
Use this service to create CDRs (charging records) in the Network Gatekeeper database. For more information, see “Using the Charging Service” on page 4-47.

Policy Manager
Use this service for retrieving the Policy service in order to create a Policy Enforcement Point (PEP). For more information, see “Using the PolicyManager” on page 4-36

SLEE Event Channel
Use this service to broadcast events and to register listeners for events originating in other SLEE services. It is used to send events (data) from one SLEE service to another, for example, in propagating changes of cached data to other instances of a service.

Below is pseudo code for broadcasting an event via the SLEEPEventChannel.

Listing 4-1  Broadcasting an event

```java
//m_sc holds the ServiceContext passed in from the SLEE
String eventId = "eventID";
EventData eventData = new EventData();
org.omg.CORBA.Any event;
event = eventData;
   m_sc.getSleeEventChannel().generateEvent(eventId, event);
```

SLEE Event Channel Listener
Below is pseudo code for receiving the event.

Listing 4-2  Receiving an event - a class that implements SLEEPEventChannelListener

```java
public void processEvent(String eventId,
```
org.omg.CORBA.Any event,
String source) {
    if (eventId == "eventID") { // Check the event ID
       EventData eventData = (EventData)event; // Cast to proper class
    }
    // source contains the instance ID of the SLEE service.
    // m_sc holds the ServiceContext passed in from the SLEE
    if (source.equals(m_sc.getInstanceName())) {
        // The event originated from this instance.
    }
}

Trace Log Service
Use this service to write tracing information to a specific trace file. Because the service may be
disabled in a particular environment for performance reasons, each service should check the trace
active flag before calling the logTrace method. For more information, see “Using the Trace
Service”.

Service Instance Handler
Use this service to locate multiple instances of a service. It provides IORs to the
ServiceAccessible interface to other SLEE services.

Configuration Store
Use this service to store configuration (not traffic) data in a way that is simpler to use than the
SLEE DB Manager. For more information, see Using the Configuration Store Service.

Services fetched from the SLEE Context
The services in the first part of the following list belong to the SLEE proper. The services in the
second part are made up of services that are (in the context of the codebase) outside of the SLEE
itself, but are still made available through the SLEE Context object.
**SLEE DB Manager**

Use this service to access the database. The SLEE DB Manager controls connections to the database for the entire SLEE. A SLEE service uses the DB manager to retrieve a DB connection. By default all users/SLEE services share the same database and database user identity. See “Using the database”. SLEE services that use JDBC must retrieve their connections using the SLEEDBManager instead of using the driver directly.

**SLEE Global Store**

Use this service to store transient data. Can be used instead of using the Database. See “Using the Global Store Service” on page 4-30.

**SLEE Load Manager**

Use this service to query the load of a SLEE. It reports the load as a number between 0 and 100.

**Note:** While it is possible to register and use listeners to report when load has reached a certain level, such reports are only generated every 5th second. So use of the synchronous method is a more reliable and granular way of monitoring the load.

**Listing 4-3   Getting the load on a SLEE Service**

```java
//m_sc holds the ServiceContext passed in from the SLEE
int loadValue _= m_sc.getSLEEContext().getLoadManager().getLoadValue();
```

**SLEE Load Manager Listener**

Use this interface to listen for periodic load level information. Listeners must be registered with the SLEE Load Manager using its `addListener(...)` method.

The Listener interface has a method `processLoadEvent(LoadEvent loadEvent)` which is used to deliver the load event information. The `LoadEvent` class reports the load level reached in the time interval - normally every 5th second. The available levels are:

- NORMAL
- OVERLOADED
- SEVERE_OVERLOADED
This service is part of the SLEE proper.

**SLEE Resource Manager**

Use this service to access the plug-in manager, which is charged with routing data from an ESPA SC to an appropriate network plug-in. See “Plug-in Manager interfaces” on page 5-10.

**SLEE Resource Listener**

Use this interface to listen for the addition of new network plug-ins.

**SLEE Resource Task**

Use this interface to get a reference to an appropriate network plug-in. The ESPA SC implements the interface, and then passes it as an argument to the `SLEEResourceManager.scheduleResourceTask` method. Once the plug-in is allocated, the SLEE calls its method `doTask(...)` on the object, passing a reference to the allocated plug-in as an argument.

**SLEE Statistics Manager**

Use this service to add new transaction types for your service capability modules, to be used for generating transaction statistics using the Statistics SLEE service. WebLogic Network Gatekeeper ships with a set of pre-defined transaction types, but new service capabilities must specify their own transaction types to allow for a transaction based pricing model. More complete examples of generating transaction statistics are provided in the template source code.

Extension-added transaction types must be associated with a numeric range from 1000 to 10000. New transaction types can be added to cover:

- All traffic generated by all applications
- All traffic per service provider
- All traffic per service provider and application.

Listing 4-4 is an example of updating the statistics counter.

Listing 4-4  Update statistics counter

```java
// m_sc holds the SLEEContext passed in from the SLEE
// serviceProviderID contains the service provider ID that performed the request
```
The new `transactionType` itself must be added to the SLEE Statistics service using the Network Gatekeeper Management Tool. That is, the value representing the new `transactionType` must be added using the Management method `addStatisticType` in the `SLEE_statistics` service. This information would need to be part of the documentation of your service.

**SLEE Task Manager**

Use the SLEE Task Manager to schedule a task that risks blocking resources. The SLEE Task Manager manages a pool of Java threads. Your code should always get a thread using the Task Manager. Never create Java threads directly.

**Listing 4-5  Pseudo code for scheduling a SLEETask**

```java
// m_sc holds the ServiceContext passed in from the SLEE
m_taskMgr = m_sc.getSLEECtx().getTaskManager();
m_taskMgr.scheduleSLEETask(new MyTask)
```

**SLEE Task**

Use SLEE Tasks for processing tasks that risk blocking resources, such as CORBA remote procedure calls, which should be performed asynchronously. SLEE Tasks should be used instead of using Java threads directly.
The module that wishes to acquire a SLEE task from the Manager implements the `SLEETask` interface and passes that object as an argument to `scheduleSLEETask` on the SLEE Task Manager (see Listing 4-6). As soon as one of the threads managed by the SLEE Task Manager becomes available `doTask` in the SLEETask will be called.

Listing 4-6  Pseudo code for a SLEETask

```java
public class MyTask implements SLEETask {
    public MyTask() {
    }

    // Will be called by the SLEETaskManager
    public void doTask() {
        // Perform the task.
    }
}
```

**SLEE Time Manager**

Use the SLEE Time Manager to schedule timers and get system time. For more information, see “Using the Time Service” on page 4-53

**SLEE Zombie Object Listener**

Instead of using this interface directly, use the Supervised List Listener interface.

The SLEE Zombie Object Listener interface handles state changes for zombie objects. If the zombie object becomes reachable again, or if it remains not alive within the AssumedDeadTimeout the listener will be notified.

**SLEE Zombie Object Supervisor**

Use the Supervised List class to supervise CORBA objects that may not be reachable but may become available in the future instead of using this class directly.
Task Chain
Use the Task Chain interface when tasks need to be executed in a given order.
Schedule the first task in a task chain using `SLEETaskManager.scheduleTaskChain`. This interface is returned and can then be used to add new tasks to the end of the chain.

Event Log Service
Use this service to write events to the common event database. See “Using the Event Log Service” on page 4-40 for more information.

Event Data Record Service
Use this service to send events to your listeners. See “Using the Event Data Record Service” on page 4-43 for more information.

Alarm Service
Use this service to generate alarms for events that need corrective action. See “Using the Alarm Service” on page 4-34.

Load Event
Use this class to fetch a definition of a load level event.

SLEE Cyclic ID Manager
Use this service to retrieve a unique ID of type `int`. The ID will be unique for all SLEE instances using the same database. This service is outside of the SLEE proper.

Note: The IDs will be reused when the ID of an `int` minus the most significant byte has been exceeded.

Listing 4-7   Below is pseudo code for how to get a unique ID

```
//m_sc holds the ServiceContext passed in from the SLEE
int ID = m_sc.getSLEEContext().getCyclicIDManager().getID()
```
SLEE DB Table
Use this class to check if a database table exists, or if it has a particular format, and to create or replace a table. It can also be used for creating temporary tables. SLEE services using the SLEE DB Manager to retrieve JDBC connections must use this class to create its tables and/or to get privileges to use the tables. See “Using the database” on page 4-20.

SLEE ID Manager
Use this service to retrieve a unique ID of type `long`. The ID will be unique for all SLEE instances using the same database.

Note: The IDs will be reused when the ID of a `long` minus the most significant byte has been exceeded.

SLEE Global Counter Manager
Use this service to maintain global counters. Global counter are automatically updated across all SLEE instances in a system.

SLEE Global Counter
Use this interface for incrementing or otherwise managing a global counter. Note that counter increments may not be distributed to all instances immediately. Updates are performed globally every 10th increment of a specific counter or every 5th second for all counters that have been updated less than 10 times during the last 5 second period.

Supervised List
Use this class to create a list for holding supervised CORBA objects, that is objects that should be moved to a Zombie Object Supervisor List if they are unreachable. In the Zombie Object list the objects are “pinged” for a certain time before they are considered dead and removed entirely from the supervised list.

Listing 4-8  Pseudo code for creating a supervised list.

```java
// m_sc holds the ServiceContext passed in from the SLEE
//listener implements the SupervisedListListener interface
SupervisedList m_supervisedList;
long timeBeforeDead = 10000; // Time given in milliseconds
```
m_supervisedList = new SupervisedList(m_sc.getSLEEContext(),
timeBeforeDead);
m_supervisedList.addListener(listener);

### Supervised List Listener

Use this interface to receive information about objects removed from a Supervised List.

#### Listing 4-9  Pseudo code for listening to events from object s in a supervised list.

```java
public class MyClass implements SupervisedListListener {
    MyClass() {
    }
    public void objectDead(SupervisedCorbaObject o) {
        // Object was removed due to zombie timeout.
        // Remove any references to the object.
    }
    public void objectZombie(SupervisedCorbaObject o) {
        // Object was declared a zombie and moved to zombie list.
    }
    public void objectLive(SupervisedCorbaObject o) {
        // Object was declared alive and put back into the list again
    }
}
```
All SLEE services that have features that need to be configured at runtime must implement the `ServiceManageable` interface so that the WebLogic Network Gatekeeper Management Tool can interact with them.

These management operations are defined first in IDL, and then Java stubs are generated from the IDL interface. This produces an abstract class named `<name of interface in IDL>POA` and the class that actually implements the OAM operations must extend this abstract class.

So, for example, if the IDL interface is `AServiceOAM`, the implementation class must extend `AServiceOAMPOA`. In the example below the implementation class is `AServiceOAM_impl`.

### Listing 4-10   Example of declaration of the `AServiceOAM` implementation class

```java
public class AServiceOAM_impl extends AServiceOAMPOA implements ServiceManageable
```

The IDL file defining the OAM interface must be packaged into the jar file for the software module, in its root directory. The deployment descriptor in the jar file lists the name of the IDL file for the OAM interface and the class that implements the interface, along with other information.

### Listing 4-11   Segment of example deployment descriptor

```xml
...<SERVICE_MANAGEABLE>
    com.my_company.my_plugin.AServiceOAM_impl
  <SERVICE_MANAGEABLE_IDL>
    AServiceOAM.idl
  </SERVICE_MANAGEABLE_IDL>
</SERVICE_MANAGEABLE>
...
```
The build files in the templates are used for creating the deployment descriptors and for packaging them into the deployable jar file.

**Implementing OAM access control**

Because not all users of the Management Tool have equal access to the Network Gatekeeper, implementing an OAM method for your service requires that you also implement an access method that checks to see if a particular user has permission to manipulate your service. The two methods are correlated by name. An OAM method with the signature `<method name>` must have a corresponding method with the signature `<method name>Allowed(...)`. The latter method is used to check the administrative user’s access privileges. See the example in Listing 4-12.

**Listing 4-12   Checking access to an OAM method**

```java
public void setOverloadPercentage(int percentage)
....

public boolean setOverloadPercentageAllowed(int userLevel) {
    return(userLevel >= SLEEOAM.SEC_LEVEL_READ_WRITE);
}
```

**Using the database**

The SLEE uses a DBMS that can be also be used by your service. Support is built into the framework for accessing the database using JDBC.

The SLEE allocates database connections that can be accessed from the SLEE Context. From the SLEEContext object you can retrieve a handle to the Database Manager, which is a singleton class that controls access to the database. By default all users and SLEE services share the same database. If your service needs to have its own tables, they must be created using the createSLEEDBTable() method. You cannot create tables using SQL directly.
This example (Listing 4-13) shows the creation of a temporary table, called mobile_users, with two columns: a user identity and its corresponding mobile subscription number. Such a table might be used, for example, to send SMS messages to a group of registered users. The createTemporaryTable() method is called when the service enters the activated state. Data is inserted through a method called register() which belongs to a class that implements the ServiceAccessible interface. To retrieve a list of all the users that are currently registered, the class uses a convenience method called getSubscribers(). Finally, to remove the table from the database when the service is deactivated, there is a removeTemporaryTable() method.

For the purpose of the example, we will create a class called DbRegister.java, that takes one argument in its constructor, a reference to the SLEE Service Context. The database manipulation is done through the database manager, SLEEDBManager.

Listing 4-13  DbRegister.java -initiation and constructor

```java
package example.helloslee;
import com.incomit.slee.*;
import java.sql.*;
import java.sql.*;
public class DbRegister {
    private ServiceContext itsServiceContext = null;
    private SLEEDBManager itsDbManager = null;
    // Constructor
    public DbRegister (ServiceContext aServiceContext) {
        itsServiceContext = aServiceContext;
        SLEEContext sc = itsServiceContext.getSLEEContext();
        itsDbManager = sc.getDBManager();
    }
}
```

The method createTemporaryTable, which takes no arguments, creates the table. It uses the utility method createSLEEDBTable() that returns a SLEEDBTable, to which the necessary
columns are added before physically creating the table in the database. The arguments to `addColumn` specify the column name, the data type, whether the column is used as the primary key, and if null values are allowed. It uses a pre-allocated connection resource that is served by the SLEE Database Manager.

Listing 4-14  DbRegister.java -method: createTemporaryTable

```java
public void createTemporaryTable () {
    try {
        Connection conn =
          itsDbManager.getConnection();
        try {
            SLEEDBTable table =
              itsDbManager.createSLEEDBTable("mobile_users");
            table.addColumn("userName", "VARCHAR(20)", true, false);
            table.addColumn("msisdn", "VARCHAR(28)", false, true);
            if (!table.exists(conn)) {
                table.create(conn);
            }
        }
        catch(Exception e){
            // Handle exception
        }
        finally {
            if (conn!=null) {
                conn.close();
            }
        }
    } catch (SQLException e) {
```

New rows are inserted in the table using the method `register()`, which takes a user name and a mobile subscription number as its arguments. The method uses JDBC methods to create a statement and execute the update in the database.

### Listing 4-15  DbRegister.java - method: register

```java
public void register(String user, String msisdn) {
    try {
        Connection conn =
            itsDbManager.getConnection();
        try {
            Statement stmt = conn.createStatement();
            try {
                stmt.executeUpdate("INSERT INTO " +
                    "mobile_users VALUES (" +
                    user + ", " + msisdn + ")");
            } catch (Exception e) {
                // Handle exception
            } finally {
                stmt.close();
            }
        } catch (Exception e) {
            // Handle exception
        }
    }
}
```
A convenience service in the class, `getSubscribers()`, returns all registered subscribers. The method takes no arguments, and executes a SQL `SELECT` query on the database. The Result Set is converted to a `Hashtable` and returned to the calling method.

Listing 4-16  DbRegister.java - method: getSubscribers

```java
public Hashtable getSubscribers () {
    Connection conn = itsDbManager.getConnection();
    Statement stmt;
    ResultSet rs;

    try {
        stmt = conn.createStatement();
        Hashtable table = new Hashtable();
        String query = "SELECT userName, msisdn " +
                        "FROM mobile_users";
        rs = stmt.executeQuery(query);
```
while (rs.next()) {
    table.put(rs.getString(1), rs.getString(2));
}
return table;
}
catch (SQLException e) {
    // Handle exception
}
finally {
    if (rs!=null) {
        rs.close();
    }
    if (stmt!=null){
        stmt.close();
    }
    if (conn!=null) {
        conn.close();
    }
}

To remove the table from the database when it is no longer needed, removeTemporaryTable() is used. It also uses a connection resource from the SLEE Database Manager, and a SQL DROP TABLE command in its JDBC execute update statement.

Listing 4-17   DbRegister.java -method: removeTemporaryTable

    public void removeTemporaryTable () {

try {
    Connection conn =
    itsDbManager.getConnection();
    Statement stmt;
    try {
        stmt = conn.createStatement();
        stmt.executeUpdate("DROP TABLE mobile_users");
    }
    catch (Exception e) {
        // Handle exception
    }
    finally {
        if (stmt!=null) {
            stmt.close();
        }
        if (conn!=null) {
            conn.close();
        }
    }
} catch (SQLException e) {
    // Handle exception
}
Using the Configuration Store Service

The Configuration Store is used for storing configuration data in the database, as an alternative to using the SLEE DB Manager. When using the Configuration Store, all JDBC and SQL code is transparent, which reduces the complexity of the code.

**Note:** Traffic-related data should not be stored using the Configuration Store.

This utility stores the configuration data in a common table in the database and provides an easier and more direct way of storing data compared to using the SLEE DB manager.

The Configuration Store can handle both configuration data that is local for a single SLEE service instance and configuration data that is shared among multiple SLEE service instances. Shared data is available to all instances of a particular service, cluster-wide. Data can only be shared among SLEE services with the same service name.

The Configuration Store stores data in a named Configuration Store. Within the Configuration Store, individual elements are stored and retrieved using a key.

Supported data types are:

- Boolean
- Integer
- Long
- String

All data stored in a Configuration Store are stored in a table in the database, named when the Configuration Store is created. The name should be the name of the SLEE service with the suffix `_store`. If it is a shared store, the key is the SLEE Service name; if it is not a shared store, it is a combination of the SLEE instance name and SLEE Service name. Data in a Configuration store is also cached in memory and in the case of a shared store, this cache is cluster-wide.

The Configuration Store also provides registration of listeners that will receive notifications when an element has been changed in another instance of the SLEE service.

The Configuration Store service is fetched from the Service Context.

When fetching the Configuration Store, the name of the store and the type is specified.
Listing 4-18  Fetching the Configuration Store

```java
m_configStore = m_sc.getConfigurationStore(CFG_STORE_NAME,
ConfigurationStore.STORE_TYPE_SHARED);
```

- `m_sc` is the Service Context.
- `CFG_STORE_NAME` is the name of the Configuration Store to use.
- The type is either `ConfigurationStore.STORE_TYPE_SHARED` if the data in the store should be accessible from all instances of the SLEE service (cluster-wide), OR
- `ConfigurationStore.STORE_TYPE_LOCAL`, it should be accessible only from the local instance of the SLEE service.

If the store is not yet created, it will be created when `getConfigurationStore` is invoked.

All data to be stored in the Configuration Store must be initialized with default data using the following methods in the class `ConfigurationStore`:

```java
void initialize(java.lang.String key, java.lang.Boolean defaultValue)
void initialize(java.lang.String key, java.lang.Integer defaultValue)
void initialize(java.lang.String key, java.lang.Long defaultValue)
void initialize(java.lang.String key, java.lang.String defaultValue)
```

The parameter `key` is used to uniquely identify the parameter to be stored within the Configuration Store.

**Note:** All data to be stored should be initialized with default values when the SLEE service starts.

Listing 4-19  Initializing a Configuration Store

```java
String MAX_BATCH_SIZE_KEY = "max_batch_size";
Integer MAX_BATCH_SIZE_DEFAULT = new Integer(50);

m_configStore.initialize(MAX_BATCH_SIZE_KEY,MAX_BATCH_SIZE_DEFAULT);
```
The key is used to get the parameter using the following methods on the class ConfigurationStore:

```java
java.lang.Boolean getBoolean(java.lang.String key)
java.lang.Integer getInteger(java.lang.String key)
java.lang.Long getLong(java.lang.String key)
java.lang.String getString(java.lang.String key)
```

Below is an example of getting the value of the parameter MAX_BATCH_SIZE_KEY.

**Listing 4-20  Get a parameter**

```java
java.lang.Integer maxBatchSize =
m_configStore.getInteger(MAX_BATCH_SIZE_KEY);
```

A parameter can be stored using the following methods on the class ConfigurationStore:

```java
void setBoolean(java.lang.String key, java.lang.Boolean value)
void setInteger(java.lang.String key, java.lang.Integer value)
void setLong(java.lang.String key, java.lang.Long value)
void setString(java.lang.String key, java.lang.String value)
```

Below is an example of setting the value of the parameter MAX_BATCH_SIZE_KEY.

**Listing 4-21  Set a parameter**

```java
Integer newValue = new Integer(100);
m_configStore.setInteger(MAX_BATCH_SIZE_KEY, newValue);
```

There are situations when one instance of a SLEE service needs to know that other instances of the same service have changed or updated some parameter value, so that, for example, they can update their own in-memory representation of the parameter. If this is the case, the SLEE service can use a listener based on the `ConfigurationListener` interface to receive notifications of the changes. Add and remove the listener using the methods `addListener(ConfigurationListener listener)` and `removeListener(ConfigurationListener listener)`. 
The listener must implement the method:

```java
configurationChange(ConfigurationStore store, java.lang.String key)
```

The name of the Configuration Store can be retrieved using `java.lang.String getStoreName()`. You can use `getStoreType()` to retrieve the type of the store (`ConfigurationStore.STORE_TYPE_SHARED` or `ConfigurationStore.STORE_TYPE_LOCAL`).

### Using the Global Store Service

The Global Store service serves as a convenient way to store data in memory in a controlled manner, and to persist it to the database under certain conditions.

It is designed to store transient data, which is likely to have a short to mid-term lifetime, such as request data. The Global Store is replicated to other SLEEs. When the Global Store reaches a certain size, it starts to flush data to persistent storage.

When a data item in the Global Store reaches the `flush age`, it is flushed to persistent storage. When a data item in the Global Store reaches the `expiration age`, it is removed from the Global Store. A user can subscribe for notifications when individual objects expire. To keep items from being persistently stored, the `flush age` and the `expiration age` can be set to the same value. In this case, nothing will be sent to persistent storage unless the `maximum in-memory store size` is reached. The `flush age`, `expiration age` and `maximum in-memory store size` are all configurable via OAM.

Data in the Global Store is accessed only via the Global Store, which is a named store which is created programmatically and stored and retrieved by name. Whether the data is fetched from memory or from the persistent storage is transparent to the user of the Global Store. The name for the specific item object must be unique within the Global Store.

To create a Global Store, three parameters must be furnished:

- A name
- A factory object which used to create the data item object instances to be stored
- A set of properties

```java
public GlobalObjectStore createObjectStore(java.lang.String name,
                                         GlobalObjectFactory objectFactory,
                                         GlobalObjectProperty[] properties)
```
The data that are stored in the Global Store must be member variables of a class extending GlobalObject. The following data types are supported:

- String
- int
- long
- byte
- short
- char
- boolean
- double
- float

The member variables must be declared private. Getting and setting the member variable must be via get- and set-methods. The set- and get-methods operating on a given member variable must have the same name, except for the set- and get-prefix. The name is given as a property when the Global Store is created. The Global Store name, given in the constructor of the GlobalObject, will be the name of the table in the database, and the suffixes in the get and set methods will be column names in the table.

**Example**

We want to store two data elements in the Global Store MyGS, one int and one String. A class that contains the following member variables is created:

```java
private String dataA;
private int dataB;
```

The class is declared:

```java
public class MyData extends GlobalObject
```

The class defines the following methods:

```java
public String getDataA()
public void setDataA(String)
public int getDataB()
```
public void setDataB(int)

A factory class must be created for the class MyData. This class is used when creating the Global Store. The factory class implements GlobalObjectFactory, which means that the method newInstance must be defined. This method instantiates and returns the class that holds the member variables.

```java
class MyDataFactory implements GlobalObjectFactory {
    public GlobalObject newInstance(String name, String id) {
        return new MyData();
    }
}
```

When creating a Global store that stores the member variables, a descriptor of the get- and set-methods must be defined.

The descriptor is an array of GlobalObjectProperty that defines each data item to be stored.

```java
GlobalObjectProperty[] props =
    new GlobalObjectProperty[] {
        new GlobalObjectProperty("DataA", "dataa", 255),
        new GlobalObjectProperty("DataB", "datab")};
```

Different constructors are used, depending on the type of the member variable.

The first argument in the constructor for GlobalObjectProperty is the name of the store, this is also the table name. The second argument is the name of the column in the database table used to store the member variable, and it must be exactly the same as the suffix of the get- and set methods for the member variable. In case of a String, the maximum length is passed as a third argument in the constructor.

It is also possible to define if the entry shall be indexed. See the JavaDoc for full details.

The Global Store is created using a call to createObjectStore(...). This method is called on the Global Store Manager, which is fetched from the SLEE Context.

```java
m_mysvcStore =
m_sc.getSLEEContext().getGlobalStoreManager().
createObjectStore("MyGS", new MyDataFactory(), props);
```

m_sc is the service Context.
A handle to the Global Store is returned.

When the Global Store is created it is possible to perform the following operations on the Global store:

- String add(GlobalObject obj), adds an object to the store, an ID is returned.
- void String add(java.lang.String id, GlobalObject obj), adds an object to the store, with a specified ID. The service must insure that the ID is unique within the cluster.
- GlobalObject get(java.lang.String id), gets an object from the store by its ID.
- String getName(), gets the Global Store name.
- GlobalObject remove(java.lang.String id), gets an object from the store and removes it.
- boolean replace(java.lang.String id, GlobalObject object), replaces the current object in the store with a new. The timestamp remains the same as when the object initially was added to the Global Store. This method is not synchronized, so it may cause inconsistencies if called simultaneously.
- void close(), closes the Global Store.
- void addExpirationListener(GlobalObjectExpirationListener listener), adds a listener that will be notified when objects expires.
- void removeExpirationListener(GlobalObjectExpirationListener listener), removes a previously added expiration listener.

**Note:** The properties Expiration age, Flush interval, and Cache max size should be defined for this named Global Store. This is done using the Management Tool, in the SLEE_global_store service. See Network Gatekeeper System Administrator’s Guide, chapter Tuning Performance for information on how to define the properties. If the parameters have not been set, default values will be used:

- Expiration Age 1 day (86 400 000 milliseconds)
- Flush Interval 60 seconds (60 000 milliseconds)
- Cache Flush Age 1 hour (3 600 000 milliseconds)
- Cache max size 10000 objects.

When defining the values, all times are given in milliseconds.
Using the Alarm Service

Alarms in the SLEE are handled by a dedicated alarm service. All alarms reported from an application are written to an alarm log in the database. There is only one table in the database for logging alarms, which means that all applications using the alarm service share the same table in the database.

Alarms are critical events that need corrective action. If an application raises too many alarms, it is taken out of service. The maximum number of critical alarms allowed for an application before it should be taken out of service can be configured in the deployment descriptor.

In this example we show setting up a reference to the alarm service, and generating alarms to be stored in the alarm log.

An alarm record contains the following information:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alarm Number</td>
<td>An alarm identifier, for example indicating the type of the alarm. For extension services, the alarm number must be in the range 500 000 to 999 999.</td>
</tr>
<tr>
<td>Severity</td>
<td>The severity of the alarm. Pre-defined constants are defined in the AlarmService interface, and should be used: AlarmService.MINOR, AlarmService.WARNING, AlarmService.MAJOR, AlarmService.CRITICAL</td>
</tr>
<tr>
<td>Service Instance Name</td>
<td>The source of the alarm, that is, the unique name for this service. Retrieved by using getInstanceName() on the ServiceContext object.</td>
</tr>
<tr>
<td>Time and Date</td>
<td>The time-stamp is be generated by the system when it receives an alarm</td>
</tr>
<tr>
<td>Additional Information</td>
<td>Any other pertinent information about the alarm. For example the stack trace can be included, if it is available.</td>
</tr>
</tbody>
</table>
The method `fireAlarm()` is used to report an alarm, and has the following signature:

```
public void fireAlarm(byte[] source, int severity, int identifier, byte[] info)
```

The following example shows how to set up and use the alarm service. It is in the form of a utility class that uses the Service Context to retrieve a handle to the alarm service. The method `raiseAlarm()` fires a simple alarm that will be written to the system alarm table in the database. The method takes one argument of the type `Throwable`, and includes the stringified error message in the alarm log.

Listing 4-22  Alarm.java

```java
package example.helloslee;

import com.incomit.slee.*;
import com.incomit.slee.alarm.*;

public class Alarm {

    private ServiceContext itsServiceContext;
    private AlarmService itsAlarmService;

    private int alarmNumber = 4711;

    public Alarm (ServiceContext aServiceContext) {
        itsServiceContext = aServiceContext;
        SLEEContext sc = itsServiceContext.getSLEEContext();
        itsAlarmService = sc.getAlarmService();
    }

    public void raiseAlarm (Throwable e) {
```
Using the PolicyManager

The Policy Manager is used for retrieving the Policy service in order to create a Policy Enforcement Point or PEP. This normally occurs in the ESPA SC Layer. The PEP interacts with a Policy Decision Point (PDP) in the policy engine, which evaluates the request in terms of the rules.
This following code is a sketch of implementing a Policy Enforcement Point. See MyServiceCapabilityManager_impl.java in module_templates\espa_sc_impl\src\com\acompany\espa\mysctype\ for further implementation details. For more information on the Policy service in general, see “Policy rules and Policy Utilities” on page 16-1.

First, the PolicyManager is fetched from the Service Context.
Listing 4-23  Fetching the PolicyManager

```java
policyManager = MyServiceCapabilityContext.getServiceContext().getPolicyManager();
```

Then a PolicyRequest object, the object that holds the data to be forwarded to the Policy Decision Point, is created. Included in the data are the application and service provider IDs for the originating application.

Listing 4-24  Creating a PolicyRequest object

```java
MyServiceCapabilityContext.getServiceContext().getSLEEContext().getTimeManager().getTime();
PolicyRequest policyRequest = new PolicyRequest_impl(
    MyServiceCapabilityContext.getServiceContext().getSLEEContext().getORB(),
    timeStamp);
    try {
        policyRequest.applicationID = m_ApplicationID;
        policyRequest.serviceProviderID = m_ServiceProviderID;
    } catch (com.incomit.espa.access.AccessException e) {
        ...}
    policyRequest.serviceName = MyServiceCapabilityContext.POLICY_SERVICE_NAME;
    policyRequest.serviceGroup = MyServiceCapabilityContext.POLICY_SERVICE_GROUP;
    policyRequest.methodName = "myMethod";
    policyRequest.serviceCode = serviceCode;
    policyRequest.requesterID = requesterID;
```
policyRequest.transactionID = -1;
policyRequest.noOfActiveSessions = -1;
policyRequest.reqCounter = 0;
AdditionalData adArray[] = new AdditionalData[2];
AdditionalDataValue targetAddressValue = new AdditionalDataValue();
AdditionalData adTargetAddressString = new AdditionalData();
targetAddressValue.stringValue(address);
adTargetAddressString.dataName = "targetAddress";
adTargetAddressString.dataValue = targetAddressValue;
adArray[0] = adTargetAddressString;
AdditionalDataValue dataValue = new AdditionalDataValue();
AdditionalData adDataString = new AdditionalData();
dataValue.stringValue(data);
adDataString.dataName = "data";
adDataString.dataValue = dataValue;
adArray[1] = adDataString;
policyRequest.additionalParameters = adArray;

Then the Policy Request data is sent to the Policy Decision Point to be evaluated.

Listing 4-25  Evaluate the Policy data

PolicyRequest modifiedRequest =
(PolicyRequest) policyManager.evaluate((PolicyRequest) policyRequest);

If the request is denied, a DenyException is thrown and a CDR is created with completion status set to Policy Rejected.
If the request is accepted, the PDP may modify the original request data before returning it to the PEP. So the potentially-modified data should be used when propagating the request instead of the original data.

**Listing 4-26  Use the request data returned from the PolicyRequest**

requestContext = new RequestContext(modifiedRequest.requesterID,
                                           modifiedRequest.serviceCode);
address = modifiedRequest.getAdditionalDataStringValue("targetAddress");
data = modifiedRequest.getAdditionalDataStringValue("data");

**Using the Event Log Service**

Events in the SLEE are handled either by the dedicated Event Log service or by the Event Data Record Service, covered in “Using the Event Data Record Service” on page 4-43. All events designated as belonging to the event log service are written to the log in the database. There is only one table in the database for logging events, which means that all applications using the Event Log Service share the same table in the database.

Events are expected occurrences of importance to the operator.
An event record contains the following information:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Number</td>
<td>An event identifier, for example indicating the type of the event. For extension services, the event number must be in the range 500 000 to 999 999.</td>
</tr>
<tr>
<td>Level</td>
<td>The level of the event. Pre-defined constants are defined in the EventLogService interface, and should be used: EventLogService.LOW, EventLogService.MEDIUM, EventLogService.HIGH</td>
</tr>
<tr>
<td>Service Instance Name</td>
<td>The source of the event, that is, the unique name for this service. Retrieved by using getInstanceName() on the ServiceContext object.</td>
</tr>
<tr>
<td>Time and Date</td>
<td>The time-stamp generated by the system when the event is received.</td>
</tr>
<tr>
<td>Additional Information</td>
<td>Any other pertinent information about the event.</td>
</tr>
</tbody>
</table>

The method `logEvent()` is used to report an event, and has the following signature:

```java
public void logEvent(byte[] source, int identifier, int level, byte[] info)
```

The following code shows an how to set up and use the Event Log Service. It takes the form of a utility class that uses the Service Context to retrieve a handle to the Event Log Service. The method `storeEvent()` logs a simple event that will be written to the system event log table in the database.

**Listing 4-27  Event.java**

```java
package example.helloslee;

import com.incomit.slee.*;
```
import com.incomit.slee.event.*;

public class Event {

    private ServiceContext itsServiceContext;
    private EventLogService itsEventService;

    private int eventNumber = 42;
    private String message = "Logged message";

    public Event (ServiceContext aServiceContext) {
        itsServiceContext = aServiceContext;
        SLEECtx sc = itsServiceContext.getSLEECtx();
        itsEventService = sc.getEventLogService();
    }

    public void storeEvent () {
        itsEventService.logEvent(itsServiceContext
            .getInstanceName().getBytes(),
            EventLogService.MEDIUM,
            eventNumber,
            message.getBytes());
    }
}

Using the Event Data Record Service

The Event Data (EDR) Service provides an alternate method of dealing with event data. Rather than sending event data to the event log, it distributes that data to registered event listeners. The EDR Service also provides a filtering mechanism that can be applied to any registered listeners. This section describes how to distribute event information to the EDR service and how to receive EDR information with a listener. For information on filtering event information, refer to the Network Gatekeeper System Administrator’s Guide, in the section “Managing Event Data Records.”

EDRs should be sent whenever an important event occurs in the traffic interfaces of the traffic path. It is up to the designer of the service to decide when to send an EDR and what data should be included.

For application-initiated traffic, EDRs should be sent when a traffic request, or its response (in the case of an asynchronous implementation), reaches the Web Services, SESPA and ESPA layer, and when the request is handed off to the network in the plug-in layer. For network initiated traffic, EDRs should be sent when a traffic request, or its response (in the case of an asynchronous implementation), reaches the plug-in layer, and when it leaves the ESPA, SESPA and Web Services layer. EDRs should also be generated in these modules when the response to a request is propagated through the layers. This makes it possible to measure latency times using EDR listeners.

The event information should, when available, include:

- Service Provider ID
- Application ID
- Address of the subscriber or the terminal that is associated with the request
- A unique EDR identifier provided by the SLEE service. This identifier is always provided.

The EDR service automatically adds the following to the EDR:

- Timestamp
- The name of the SLEE service from which the EDR originates
- The name of the SLEE from which the EDR originates

In addition to this information, request-specific data can be added to the EDR.
Creating an EDR

All EDR information is stored as name-value pairs. All values must be of the data type String. Several values can be stored for one name. This is achieved using the classes StringSetMap and StringSet in the package com.bea.wlcp.wlnc.common.

For each value that is to be provided in the EDR, instantiate a StringSet. Use the method add(String value) to merge the specified value into the StringSet, and put the named Stringset into the StringSetMap.

Listing 4-28  Creating and sending an EDR to the EDR service.

```java
long EDR_ID = 50000;
EdrInformation edr = new EdrInformation(EDR_ID);
StringSetMap map = new StringSetMap();
StringSet set1 = new StringSet();
set1.add("A");
set1.add("B");
map.put("myName1", set1);
StringSet set2 = new StringSet();
set2.add("C");
map.put("myName2", set2);
edr.setAdditionalProperties(map);
sc.getSLEEContext().getEdrService().logEdr(edr);
```

In the example above, an EDR is sent to the EDR service. The EDR contains two names: myName1 and myName2. myName1 is assigned two values, “A” and “B”. myName2 is assigned a single value “C”.

When the StringSetMap is populated, it is added to the object representing the EDR, using the method setAdditionalProperties in the class EdrInformation. This class is located in the package com.bea.wlcp.wlnc.edr. Finally, the EDR is sent to the EDR Service using the
method logEdr. The EDR service is fetched from the SLEE Context. In the example, sc contains a reference to the Service Context.

As mentioned earlier, if they are available, the Service Provider ID, Application ID and address information must be added to the EDR. For these entries, the following names should be used:

- ServiceProviderId for the Service Provider ID
- ApplicationId for the Application ID
- Address for the address of the subscriber or the terminal that is associated with the request.

In Listing 4-29 the Service Provider ID, Application ID, and a set of destination addresses are added to the EDR.

### Listing 4-29  Adding data to the EDR

```java
String serviceProviderId = "SP1";
String applicationID = "APP1";
String[] addresses = {"555111", "555222"};
StringSetMap map = new StringSetMap();
StringSet set1 = new StringSet();
set1.add(serviceProviderId);
map.put("ServiceProviderId", set1);
StringSet set2 = new StringSet();
set2.add(applicationId);
map.put("ApplicationId", set2);
StringSet set3 = new StringSet();
for (int i = 0; i < addresses.length; i++) {
    set3.add(addresses[i]);
}
map.put("Address", set3);
edr.setAdditionalProperties(map);
```
Receiving an EDR from the EDR service

When an EDR is received by the EDR service, filtering can be applied based both on the names (myName1 and myName2) and the values associated with the names before being distributed to a registered EDR listener.

An EDR listener is a separate process that receives the EDR over CORBA. This EDR listener implements the interface EdrListener defined in the IDL file SLEEedr.idl. This file is included in the Network Gatekeeper Integration package.

The EDR service propagates the EDR to the EDR listener via a call to the method public void handleEdrs(EventDataRecord[] edrBatch).

The EDR listener is registered in the EDR service via OAM. See the System Administrator’s Guide section “Managing Event Data Records”.

Several EDRs can be distributed to the EDR listener using the array edrBatch.

The standard data provided in the EDR can be fetched directly from edrBatch[i]. In this case:

- edrBatch[i].edrId contains the unique ID for the EDR
- edrBatch[i].timestamp contains the timestamp.
- edrBatch[i].sleeName contains the name of the SLEE from which the EDR originates.

All other parameters are fetched via the additionalProperties part of the EDR:

- edrBatch[i].additionalProperties[j].name contains the name part of the name-value pair that was added to the EDR.
- edrBatch[i].additionalProperties[j].value contains the value part of the name-value pair that was added to the EDR.

The sample EDR generated in “Creating and sending an EDR to the EDR service.” on page 4-44 contains three additionalProperties which can be retrieved in the listener:

- additionalProperties[j].name equals myName1
- additionalProperties[j].value equals “A”
- additionalProperties[j+1].name equals myName1
Using the Charging Service

- additionalProperties[j+1].value equals “B”.
- additionalProperties[j+2].name equals myName2
- additionalProperties[j+2].value equals “C”.

Note: The order of the name-value pairs is not guaranteed.

The sample EDR generated in “Adding data to the EDR” on page 4-45 contains four additionalProperties that can be retrieved in the listener:

- additionalProperties[j].name equals ServiceProviderId
- additionalProperties[j].value equals SP1
- additionalProperties[j+1].name equals ApplicationID
- additionalProperties[j+1].value equals APP1
- additionalProperties[j+2].name equals Address
- additionalProperties[j+2].value equals 555111
- additionalProperties[j+3].name equals Address
- additionalProperties[j+3].value equals 555222

This means that the all generically recommended parameters are fetched from the additionalProperties array, in the same way that request-specific parameters are.

Note: The order of the name-value pairs is not guaranteed.

Using the Charging Service

The charging service works much as the event service does, except that it stores a different kind of information; that is, it stores usage information that can be used in charging for the service. Each SLEEE service has a dedicated instance of the charging service, which is retrieved via the Service Context.

The signature of the logChargingInfo() method looks like this:

public void logChargingInfo(ChargingInfo info)

The ChargingInfo class is a container with fields and corresponding set-methods for all charging-specific data. Before applying the logChargingInfo() method, all mandatory parameters must be set. Parameters that are not set will cause a null value to be written to the Charging Log. The methods available for the ChargingInfo class are shown in the table below.
All fields representing time are of type `long` and are reported in milliseconds. A start or stop time is represented as the number of milliseconds since January 1, 1970 00:00:00.000 GMT.

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addAdditionalInfo</td>
<td>(String xmlTag, String xmlValue)</td>
<td>Adds user defined additional charging parameters. These parameters will be saved in the <code>additional_info</code> field of the charging database as XML elements. This method can be called more than once to add more parameters.</td>
</tr>
<tr>
<td>clear</td>
<td>()</td>
<td>Clears all data and restores the default values in the ChargingInfo instance</td>
</tr>
<tr>
<td>setAmountOfUsage</td>
<td>(long amountOfUsage)</td>
<td>Sets the used amount. Used when the charging is not time dependent, for example, flat rate services.</td>
</tr>
<tr>
<td>setCompletionStatus</td>
<td>(int status)</td>
<td>Sets transaction completion status. Indicates if the transaction was completed or not. If the transaction is divided into parts, the completion status also indicates if all transaction parts have been sent. The class defines constants that should be used: ChargingInfo.COMPLETION_STATUS_COMPLETED ChargingInfo.COMPLETION_STATUS_PARTIAL ChargingInfo.COMPLETION_STATUS_FAILED ChargingInfo.COMPLETION_STATUS_POLICY_REJECTED</td>
</tr>
<tr>
<td>setConnectTime</td>
<td>(long connectTime)</td>
<td>Sets a timestamp telling when the destination party responded</td>
</tr>
<tr>
<td>setDestinationParty</td>
<td>(String destinationParty)</td>
<td>Sets the destination party's address</td>
</tr>
</tbody>
</table>
The following example shows obtaining a reference to the charging service and creating a Charging Data Record (CDR) in the charging table in the database. The class is in the form of a helper; that is, it is created at the start of service usage for a specific subscriber. When the subscriber stops using the service, the `createChargingRecord()` method is called to produce the charging record. The class uses the `timeManager` utility service from the `com.incomit.time` package to retrieve the system time at start and stop of service usage.

**Listing 4-30  ChargingHelper.java**

```java
package example.helloslee;
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setDurationOfUsage</code></td>
<td><code>(long durationOfUsage)</code></td>
<td>Sets the total time the service used network resources</td>
</tr>
<tr>
<td><code>setEndOfUsage</code></td>
<td><code>(long endOfUsage)</code></td>
<td>Sets a timestamp telling when the service stopped using network resources</td>
</tr>
<tr>
<td><code>setOriginatingParty</code></td>
<td><code>(String originatingParty)</code></td>
<td>Sets the originating party's address</td>
</tr>
<tr>
<td><code>setServiceName</code></td>
<td><code>(String serviceName)</code></td>
<td>Sets the name of the used service</td>
</tr>
<tr>
<td><code>setSessionID</code></td>
<td><code>(long sessionId)</code></td>
<td>Sets the session ID. The session ID correlates between related charging transactions.</td>
</tr>
<tr>
<td><code>setTransactionPartNumber</code></td>
<td><code>(int transactionPartNumber)</code></td>
<td>Sets the transaction part number. Used if the transaction is divided into different parts. Increment the number by one for each transaction part.</td>
</tr>
<tr>
<td><code>setStartOfUsage</code></td>
<td><code>(long startOfUsage)</code></td>
<td>Sets a timestamp telling when the service started using network resources</td>
</tr>
<tr>
<td><code>setUserID</code></td>
<td><code>(String userId)</code></td>
<td>Sets the ID of the application that has used the service</td>
</tr>
</tbody>
</table>
The SLEE and Interacting with SLEE Services and Utilities

```java
import com.incomit.slee.*;
import com.incomit.slee.time.*;
import com.incomit.slee.charging.*;

public class ChargingHelper {

  private ServiceContext itsServiceContext = null;
  private ChargingService itsChargingService = null;
  private SLEETimeManager itsTimeManager = null;
  private ChargingInfo itsInfo;

  public ChargingHelper (ServiceContext aServiceContext,
                          int aSessionId,
                          String aUserId) {
    itsServiceContext = aServiceContext;
    itsChargingService = itsServiceContext.getChargingService();
    SLEEContext sc = itsServiceContext.getSLEEContext();
    itsTimeManager = sc.getTimeManager();
    itsInfo = itsChargingService.createChargingInfo();
    itsInfo.setSessionID(aSessionId);
    itsInfo.setServiceName(itsServiceContext.getName());
    itsInfo.setUserID(aUserId);
    itsInfo.setStartOfUsage(itsTimeManager.getTime());
    itsInfo.addAdditionalInfo("additionalinfo", "testvalue");
  }
```
public void createChargingRecord () {
    itsInfo.setEndOfUsage(itsTimeManager.getTime());
    itsInfo.setCompletionStatus(ChargingInfo.COMPLETION_STATUS_COMPLETED);
    itsInfo.addAdditionalInfo("moreinfo", "someothervalue");
    try {
        itsChargingService.logChargingInfo(itsInfo);
    } catch(ChargingException ce) {
        //Handle Exception
    }
}
implementation of the northbound interface. It is, however, possible to retrieve the parameters in any layer of the traffic path.

The tunneled parameters are stored as name-value pairs. All tunneled parameters must be of type String. The parameters are added to a StringSet, and each StringSet is put into a StringSetMap under a name you designate. The classes StringSetMap and StringSet are defined in the package \texttt{com.bea.wlcp.wlng.common}.

For each value that is to be tunneled, instantiate a StringSet. Use the method \texttt{add(String)} on the StringSet to add the parameter or parameters to the StringSet. Add the StringSet to the StringSetMap, and include a name to be associated with it, using the method \texttt{put()} on the StringSetMap.

The StringSetMap is added to a RequestLocal object using the method \texttt{setTunneledObject} on the RequestLocal object.

The RequestLocal object is fetched from the Cluster Context, which is fetched from the SLEE Context.

See Listing 4-31 below.

\begin{Verbatim}
Listing 4-31 Add tunneled parameter.

String TUNNELED\_DATA\_NAME = "wespa.tunneledData";
String tunneledData = "mydata";
StringSetMap tunneledParameters = new StringSetMap();
StringSet stringSet1 = new StringSet();
stringSet1.add(tunneledData);
tunneledParameters.put(TUNNELED\_DATA\_NAME, stringSet1);
m\_sc.getSLEEContext().getClusterContext().getRequestLocal().setTunneledObject(tunneledParameters);
\end{Verbatim}

To retrieve a tunneled parameter from the Request Context, use the method \texttt{getTunneledObject}. The tunneled data in the StringSet is retrieved by name. See example below.
Listing 4-32  Retrieve tunneled parameter.

```java
StringSetMap tunneledParameters = 
(StringSetMap)m_sc.getSLEEContext().getClusterContext().getRequestLocal().
getTunneledObject();
String tunneledData = null;
if(tunneledParameters != null) {
    StringSet set = tunneledParameters.get(TUNNELED_DATA_NAME);
    if (set != null) {
        Iterator it = set.iterator();
        if(it.hasNext()) {
            tunneledData = (String)it.next();
        }
    }
}

m_sc.getSLEEContext().getClusterContext().getRequestLocal().setTunneledObject(null);
```

**Using the Time Service**

The time service provides a mechanism for retrieving system time and setting up various timers. Timers can help the application programmer protect programs from hanging and avoid dead-lock situations. For example, when requesting a resource over a high latency network, the programmer can include a timer with a certain threshold value. If the network resource is unavailable, the timer expires, and the program is released to continue execution, taking any actions necessary to report the faulty resource, etc. If the resource is available, the timer is reset and execution continues normally.

Periodic timers can generate certain events at regular intervals. For example, an application can read the contents of a database table containing subscription numbers, and use the results to submit a group SMS.
You should use a new object reference for each timer to be used. If the same reference is used and you want to cancel a timer, any of the timers using the same object reference could be cancelled.

**Listing 4-33  Timer.java**

```java
package example.helloslee;

import com.incomit.slee.*;
import com.incomit.slee.time.*;

public class Timer {

    private ServiceContext itsServiceContext = null;
    private SLEETimeManager itsTimeManager = null;
    private String timerRef;

    Object synchObject = new Object();

    public Timer (ServiceContext aServiceContext,
        String reference)
    {
        itsServiceContext = aServiceContext;
        SLEEContext sc = itsServiceContext.getSLEEContext();
        itsTimeManager = sc.getTimeManager();
        timerRef = reference;
    }

    public void removeTimer () {
    }
}
public void startTimer () {
    itsTimeManager.scheduleTimer(false,
                               10 * 1000, // 10 seconds
                               new MyTimerListener(this),
                               timerRef);
}
}

Listing 4-34  Timer.java -method: MyTimerListener

class MyTimerListener implements SLEETimerListener {

    private Timer timer = null;

    MyTimerListener(Timer timer) {
        this.timer = timer;
    }

    public void processTimer(Object reference) {
        synchronized (timer.synchObject) {
            timer.synchObject.notify();
        }
    }
}

itsTimeManager.cancelTimer(timerRef);

}
package example.helloslee;

import com.incomit.slee.*;
import com.incomit.slee.time.*;
import com.incomit.slee.event.*;

public class PeriodicTimer {

    private ServiceContext itsServiceContext = null;
    private SLEETimeManager itsTimeManager = null;
    private EventLogService itsEventService;

    private String timerRef = "periodic timer";

    long PERIOD = 2000;

    Object synchObject = new Object();

    public PeriodicTimer (ServiceContext aServiceContext) {
        itsServiceContext = aServiceContext;
        SLEEContext sc = itsServiceContext.getSLEEContext();
        itsEventService = sc.getEventLogService();
    }
}
Using the Time Service

```java
itsTimeManager = sc.getTimeManager();
}

public void doPeriodicTask (int cnt) {
    String message = "This is periodic task " + cnt;
    itsEventService.logEvent(itsServiceContext
        .getInstanceName().getBytes(),
        EventLogService.LOW,
        cnt,
        message.getBytes());
}

public void removeTimer () {
    itsTimeManager.cancelTimer(timerRef);
}

public void startTimer () {
    itsTimeManager.scheduleTimer(true,
        period,
        new MyTimerListener(this, 10),
        timerRef);
    try {
        synchronized (synchObject) {
            synchObject.wait();
        }
    } catch (java.lang.InterruptedException ie) {
        }
```
Listing 4-37   PeriodicTimer.java (continued)

public class MyTimerListener implements SLEEPTimerListener {

    private int counter = 0;
    private int noOfEvents;
    private PeriodicTimer timer;

    MyTimerListener(PeriodicTimer timer, int noOfEvents) {
        this.timer = timer;
        this.noOfEvents = noOfEvents;
    }

    public void processTimer(Object reference) {
        if (counter < noOfEvents) {
            timer.doPeriodicTask(counter);
            counter++;
        } else {
            synchronized (timer.synchObject) {
                timer.synchObject.notify();
                //do stuff
            }
        }
    }
}
Using the Trace Service

The trace service allows you to trace the execution path of a one or more parts of your service in order to help you understand issues that may be arising. The trace output is written to file. Because trace may be disabled in the running environment, for performance reasons, your service should always check for the trace active flag before calling the `logTrace` method. The file that is generated is specific to the service from which the data originated.

All services using the trace service must have an associated buffer class. This class buffers all trace messages from the service until the buffer is full, and then flushes the buffer to file.

The level and type of information that gets written to the trace log is controlled by using trace filter groups. It is possible to turn on and off different pre-defined trace groups at runtime by supplying a new filter value. The following filters are available:

<table>
<thead>
<tr>
<th>Filter</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>METHOD_IN</td>
<td>Log trace at entry of method.</td>
<td>1</td>
</tr>
<tr>
<td>METHOD_OUT</td>
<td>Log trace at exit of method.</td>
<td>2</td>
</tr>
<tr>
<td>USERDEF_1</td>
<td>User defined trace (DEBUG)</td>
<td>4</td>
</tr>
<tr>
<td>USERDEF_2</td>
<td>User defined trace (INFO)</td>
<td>8</td>
</tr>
<tr>
<td>USERDEF_3</td>
<td>User defined trace (WARNING)</td>
<td>16</td>
</tr>
<tr>
<td>USERDEF_4</td>
<td>User defined trace (ERROR)</td>
<td>32</td>
</tr>
<tr>
<td>USERDEF_5</td>
<td>User defined trace</td>
<td>64</td>
</tr>
</tbody>
</table>
To specify which groups to include in the trace log, the values for the different filters are added together. So, for example, if you want to include the entry and exit of a method, as well as information in user defined trace group 4, you would specify a filter value of 35:

$$1 + 2 + 32, \text{ or METHOD\_IN + METHOD\_OUT + USERDEF\_4}$$

If you wish to disable trace logging you should deactivate trace for that service rather than setting the filter value to 0. Even a filter value of 0 has a performance cost.

The code example below shows setting up a reference to the trace service and then defining a set of methods that use the trace methods available in the API. The methods, along with initializations, are executed from the `trace()` method, which is defined at the end of the example.

**Listing 4-38  Tracer.java**

```java
package example.helloslee;

import com.incomit.slee.*;
import com.incomit.slee.trace.*;
```
public class Tracer {

    private ServiceContext itsServiceContext;
    private TraceLogService itsTraceService;

    public Tracer (ServiceContext aServiceContext) {
        itsServiceContext = aServiceContext;
        itsTraceService = itsServiceContext.getTraceService();
    }

    The call to the ordinary Trace() method causes the trace to attempt a trace of method entry and exit. It uses the API methods logTraceIntoMethod() and logTraceOutOfMethod() with the following signatures:

    public void logTraceIntoMethod(java.lang.String className,
                                    java.lang.String methodName)

    public void logTraceOutOfMethod(java.lang.String className,
                                     java.lang.String methodName)

    The parameter className denotes the current, defining class, and methodName is the name of the method in which the statement is included. The calls to logTraceIntoMethod() and logTraceOutOfMethod() always cause its information to be included in its corresponding trace group.

    Listing 4-39 Tracer.java (continued) -Example of method in and method out trace

    public void methodInmethodOut () {
        if (itsTraceService.isGroupTraceActive(TraceLogService.METHOD_IN)) {
            itsTraceService.logTraceIntoMethod("Tracer",
                                             "methodInmethodOut");
    }
The SLEE and Interacting with SLEE Services and Utilities

```java
// Perform method-specific functionality
if (itsTraceService.isGroupTraceActive(TraceLogService.METHOD_OUT)) {
    itsTraceService.logTraceOutOfMethod("Tracer",
         "methodInmethodOut");
}
```

The call to `userdef1Trace()` causes the trace to attempt a trace on user defined level 1. It uses the API methods `logTrace()` with the following signature:

```java
public void logTrace(java.lang.String className, java.lang.String methodName, int traceGroup, java.lang.String info)
```

The information from `logTrace()` can be included in any trace group (the value of `traceGroup`). `logTrace()` also allows the inclusion of an arbitrary string in the generated trace log message.

**Listing 4-40   Tracer.java (continued) -Example of userdef 1 trace**

```java
public void userdef1Trace () {
    if (itsTraceService.isGroupTraceActive(TraceLogService.USERDEF_1)) {
        for (int i = 0; i < 10; i++) {
            itsTraceService.logTrace("Tracer",
               "userdef1Trace",
               TraceLogService.USERDEF_1,
               "Trace Log Record " + i);
        }
    }
}
```
The call to `traceFaultyMethod()` generates an arithmetic exception, which will be logged in the trace log. To facilitate the tracing of exceptions, the API offers the `logTraceException()` method. It has the following signature:

```java
public void logTraceException(java.lang.String className,
                              java.lang.String methodName,
                              int traceGroup,
                              java.lang.String info,
                              java.lang.Throwable exception)
```

The `className` and `methodName` are the same as they are for the `logTrace()` method. The trace log message will be included in the trace group corresponding to the value of the `traceGroup` parameter. Information about the exception, that is, the exception stack trace from the `exception` object, will also be included in the trace log message, along with an arbitrary string (`info`).

**Listing 4-41  Tracer.java (continued) -example of exception trace**

```java
public void traceFaultyMethod () {

    try {
        int a = 42;
        int b = 0;
        int c = a / b;
    } catch(ArithmeticException ae) {
        if (itsTraceService.isExceptionTraceActive()) {
            itsTraceService.logTraceException("Tracer",
```

Using the Trace Service
The traceRawData() method below shows how to trace any data in the form of a byte array. For this, the API offers the logTrace() method with the following signature:

public void logTrace(byte[] buf,
                     int off,
                     int len)

The buf parameter is a byte array, off indicates the starting position in this buffer, and len how many bytes from the buffer to be included in the trace log message. The call to logTrace below causes a byte array representing the string “rawdata 1 2 3” to be included in the trace log.

Listing 4-42  Tracer.java (continued) -example of using raw data trace

public void traceRawData () {
    if (itsTraceService.isGroupTraceActive(TraceLogService.RAW_DATA)) {
        String rawData = "Testing rawdata 1 2 3";
        itsTraceService.logTrace(rawData.getBytes(),
                                 8,
                                 rawData.length());
When exceptions occur, but alternative execution paths that nonetheless allow the request to be executed exist, the method logTraceException(...) should be used as in the example below. **itsTraceService** is the trace service.

**Listing 4-43  Example of use of logTraceException**

```java
if (itsTraceService.isExceptionTraceActive()) {
    m_ts.logTraceException(CLASSNAME, METHODNAME, TraceLogService.EXCEPTION_LOG, e.getMessage(), e);
}
```

To trace the traffic execution flow, the method **logTrafficFlowTrace(...)** should be used. It is only for traffic interfaces. A traffic flow context should be provided, with information on which Service Provider and application that initiated the request. In the example below, **m_trafficFlowContext** contains the Service Provider ID and application ID of the requester in the format **m_trafficFlowContext = applicationId + "\" + serviceProviderId;**

**Listing 4-44  Example of use of logTrafficFlowTrace**

```java
if (m_ts.isTrafficFlowTraceActive()) {
    Object[] params = {listener, data, address, serviceCode, requesterID};
```
Context Trace

To focus and/or expand the information produced by the trace service, Network Gatekeeper users can employ the context trace capability. Using the Network Gatekeeper Management Tool, an operator can add a context trace filter, which creates a single log file from all trace information matching particular criteria, regardless of which SLEE service produced the information. Context trace is used for acquiring information only for services that are directly in the traffic flow.

As a developer of one or more SLEE services, you must provide the mechanism on which the filters depend.

The information that may be used to create context trace filters consists of:

- Service name
- Service Provider ID
- Application ID
- Address(es)

Adding context trace information

Prior to calling any method to log trace on the Trace service, a `StringSetMap` is populated with context trace information. The context trace information is added as name-value pairs in a `StringSetMap` where the names are:

- `sp` for Service Provider ID
- `app` Application ID
- `addr` for the Address(es)
- `trans` for the Transaction ID
Listing 4-45  Adding addresses and transaction IDs to the context trace

RequestLocal rl =
sc.getSLEECtx().getClusterContext().getRequestLocal();
StringSetMap map = rl.getRequestContext();
if (map == null) {
    map = new StringSetMap();
    rl.setRequestContext(map);
}
if (addresses != null) {
    StringSet addrs = new StringSet();
    for (int i = 0; i < addresses.length; i++) {
        addrs.add(addresses[i]);
    }
    map.put("addr", addrs);
}
if (transactionId != null) {
    StringSet trans = new StringSet();
    trans.add(transactionId);
    map.put("trans", trans);
}

First, the request information is fetched from the Cluster Context. The Cluster Context contains context information for a clustered set of SLEEs rather than a single SLEE.

If the local request information does not exist, a StringSetMap is instantiated and added to the RequestLocal object using setRequestContext.

If there are addresses in the RequestLocal object, they are fetched using the StringSetMap get(...) method. The addresses are fetched by name, that is, get("addr"). If the name does
not yet exist in the map, use the put(...) method to insert it. Additional address values can added to the StringSet by using add(...) method.

If there is a transaction ID in the RequestLocal object, it is fetched using the get(...) method on the StringSetMap. The transaction ID is fetched by name, that is, get("trans"). If the name does not yet exist in the map, use the put(...) method to insert it. A transaction ID value is added to the StringSet by using the add(...) method.

Listing 4-46 Adding Service Provider ID and Application ID to the context trace

```java
if (serviceProviderId != null) {
    StringSet sp = map.get("sp");
    if (sp == null) {
        sp = new StringSet();
        map.put("sp", sp);
    }
    sp.add(serviceProviderId);
}
if (applicationId != null) {
    StringSet app = map.get("app");
    if (app == null) {
        app = new StringSet();
        map.put("app", app);
    }
    app.add(applicationId);
}
```

In some circumstances - for example, in the case of a network-triggered request as it is being processed in the network plug-in - the Service Provider ID and the Application ID might not be
known to the system. Except in such situations, however, they should be made available for context trace use.
The Traffic Path Framework

As you saw in The SLEE and Interacting with SLEE Services and Utilities, modules in WebLogic Network Gatekeeper interact with each other mostly by using CORBA-based calls in the context of the SLEE, Network Gatekeeper’s execution environment. This chapter adds to that model by explaining, at a fairly high level, the structures that enable the appropriate routing of request data flow through the Network Gatekeeper, using both CORBA and other based mechanisms. In other words, this chapter describes the framework that creates and supports the Traffic Path Map.

The description follows, roughly, the path of a generic application request as it enters the Network Gatekeeper through the Northbound Interface, is passed on to SESPA, moves through ESPA, is routed through the Plug-in Manager, and finally is forwarded on through the Network Plug-in to the underlying network element. In addition, it also covers the Service Capability Manager, which serves basically the same routing function as the Plug-in Manager, but for network-triggered traffic.

The sections are as follows:

- “Web Services layer to SESPA” on page 5-2
- “SESPA to ESPA” on page 5-7
- “ESPA to Plug-in Manager” on page 5-9
- “Plug-in Manager to Network Plug-in” on page 5-16
- “Network Plug-in to Service Capability Manager” on page 5-23
Web Services layer to SESPA

Most of the Northbound Interfaces to Network Gatekeeper are Web Services based. In such cases, the actual implementation of the interface is developed in the standard way and deployed in the provided Embedded Tomcat instance. The Tomcat instance takes care of unwrapping and parsing the Web Service request, but there is no immediate way for it to pass the data on to the next layer in the traffic path, the stateless adapter level or SESPA.

This problem is resolved using a mechanism called the SLEE Common Loader. The SLEE Common Loader provides a registration and lookup mechanism to allow the WS layer to access objects in the SESPA layer and the SESPA layer to access objects in the WS layer.

Registering the class

First a class is registered in the SLEE Common Loader.

Listing 5-1   Registering a class in the SLEE Common loader

```java
m_sc = serviceContext;
String[] ifClasses = {
    "com.incomit.sespa.myservicecapability.MyServiceCapability",
    "com.incomit.sespa.myservicecapability.MyServiceCapabilityListener"
};
try {
    for (int i=0; i<ifClasses.length; i++) {
        SleeCommonLoader.getInstance().registerClass(m_sc.getJarName(),
            ifClasses[i]);
    }
} catch (SleeCommonLoaderException ex) {
    ....
}
```
This SESPA implementation registers its interfaces classes `MyServiceCapability` and `MyServiceCapabilityListener` with the SLEE Common Loader. The class names are defined in a list, and registered in the SLEE Common Loader along with the name of the jar file in which they are found. In this case, the jar file is resolved using a method of the object `m_sc`, which is of type `ServiceContext`. See “The ServiceContext object and accessing SLEE utilities” on page 4-7 for a description of the Service Context.

Next an object implementing the registered interface is instantiated and added to the SLEE Common Loader. In the example below, this is an object implementing the interface given when registering the class

```java
com.incomit.sespa.myservicecapability.MyServiceCapability
```

The object is given a registration name which is used in the lookup procedure. In the example, the name assigned is `OBJ_SESPA_MY_SERVICE_CAPABILITY`. This name must be unique, and in general it is good practice to use the actual class name as a part of the assigned name.

```java
try {
    m_myScImpl = new MyServiceCapabilityImpl(m_sc, m_dbHelper);
} catch (Exception ex) {
    String errorMsg = "Failed to create MyServiceCapabilityImpl. Reason: "+ ex.getMessage();
    throw new ServiceDeploymentException(errorMsg,
    errorMsg,
    m_sc.getName(),
    0);
}

try {
    SleeCommonLoader.getInstance().addObject(OBJ_SESPA_MY_SERVICE_CAPABILITY,
    m_myScImpl.getProxy());
} catch (SleeCommonLoaderException ex) {
```

Listing 5-2   Add object to the SLEE common class loader, allowing WS layer to access the object.
You’ll note that a proxy, and not the object itself, is provided. The proxy arrangement is part of the high availability strategy for the system, and is further explained in the section “High availability” on page 6-1.

**Using the Common Loader to access a SESPA object**

In order for the WS layer to use a SESPA object, it must know the name under which the object is registered in the SLEE Common Loader. See Listing 5-2 for more information about registration names.

**Listing 5-3  Obtain a reference to the registered SESPA object**

```java
try {
    if (m_serviceCapabilities == null) {
        Object obj = SleeCommonLoader.getInstance().getObject(
            OBJ_SESPA_MY_SERVICE_CAPABILITY);
        m_serviceCapability =
            (com.incomit.sespa.myservicecapability.MyServiceCapability) obj;
    }
} catch (Throwable t) {
    ....
}
```

The SLEE Common Loader is queried for the SESPA object registered under the name `OBJ_SESPA_MY_SERVICE_CAPABILITY`. Note that the returned object is cast to the correct class. Since the SESPA may also need to use methods in the interface exposed by the WS layer, the WS layer must also register its interface classes and add the implementing objects into the SLEE Common loader.
Using the SESPA object

One of the key functions of SESPA is marrying the stateless Web Services request to the originating application’s Network Gatekeeper login session, which is started when the application first logs in. This is done by retrieving the login ticket that every WS request must provide in the header of the SOAP envelope in which the request is sent.

The login ticket is provided in the WSSE part of the SOAP header as illustrated below. The application login user ID is also provided in the header.

Listing 5-4  WSSE header in SOAP Header

```xml
      domain_user@app_domain_1.default_provider
    </wsse:Username>
      app:73183493944772289
    </wsse:Password>
  </wsse:UsernameToken>
</wsse:Security>
```

The login user ID is found in the `<wsse:Username>` element, and is given in the form

`<Application Instance group>@<Application Account ID>.<Service Provider ID>`

(For more information on these terms, see the “Managing Application Service Providers” chapter of the *Architectural Overview* in the Network Gatekeeper documentation set.) The login ticket itself is found in the `<wsse:Password>` element in the form `app:<login Ticket>`.

Using the SESPA object obtained through the SLEE Common Loader, the WS layer passes in the the login ticket as a parameter for each request it makes. See the example in *Listing 5-5* below.
The SOAPHeaderHandler used to create m_SOAPHeaderHandler in the example is a utility class found in com.bea.wespa.util.SOAPHeaderHandler.

Listing 5-5  Calling a SESPA method

```java
returnValue = m_serviceCapability.myMethod(
    m_SOAPHeaderHandler.getCurrentSessionTicket(),
    endpoint,
    data,
    address,
    serviceCode,
    requesterID);
```

Removing an object from the SLEE Common Loader

When the SLEE calls deactivate(...) on the service, objects should be removed from the SLEE Common Loader. All resources should be cleaned up, and the affected applications should be notified. Listing 5-6 provides an example of removing an object from the SLEE Common Loader.

Listing 5-6  Removing an object from the SLEE Common Loader

```java
try {
    SleeCommonLoader.getInstance().removeObject(OBJ_SESPA_MY_SERVICE_CAPABILITY);
} catch (SleeCommonLoaderException ex) {
    String errorMsg = "Failed to deregister SESPA MyServiceCapability service from SleeCommonLoader. Reason: " + ex.getMessage();
    throw new ServiceDeploymentException(errorMsg, errorMsg,
```
Once the SESPA layer has the login ticket from the WS layer, it can fetch an ESPAManager object from ESPA. SESPA uses the ESPAManager object to invoke methods on ESPA. In general this process takes three steps:

1. The SESPA SC logs in to ESPA Access.
2. The SESPA SC gets an application session that is correlated with the login ticket.
3. The SESPA SC invokes methods on the ESPA SC using the ESPA Manager.

There are utility classes in SESPA which assist in retrieving the session object and the Manager object based on the login ticket. Listing 5-7 is an example of retrieving the application session and a Manager.

**Listing 5-7  Getting an application session and a manager**

```java
ApplicationSession appSession = m_dbHelper.getApplicationSession(loginTicket);
MyServiceCapabilityManager espaManager = getEspaManager(loginTicket);
result = espaManager.myMethodWait(data,
    address,
    waitTimeoutSeconds,
    serviceCode,
    requesterID,
    appSession);
```
First, the application session is fetched from the database where the correlation between the login ticket and the application session object is stored. This is done using the SESPA utility class `com.incomit.sespa.util.DbHelper.m_dbHelper` is an object instantiated from this class.

Then the ESPA Manager for the SC is fetched based on the login ticket. The ESPA Manager represents the ESPA SC, and the methods implemented in the ESPA SC can be invoked on this object as illustrated in Listing 5-7. A more detailed look at `getEspaManager(…)` is shown in Listing 5-8.

```
Listing 5-8 getEspaManager(…)

public MyServiceCapabilityManager getEspaManager(String loginTicket)
    throws GeneralException{
    MyServiceCapabilityManager manager = null;
    try {
        ESPAManager espaManager = null;
        espaManager = m_dbHelper.getEspaManager(loginTicket,
            MyServiceCapabilityManager.SERVICE_NAME);
        manager = MyServiceCapabilityManagerHelper.narrow(espaManager);
    } catch(AccessException ex) {
        ...
    }
    return manager;
}
```

This code fragment shows the ESPA manager being fetched based on the login ticket and a Service Name, `MyServiceCapabilityManager.SERVICE_NAME`. This Service Name is defined in the IDL file for the ESPA SC interface.

The manager is then narrowed to the actual class.
ESPA to Plug-in Manager

Once ESPA, using the Policy Service, has decided that a particular request has been accepted, it must locate a network plug-in that is appropriate for passing the request on to the underlying network element. To do this it uses the Plug-in Manager.

Accessing the Plug-in Manager

The Plug-in Manager (accessed as a SLEEResourceManager) serves a routing and load balancing function. It maintains a registry which lists the characteristics of all available network plug-ins. It evaluates the information ESPA passes into it and determines the best available plug-in for the task. This process is called “scheduling a resource task” and is illustrated in the sequence diagrams in “Traffic Path Flow” on page 3-1.

Using the SLEE Context, ESPA retrieves the Plug-in Manager and passes it the information it needs to determine the correct network plug-in (type, priority, etc.) along with a SLEE Resource Task object which implements the request that is to be forwarded to the plug-in in a separate thread.

In Listing 5-9 below this request-implementing SLEEResourceTask object is named task. The task object implements the method dotask(...), and when the Plug-in Manager has retrieved a suitable plug-in, it invokes the method doTask(...) on the object. The plug-in is provided as an argument to doTask(...). See MyServiceCapabilityManager_impl.java in \module_templates\espa_sc_module_impl\src\com\incomit\espa\my_espa_sc\ for a detailed example.

Listing 5-9  Example of getting a matching plug-in and scheduling a task

```java
MyServiceCapabilityContext.getServiceContext().getSLEEContext().getResourceManager().scheduleResourceTask(new TrProperty[0],
   MyServiceCapabilityContext.PLUGIN_TYPE,
   rAddress,
   resourceContext,
   0, // prio
   MyServiceCapabilityContext.POLICY_SERVICE_GROUP,
   1,
```
Plug-in Manager interfaces

The Plug-in Manager interacts using several interfaces. The main three are described in Figure 5-1.

An additional interface supports the callback Resource Listener implemented by an ESPA SC that is notified by the Plug-in Manager when new plug-ins have registered or de-registered. These interfaces support the functionality described in Table 5-1 and presented in fuller detail below.
SLEEResourceManager

Table 5-1 Plug-in manager interfaces

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLEEResourceManager</td>
<td>Initial object implemented by the Plug-in Manager.</td>
</tr>
<tr>
<td>Resource</td>
<td>Base interface that all plug-ins must implement.</td>
</tr>
<tr>
<td>ResourceListener</td>
<td>Call back interface used to notify that a plug-in has registered or de-registered itself in the Plug-in Manager</td>
</tr>
<tr>
<td>SCS</td>
<td>Implemented by the ESPA SCs. More detailed descriptions can be found in Network Plug-in to Service Capability Manager.</td>
</tr>
</tbody>
</table>

This interface provides ESPA with access to the Plug-in Manager.

Table 5-2 ResourceMgr

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addListener</td>
<td>Adds a listener, interested in knowing if plug-ins have been added or removed.</td>
</tr>
<tr>
<td>removeListener</td>
<td>Removes a registered listener.</td>
</tr>
<tr>
<td>getBestResource</td>
<td>Use scheduleResourceTask instead. Gets the resource which handles the specified type and address plan that has experienced the lowest load level and/or has been idle the longest time.</td>
</tr>
<tr>
<td>getResourceFromProperties</td>
<td>Use scheduleResourceTask instead. Performs the same tasks as getBestResource, but returns only resources that match specified properties.</td>
</tr>
<tr>
<td>getResource</td>
<td>Use scheduleResourceTask instead. Gets the appropriate resource having least load level which has been idle the longest time.</td>
</tr>
<tr>
<td>getResourceCtx</td>
<td>Gets a plug-in.</td>
</tr>
<tr>
<td>getResourceCtxSendList</td>
<td>Gets a plug-in capable of handling sendlists.</td>
</tr>
</tbody>
</table>
Resource

This is the base interface that all plug-ins must implement to interact with the Plug-in Manager. All CORBA objects that implement this interface must be persistent, that is, they must always use the same IOR, so that the plug-in routing information as configured in the Plug-in Manager remains correct. In the individual plug-in this interface is extended by aspects of the plug-in that are Service Capability-specific. There is one extension per SC type, such as call control, messaging, user location and so on.

Table 5-2 Resource

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scheduleResourceTask</td>
<td>Gets a plug-in and schedules a task to be performed. This is an asynchronous method for scheduling a resource task for sending a request to a plug-in matching the specified criteria. If a plug-in can be allocated for the request the supplied resource task will be executed in a separate thread.</td>
</tr>
<tr>
<td>registerResourceProperties</td>
<td>Registers properties for a plug-in.</td>
</tr>
<tr>
<td>registerResource</td>
<td>Registers a plug-in in the plug-in manager.</td>
</tr>
<tr>
<td>unregisterResource</td>
<td>Unregisters a plug-in.</td>
</tr>
<tr>
<td>getResourceNodeId</td>
<td>Gets the node ID for a plug-in.</td>
</tr>
</tbody>
</table>

Table 5-3 Resource

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getAddressPlan</td>
<td>Gets the supported address plans, such as, for example, E_164. A plug-in can support several address plans.</td>
</tr>
<tr>
<td>getLoadLevel</td>
<td>Gets the load level.</td>
</tr>
<tr>
<td>getLoadValue</td>
<td>Gets the load value in percent.</td>
</tr>
<tr>
<td>getType</td>
<td>Gets the type of plug-in, for example, call control or messaging.</td>
</tr>
<tr>
<td>getSubSystemLoadLevel</td>
<td>Gets the load level of the underlying system for this resource.</td>
</tr>
<tr>
<td>getSubSystemLoadValue</td>
<td>Gets the load value of the underlying system for this resource.</td>
</tr>
<tr>
<td>isActive</td>
<td>Checks to see if the plug-in is active or not.</td>
</tr>
</tbody>
</table>
**ResourceListener**

Individual ESPA SCs can implement this interface to be notified when new plug-ins are registered or de-registered in the Plug-in Manager.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resourcesUpdated</td>
<td>Notifies the ESPA SC that a resource has been registered or de-registered.</td>
</tr>
</tbody>
</table>

**Registration and De-registration**

All network plug-ins must register with the Plug-in Manager. The plug-in must implement the Resource interface so that it can be queried by the Plug-in Manager after it is registered. If a plug-in supports more than one type it must repeat this process for each type.
Figure 5-2  Registration and de-registration of a plug-in

Note: If the plug-in returns false when the `isActive` method is invoked, it will not be accessible until it is activated.

**Typical Traffic Path for Application-Initiated Requests**

The following sequence diagram shows how an application-initiated request - for example, create a call - is handled. The Plug-in Manager maintains routing information for each of the plug-ins, including a list of the address ranges a particular plug-in supports.

Note: The list is specified using the WebLogic Network Gatekeeper Management Tool.
1. The ESPA SC asks the Plug-in Manager (through the SLEEResourceManager interface) to schedule a task. This request contains the plug-in type, the address plan, the required address, a set of properties, and a SLEEResourceTask object on which the SLEE calls doTask(...) when a suitable plug-in is determined. From this information the Plug-in Manager will locate all active plug-ins whose internal routing information (types, properties, address plans and ranges) matches the request. Properties is a name-value pair array used to correlate the properties that the SC needs with those that the plug-ins supports. The following properties are used in the standard product:
   - NOTIFICATION_SUPPORT: Are notifications supported by the plug-in? Possible values- TRUE or FALSE.
   - SUBTYPE: What transport does Messaging User Interaction use? Possible values - USSD or SMS.

2. The Plug-in Manager checks to see if the plug-in is still active.

3. The Plug-in Manager checks the plug-in’s load level. If the plug-in reports that it is Overloaded, no new requests are sent to that plug-in.
4. The SLEE calls `doTask(...)` on the `SLEEResourceTask` object provided with the request.

5. The information is sent to the plug-in.

**Plug-in Manager to Network Plug-in**

The network plug-in must be able to interact with both the ESPA Service Capability that it supports and the underlying network element with which it communicates. Because the implementations making up the underlying network elements are often either proprietary or understand the supported protocol standards in quite specific ways, creating new and/or extending existing network plug-ins to support the needs of a particular installation is the most common use of the Extension SDK.

**Service Capability Interaction**

On its northbound side, the network plug-in implements the plug-in interface for a Service Capability (an ESPA SC). There are established plug-in interfaces for the following service capabilities:

- Call Control
- Charging (charging based on content)
- Messaging
- Message Sender
- User Location
- User Status
- Subscriber Profile
- User Interaction (call and message based)

If a new service capability is introduced, it must define a plug-in interface specific to itself.

A plug-in has a set of properties:

- `PLUGIN_TYPE`: Defines the service capability with which it is associated. This property is specified when the plug-in registers itself with the Plug-in Manager. The Plug-in Manager uses this type identifier when retrieving plug-ins for handling service requests. The type must match one of the allowed types in the Plug-in Manager service. Custom types are allowed, but must be registered in the Plug-in Manager.
• **SUBTYPE**: Defines a sub-type of an interface where relevant. For example, a messaging user interaction sub-type might be GUI, USSD, or SMS. The subtype designation indicates a particular extension of the a Resource interface type. Note that one Resource (plug-in) or one ESPA SC may support multiple subtypes.

• **TrAddressPlan**: Defines the type of address plans the plug-in supports. Examples include IP or E.164.

The Plug-in Manager uses these properties to assign the appropriate plug-in to the ESPA scheduleResourceTask request.

**Note**: Additional parameters, such as routing criteria and policy based routing settings are also taken into consideration, but these parameters are configurable in contrast to the properties listed above.

Below is a table outlining the properties.

### Table 5-5 Service Capabilities, plug-in types and plug-in subtypes

<table>
<thead>
<tr>
<th>Service capability</th>
<th>Plug-in to be used by the Service capability</th>
<th>Subtype of plug-in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call Control</td>
<td>CALL_CONTROL_TYPE</td>
<td></td>
</tr>
<tr>
<td>Call user interaction</td>
<td>USER_INTERACTION_CALL_TYPE</td>
<td></td>
</tr>
<tr>
<td>Charging (charging based on content)</td>
<td>CHARGING_TYPE</td>
<td></td>
</tr>
<tr>
<td>Messaging</td>
<td>MESSAGING_TYPE (for SMS)</td>
<td>SMS</td>
</tr>
<tr>
<td></td>
<td>MMS_TYPE (for MMS)</td>
<td>MMS</td>
</tr>
<tr>
<td>Messaging user interaction</td>
<td>USER_INTERACTION_TYPE</td>
<td>SMS -when SMS is the bearer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>USSD -when USSD is the bearer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GUI -when OSA Generic User interaction is the bearer</td>
</tr>
<tr>
<td>User location</td>
<td>USER_LOCATION_TYPE</td>
<td></td>
</tr>
<tr>
<td>User status</td>
<td>USER_STATUS_TYPE</td>
<td></td>
</tr>
<tr>
<td>Subscriber profile</td>
<td>SUBSCRIBER_PROFILE_TYPE</td>
<td></td>
</tr>
</tbody>
</table>
All plug-ins extend the base module `com.incomit.resources`, defined in `resource_common.idl`.

Resource code modules are mapped to service capabilities as follows.

<table>
<thead>
<tr>
<th>Module</th>
<th>Definition in</th>
<th>Corresponding Service capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.incomit.resources.callcontrol</td>
<td>CallControlResource_data.idl CallControlResource_IF.idl</td>
<td>Call Control</td>
</tr>
<tr>
<td>com.incomit.resources.charging</td>
<td>ChargingResource_data.idl ChargingResource_IF.idl</td>
<td>Charging</td>
</tr>
<tr>
<td>com.incomit.resources.messaging</td>
<td>messaging_mms_resource_if.idl messaging_resource_data.idl messaging_resource_if.idl</td>
<td>Messaging</td>
</tr>
<tr>
<td>com.incomit.resources.mm</td>
<td>MobilityResource_data.idl</td>
<td>User Location and User Status</td>
</tr>
<tr>
<td>com.incomit.resources.mm.ul</td>
<td>URIResource_data.idl URIResource_IF.idl</td>
<td>User Location</td>
</tr>
<tr>
<td>com.incomit.resources.mm.us</td>
<td>UsResource_data.idl UsResource_IF.idl</td>
<td>User Status</td>
</tr>
<tr>
<td>com.incomit.resources.sp</td>
<td>sp_data.idl sp_interfaces.idl</td>
<td>Subscriber Profile</td>
</tr>
<tr>
<td>com.incomit.resources.ui</td>
<td>UserInteractionResource_data.idl UserInteractionCallResource_IF.idl</td>
<td>Call User Interaction and Messaging User Interaction</td>
</tr>
<tr>
<td>com.incomit.resources.ui</td>
<td>UserInteractionResource_IF.idl</td>
<td>Call User Interaction</td>
</tr>
<tr>
<td>com.incomit.resources.ui</td>
<td>UserInteractionResource_IF.idl</td>
<td>Messaging User Interaction</td>
</tr>
</tbody>
</table>

All modules contain a set of interfaces to be implemented or used by the plug-in. Refer to the IDL definitions for the plug-in interfaces for detailed information. These files are located in `bea\wing22\esdk\idl\plugin_if`.
Plug-in States

Because communication with network elements relies on factors outside the control of the Network Gatekeeper, network plug-ins must be able to tell the system they are currently able to send information on to the network. This information is called the plug-in’s state. Figure 5-4 below shows possible plug-in states as defined in the Plug-in Manager.

Once a plug-in (called a Resource in the diagram) has registered itself (the types, subtypes, and address plans it supports) with the Plug-in Manager, it can report two states, Active and Inactive. A reference to the plug-in can be returned as a result of a scheduleResourceTask request from an ESPA SC only when the plug-in is in the Active state. The plug-in sets itself to Inactive when it loses contact with the underlying network element.

ESPA SCs can, however, continue to use existing references (for example call objects) to complete the processing of any active traffic. In addition, plug-in state is only pertinent for application-initiated requests. Network-triggered events (which can only occur if the plug-in is in contact with the network element) do not check the plug-in’s state.

Figure 5-4  Plug-in states

When the plug-in changes state from active to inactive, it is not required to inform the Plug-in Manager until the Manager queries it in response to a request from an ESPA SC, at which point it returns a false to the isActive query. When the plug-in moves from inactive to active, however, it must explicitly invoke the resourceisActive method as well as making sure it returns true to the Plug-in Manager’s query.
Suspending a plug-in

In order to support, for example, a graceful plug-in shutdown, it is possible to suspend it. This means that no new traffic will be sent to the plug-in. Suspending a call control plug-in, for example, would mean that no new calls can be created, but all active calls will work as normal.

Plug-ins that wish to implement a suspend option must implement the ServiceDeploymentExt interface. A suspended plug-in simply returns false when the isActive method is invoked. To resume from suspended state the plug-in calls resourceIsActive in the PluginRegistration interface.

Using plug-in states for high availability

It is also possible to use plug-in states for the purposes of performing a HA switch when using an active-standby system. See details about this in section High availability.

Best practices for the plug-in interfaces

This section contains details all plug-in interfaces have in common. Specific details for each service type (call control, user interaction, etc.) are described in separate chapters. See “A Closer Look at One Layer: the Network Plug-in” on page 7-1 for more information.

The session id parameter

To reduce the number of active CORBA objects, a plug-in may use the session id parameter. For example, when creating a new call (either network-triggered or application-initiated) the plug-in can send a session id that is associated with the call. Whenever the ESPA SC proxy communicates with the plug-in it sends this ID, which means that only one CORBA instance of the call object is required. But for this to work, the plug-in must have some internal method of dispatching invocations. Other uses of the session id might be setting up call legs of user interaction calls. It is up to the plug-in developer to decide whether the session id mechanism or the creation of multiple objects is to be used.

Note: In either case, the callback interfaces that the ESPA SC implements are always newly created CORBA objects.

As with all layer intercommunication, you should always use timers (see “Using the Time Service” on page 4-53 for more information) to track responses and detect and cleanup resources when no response is received.
Threading

Each request sent from the ESPA SC to a network plug-in using the SLEEResourceTask object should occur on its own thread. This will allow the CORBA thread to return as quickly as possible, reducing the time the CORBA threads are blocked in the ESPA SC proxy. Because most method calls are asynchronous (that is they have a request and a separate response method) this is not usually an issue.

The ESPA SCs use a thread pool to implement this functionality, using the SLEE task manager as described in “SLEE Task Manager” on page 4-14 and “SLEE Task” on page 4-14.

Helper classes for network plug-ins

Plug-ins are SLEE services and must implement the interfaces required to interact with the SLEE. To help the plug-in developer with the implementation of the SLEE services parts of the network plug-in, Network Gatekeeper provides a set of service helper classes. The template for the SLEE service part of the plug-in templates provided in the Extension SDK module uses these classes.

The helper classes aid developers in implementing the Service Deployable and Service Accessible interfaces by doing some of the basic work for them.

In the package com.bea.wlcp.wlng.esdk there are two general helper classes for SLEE services.

- com.bea.wlcp.wlng.esdk.SLEEService: An abstract helper class that implements the ServiceDeployableExt interface. This class can serve a base class for general services executing in the SLEE.

- com.bea.wlcp.wlng.esdk.Context: An abstract helper class that provides methods for getting and setting the POA and for getting the Service Context.

An additional helper class is specific to network plug-ins -

- com.bea.wlcp.wlng.esdk.plugin.PluginSleeService: An abstract base class that implements the ServiceAccessible interface, and implicitly through the SLEEService class, the ServiceDeployableExt interface. See Figure 5-5 for more information.
**Figure 5-5** SleeService and PluginSleeService inheritance structure

PluginSleeService handles the registration of the plug-in with the Plug-in Manager. It extends the abstract class `com.bea.wlcp.wlng.esdk.SleeService`. There is also an abstract class, `com.bea.wlcp.wlng.esdk.plugin.PluginContext`, that provides methods for getting and setting the `resourceID`, as well as declaring methods that should be implemented in the sub-context class. Plug-in developers should extend this PluginSleeService class rather than creating their own implementation of the `ServiceAccessible` and `ServiceDeployableExt` interfaces.

One benefit of extending the PluginSleeService class for your implementation class is that it will then handle service change of states in the appropriate way:

- To change to `Started` is handled by the method `doStarted(...)`. 

• The change to Activated is handled by the method `doActivated(...)`.  
  `PluginSleeService` manages registering the plug-in with the Plug-in Manager.

• The change to Deactivated (blocked from handling further CORBA requests) is handled by 
  the method `doDeactivated(...)`

For more information on SLEE service states, see “SLEE service states” on page 4-3

Traffic interfaces (the network element facing interfaces) for plug-ins are implemented in a 
separate class. Your class that extends the `PluginSleeService` helper class holds a 
`TrafficObject` as a member variable. If the method `getTrafficObject()` is called before the 
traffic interface object exists, it is automatically created.

**Network Plug-in to Service Capability Manager**

The Service Capability Manager provides the same sort of service for traffic that originates in the 
network as the Plug-in Manager provides for application-initiated traffic. Its main responsibility 
is delivering references for the appropriate ESPA SC to the plug-ins. ESPA SCs register the sort 
of functionality they support with the Manager, and the Manager parcels out references to them 
in response to requests from the network plug-in. Any ESPA SC that wants to be accessed by 
plug-ins must implement the SC base interface.

In installations where several SLEEs are running, each instance has its own SC Manager. In this 
case it does not matter which SC Manager is used, as they all share the same information.

**Note:** In some situations it may be better not to use the Service Capability Manager at all and 
plug-ins are not required to do so, although the plug-in developers need to do additional 
work to support this. Some plug-ins operate outside of the SLEE environment, and using 
alternate connection methods may produce better performance, as no extra CORBA 
invocations are needed. In this case, the ESPA SC registers callbacks directly with the 
plug-in and the plug-in must be prepared to deal with receiving an overload exception 
from one ESPA SC by trying another ESPA SC.

**Accessing the SC Manager**

The Service Capability Manager is retrieved from the SLEEContext. The SLEE Context is 
fetched from the Service Context, provided by the SLEE via the `ServiceAccessible` interface. 
For more on this process, see “The ServiceContext object and accessing SLEE utilities” on 
page 4-7.
The Traffic Path Framework

Listing 5-10  Getting the SC manager

m_scsManager = m_sc.getSLEEContext().getSCSManager();

Here, m_sc is the ServiceContext. In Listing 5-11 a list of all registered listeners of MESSAGING_TYPE is requested. To fetch other types of ESPA SCs, only the type identifier would change.

Listing 5-11  Getting a list of ESPA Messaging SCs

scsArray = m_scsManager.getSLEESCSDiscovery().getSCSCtx(capabilityProperties,

SCS.MESSAGING_TYPE,

m_eventAny,

m_resourceID);

In general, the first ESPA SC on the list should be used, as the Service Capability Manager also provides load balancing. Nonetheless, it is possible to use any SC on the list.

**Interfaces**

The Service Capability interacts with the rest of the system using the interfaces detailed in Figure 5-6.
These interfaces support the functionality described in Table 5-7 and are presented in fuller detail below.

**Table 5-7 SC manager interfaces**

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLEESCSManager</td>
<td>Initial object implemented by the SC Manager.</td>
</tr>
<tr>
<td>SCSDiscovery</td>
<td>Interface that plug-ins use to obtain SC references.</td>
</tr>
<tr>
<td>SCSRegistration</td>
<td>Interface used to register an SC.</td>
</tr>
<tr>
<td>SLEESCSDiscoveryOperations</td>
<td>Interface implemented by the SCs’ proxies.</td>
</tr>
<tr>
<td>SCSRegistrationOperations</td>
<td>The base interface that all plug-ins must implement.</td>
</tr>
</tbody>
</table>
SLEESCSManager

This interface is the initial object in the SC Manager. It is used to access other SCS Manager instances and to retrieve the registration and discovery interfaces.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getSCSRegistration</td>
<td>Retrieves the SCSRegistration object.</td>
</tr>
<tr>
<td>getSCSDiscovery</td>
<td>Retrieves the SCSDiscovery object.</td>
</tr>
</tbody>
</table>

SLEESCSRegistrationOperations

This interface is used by an ESPA SC to register itself with the SC manager.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>registerSCS</td>
<td>Registers an SC. This method is to be used when only the type of the SC has significance.</td>
</tr>
<tr>
<td>registerSCSWithProperties</td>
<td>Registers an SC using a set of properties. A property of type INSTANCE_ID with a value of the returned SCS ID is always appended to the properties.</td>
</tr>
<tr>
<td>unregisterSCS</td>
<td>Unregisters an SC.</td>
</tr>
<tr>
<td>SCSIsActive</td>
<td>Called by SC to indicate that it is active.</td>
</tr>
</tbody>
</table>

SLEESCSDiscoveryOperations

This interface is used by plug-ins to retrieve references to SC instances. Using the SCs retrieved with this interface in the order presented enforces load balancing in network-initiated traffic.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getSCS</td>
<td>Deprecated</td>
</tr>
<tr>
<td>getSCSCtx</td>
<td>Retrieves all SCs that have enabled a criteria matching the supplied parameters.</td>
</tr>
</tbody>
</table>
Network-triggered events and the SC Manager

The following sequence shows a network plug-in passing a network-triggered event to an SC that has been retrieved using the SC Manager.

**Figure 5-7 Figure Network-triggered event**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getSCSFromEvent</td>
<td>Deprecated</td>
</tr>
<tr>
<td>getSCSFromProperties</td>
<td>Deprecated</td>
</tr>
</tbody>
</table>

Table 5-10 SSLEECSDiscoveryOperations
The following sections contain descriptions of high-availability aspects for extensions to WebLogic Network Gatekeeper:

- Introduction
- High Availability Between Network Plug-ins and ESPA SCs
- High Availability between SESPA and ESPA

**Introduction**

WebLogic Network Gatekeeper operates in an environment in which robustness and high availability are of utmost importance. As a result, Network Gatekeeper is normally run in a cluster of servers that run in parallel. (For more information on hardware/software configurations, see the “Hardware Architecture” and “Software Architecture Overview” chapters of the Architectural Overview). Some load balancing and high availability functionality is handled by hardware components, but aspects of Network Gatekeeper’s internals also provide support for these important characteristics.

Each server by layer runs exactly the same modules that all other servers of its type runs, and each module contains exactly the same information, and is structurally interchangeable with its fellows, although at any point each instance holds different active sessions. All layers that run ESPA SCs also run Plug-in Managers and Service Capability Managers, each of which is synchronized with all the others and is treated as equivalent by the system. By evaluating plug-in load and status as part of the resource scheduling task, the Plug-in Manager plays a key role in
maintaining high availability for application-initiated traffic and the Service Capability Manager, evaluating various ESPA modules, plays the same role for network-triggered traffic.

As well, a proxy-based mechanism, facilitated by the SLEE Common Loader, provides high availability for the connection between the SESPA and ESPA layers.

**High Availability Between Network Plug-ins and ESPA SCs**

The Plug-in Manager and the Service Capability Manager play a key role in maintaining high availability between the network plug-in layer and the ESPA SC layer. Each server containing these layers also contains an instance of both managers. All instances share all the same information, so that it makes no difference which one is used. Once a reference to one instance is obtained, references to all other instances can be acquired, using the `getAllResourceManagers` method in the `ResourceManager` interface and `getAllSCSManagers` method in the `SCSManager` interface. If desired, an external plug-in can poll the Plug-in Manager and the SC Manager at regular intervals to see if additional managers have started.

**Note:** There is no session level redundancy at this level, so all recovery and distribution mechanisms apply only to newly created sessions. If one server crashes, traffic should continue to flow uninterrupted, but all active sessions on the faulty machine will be lost.

**Application-Initiated**

The application-initiated data path is the simpler of the two situations, because the Plug-in Manager is always used to deliver the network plug-in to the ESPA SC. At the moment the ESPA SC calls `scheduleResourceTask`, the Plug-in Manager queries the appropriate plug-in(s) to make sure it is available. If the Manager detects an error in a plug-in (for example, if there is a CORBA system exception or the plug-in reports that it is in an Inactive state) it removes it. It also checks to make sure that the plug-in is not overloaded, and, if it is, and multiple acceptable plug-ins are available, finds one with a lesser load level.

**Network-Triggered**

In many respects the interactions involved in routing traffic from a network plug-in to the appropriate service capability are very similar to those in the opposite direction. The major difference is not all traffic necessarily uses the SC Manager to accomplish this task, as the initial set-up of network-triggering may involve adding a listener from an ESPA instance directly to the plug-in, depending on design choices made by the developer. See the note at “Network Plug-in to Service Capability Manager” on page 5-23 for more information. In the description that follows, the general assumption is that the plug-in is using the SC Manager to retrieve its...
reference to the appropriate Service Capability, but a brief section at the end covers the alternative option.

**Plug-ins using the Service Capability Manager**

As in the case of the Plug-in Manager, each server/layer that contains ESPA SCs runs an instance of the SC Manager. All instances within one Network Gatekeeper node are synchronized and are treated as equals. Upon startup of a Network Gatekeeper SLEE, all ESPA SCs dealing with network-triggered sessions register their callback interfaces in the SC Manager executing in the same SLEE, and the change is propagated among all SC Manager instances.

The SC Manager checks to make sure that there is an appropriate ESPA SC that is active and not under severe overload. If no such EPSA SC is found, the Manager raises an SCSMgmtException to the getSCS method call. Under such a condition the plug-in should abort the dialogue since no suitable SC is available in the Network Gatekeeper cluster.

An SC returned by the SC Manager has always been checked and found working, but it is possible that something might have happened to it during the time it takes the plug-in to invoke the reportNotification method. Under such a condition the plug-in can either choose to use the getSCS method again or to abort the dialogue.

There is a pinging mechanism between the SC Manager and the SCs. If an SC is found to be not reachable, it is put in a “zombie list” maintained by the SC Manager. All entries in the zombie list are checked periodically by the SC Manager, and zombies that are found working after some time will be put back in the list of active SCs again. This mechanism helps to deal with cases where network connectivity is lost for some time between Network Gatekeeper hosts.

In the case of SC Manager inactivity, the plug-in can check periodically by invoking __non_existant() on the SCSManager. While in systems that use a variety of ORBs this action may produce variable results, as all Network Gatekeeper nodes now all use the same ORB, this is not an issue.

**Failure on notification reporting**

If the plug-in gets a CORBA exception when it attempts to send a reportNotification to an ESPA SC instance, it must evaluate the exception type.

If the exception is an org.omg.CORBA.SystemException and its status is org.omg.CORBA.CompletionStatus.COMPLETED_NO, the plug-in should try to acquire a reference to a different SC, either by invoking SCSDiscovery.getSCS if it is using the SC Manager or by changing to another directly registered callback if it is not. To guarantee that a broken network connection does not result in a lost relationship between Network Gatekeeper
High availability

and the underlying platform on that ESPA SC, the SC must be put in a zombie list until it reports that its status is now Active.

Listing 6-1  Examining the type of exception

```java
if ( ex instanceof org.omg.CORBA.SystemException ) {
    org.omg.CORBA.SystemException coSyEx = (org.omg.CORBA.SystemException) ex;
    if ( coSyEx.completed == org.omg.CORBA.CompletionStatus.COMPLETED_NO)
        retry = true;
}
```

If the completion status COMPLETED_YES or COMPLETED_MAYBE the plug-in cannot know for sure whether the notification has been handled or not and it should therefore treat the call normally. The plug-in should either start an activity supervision timer on the call that will expire after a certain time if no action is performed or it can rely on supervision timers in the MSC that will cause a TC_ABORT from the MSC after a period of time.

**Plug-ins not using the Service Capability Manager**

If the plug-in is designed not to use the Service Capability Manager, and to instead receive callback registrations directly from an ESPA SC, it must manage the high-availability tests that the SC Manager usually performs itself. If the plug-in detects an error that is not transient (as the plug-in defines transient), the faulty listeners should be removed. The SC will register as a listener again once it is activated/restarted. On transient errors the plug-in should keep the call back interface and try to reuse it with subsequent calls. In the case of several repeated errors the interface may be discarded even in this case. If no SCs are available or an error (for example a CORBA system exception) is encountered in an active session then the plug-in must take its own default action and also destroy all objects related to that session.

**High Availability between SESPA and ESPA**

High availability between SESPA and the ESPA SC it is currently using is provided by the SLEE Common Loader mechanism (See “Web Services layer to SESPA” on page 5-2) and the HA Handler. A SESPA SC registers its proxy object with an instance of HAHandler, as seen in Listing 6-2. This Handler is added to the SLEE Common Loader. If a SESPA SC looses contact
with the ESPA SC it currently uses, the Handler automatically switches the ESPA session object and the ESPA Manager object. This action is transparent to the SESPA object, so the new objects can be used for new actions automatically.

**Listing 6-2  Registering an object in the HA Handler**

```java
m_haHandler = HAHandler.createInstance(m_sc,
    this,
    null);

m_haProxy = (com.incomit.sespa.myservicecapability.MyServiceCapability)
    m_haHandler.getHAProxy(
        com.incomit.sespa.myservicecapability.MyServiceCapability.class);
```

If, however, the SESPA object has objects that have been created using the previous session or Manager objects, these are not automatically restored. The SESPA must implement this recovery mechanism itself should the developer want this capability. For example, if a SESPA implementation keeps track of the opened mailboxes it is using in an ESPA Messaging Service Capability, it would need to reopen those boxes after the HA switch has been accomplished. In the code fragment above, the HA handler is fetched and the Service Context and the object implementing the SESPA SC are provided as arguments. The third parameter, null in the example, is used to pass in any service specific recovery manager that SESPA chooses to implement. The method recoverSession(...) is called on this recovery manager after the HA Handler has restored the ESPA Session and Manager objects.
High availability
A Closer Look at One Layer: the Network Plug-in

Because telecom network elements can be quite varied, the most commonly developed extension to WebLogic Network Gatekeeper is a network plug-in. In this case, the functionality offered by the standard northbound interfaces and service capabilities is perfectly adequate to the support of the needs of the installation, but the piece connecting the Network Gatekeeper to the underlying network needs to be either customized or created whole cloth. The following eight chapters present a more detailed picture of the interfaces that standard Network Gatekeeper Service Capabilities present to network plug-ins as a way of introducing a lower level view of the extension process. These chapters include:

- “Call Control” on page 8-1
- “Call User Interaction” on page 9-1
- “SMS and MMS Messaging” on page 10-1
- “Message Sender” on page 11-1
- “Content Based Charging” on page 12-1
- “Subscriber Profile” on page 13-1
- “User Location” on page 14-1
- “Plug-ins that execute as both a SLEE service and a web application” on page 15-1
A Closer Look at One Layer: the Network Plug-in
Call Control

The following sections contain descriptions of the interface between the Call Control service capability and network plug-ins of the Call Control type:

- Call Control Interfaces
- Use cases

Call Control Interfaces

All Call Control interfaces are defined in the package `com.incomit.resources.callcontrol`. The interfaces are similar to the Parlay 3.2 call control interfaces.

Out of the box, Call Control plug-ins work with the following northbound interfaces:

- Extended Web Services Call Control
- Parlay X 1.0 Network Initiated Call
- Parlay X 1.0 Third Party Call

in conjunction with the Call Control Service Capability.
Figure 8-1 Call control interface
Implemented by the Call Control SC

The following interfaces are implemented in the Call Control service capability.

<table>
<thead>
<tr>
<th>Table 8-1 Implemented by the Call Control Service Capability; used by a Call Control plug-in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface</td>
</tr>
<tr>
<td>IrCallControlPlugInListener</td>
</tr>
<tr>
<td>IrAppCall</td>
</tr>
<tr>
<td>IrAppCallLeg</td>
</tr>
</tbody>
</table>

**IrCallControlPlugInListener**

The *IrCallControlPlugInListener* interface listens for events originating in the plug-in. There may be several listeners registered for each plug-in.

This interface inherits from the *SCS* interface. This makes it possible to narrow a Service Capability object retrieved from the SC Manager to an *IrCallControlPlugInListener*.

**Table 8-2 IrCallControlPlugInListener**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>callOverloadCeased</td>
<td>Reports that overload has ceased.</td>
</tr>
<tr>
<td>callOverloadEncountered</td>
<td>Reports an overload in the plug-in.</td>
</tr>
<tr>
<td>reportNotification</td>
<td>Sends notification about a network-triggered call.</td>
</tr>
</tbody>
</table>

**IrAppCall**

The *IrAppCall* interface is used to receive events related to a specific call.

**Table 8-3 IrAppCall**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>callEnded</td>
<td>Provides notification that the call has ended.</td>
</tr>
<tr>
<td>superviseRes</td>
<td>Responds to a previous call to <em>superviseReq</em>.</td>
</tr>
</tbody>
</table>
Table 8-3 IrAppCal

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>superviseErr</td>
<td>Responds to a previous call to superviseErr.</td>
</tr>
<tr>
<td>getInfoErr</td>
<td>Reports that the original request was erroneous, or resulted in an error condition. Asynchronous</td>
</tr>
<tr>
<td>getInfoRes</td>
<td>Reports time information for the finished call or call attempt, as well as the cause of the release based on what the getInfoReq request it has received specifies. Asynchronous</td>
</tr>
</tbody>
</table>

**IrAppCallLeg**

The IrAppCallLeg interface is used to receive events related to a specific call leg.

Table 8-4 IrAppCallLeg

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attachMediaRes/Err</td>
<td>Responds to a previous call to attachMediaReq.</td>
</tr>
<tr>
<td>callLegEnded</td>
<td>Provides a notification that the call leg has ended.</td>
</tr>
<tr>
<td>detachMediaRes/Err</td>
<td>Responds to a previous call to detachMediaReq.</td>
</tr>
<tr>
<td>eventReportRes/Err</td>
<td>Responds to a previous call to eventReportReq.</td>
</tr>
<tr>
<td>getInfoRes/Err</td>
<td>Responds to a previous call to getInfoReq.</td>
</tr>
<tr>
<td>routeErr</td>
<td>Reports an error when routing call leg.</td>
</tr>
<tr>
<td>superviseRes/Err</td>
<td>Responds to a previous call to superviseReq.</td>
</tr>
</tbody>
</table>
Implemented by the Network Plug-in

The following interfaces are implemented by a Call Control Network Plug-in:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CallControlResource</td>
<td>The initial object obtained from the Plug-in Manager.</td>
</tr>
<tr>
<td>IrCallControlManager</td>
<td>Deprecated.</td>
</tr>
<tr>
<td>IrCallControlManagerExt</td>
<td>The extended resource multi-party call control manager. Provides management</td>
</tr>
<tr>
<td></td>
<td>functions to the multi-party call control plug-ins.</td>
</tr>
<tr>
<td>IrCall</td>
<td>The logical representation of a call.</td>
</tr>
<tr>
<td>IrCallLeg</td>
<td>The logical representation of a call leg.</td>
</tr>
</tbody>
</table>

**CallControlResource**

The CallControlResource interface inherits from the Resource interface and adds an additional method that is used to obtain the Call Control manager.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getCallControlManager</td>
<td>Retrieves the Call Control manager.</td>
</tr>
</tbody>
</table>

**IrCallControlManager - Deprecated -**

The IrCallControlManager interface is deprecated.
IrcallControlManagerExt

The IrcallControlManagerExt interface is used to create calls, handle load control, and register plug-in listeners. Only one instance of this type is required.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>createCallCtx</td>
<td>Creates a new call object.</td>
</tr>
<tr>
<td>disableNotification</td>
<td>Disables a call notification.</td>
</tr>
<tr>
<td>enableNotification</td>
<td>Enables a call notification for network initiated traffic.</td>
</tr>
<tr>
<td>setCallLoadControlCtx</td>
<td>Used for call gapping.</td>
</tr>
</tbody>
</table>

Ircall

The Ircall interface represents a call. Normally, each active call has one object instance implementing this interface. If the plug-in uses the session id, however, only one instance is required.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>createCallLeg</td>
<td>Creates a new call leg.</td>
</tr>
<tr>
<td>deassignCall</td>
<td>De-assigns the call without disconnect.</td>
</tr>
<tr>
<td>getCallLegs</td>
<td>Retrieves a list of all call legs belonging to this call.</td>
</tr>
<tr>
<td>getInfoReq</td>
<td>Requests call information be sent when the call is disconnected.</td>
</tr>
<tr>
<td>release</td>
<td>Releases this call and disconnects all call legs.</td>
</tr>
<tr>
<td>setAdviceOfCharge</td>
<td>Sets advice of charge information.</td>
</tr>
<tr>
<td>setCallback</td>
<td>Changes the callback object used for this call.</td>
</tr>
<tr>
<td>setChargePlan</td>
<td>Sets the call charge plan.</td>
</tr>
<tr>
<td>superviseReq</td>
<td>Sets the granted connect time for this call.</td>
</tr>
</tbody>
</table>

Table 8-7 IrcallControlManagerEx

Table 8-8 Ircall
**IrCallLeg**

The IrCallLeg interface represents a call leg. For each leg in a call there must be an instance implementing this interface, unless the plug-in uses the session id, in which case only one instance is required.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attachMediaReq</td>
<td>Attaches a detached call leg.</td>
</tr>
<tr>
<td>continueProcessing</td>
<td>Resumes call processing.</td>
</tr>
<tr>
<td>deassign</td>
<td>Releases control of leg. The call will remain, but all requested call info and events are disabled.</td>
</tr>
<tr>
<td>detachMediaReq</td>
<td>Detaches this leg from the call. No connection will exist with other call legs.</td>
</tr>
<tr>
<td>eventReportReq</td>
<td>Requests event reports.</td>
</tr>
<tr>
<td>getCall</td>
<td>Retrieves the call object to which this leg belongs.</td>
</tr>
<tr>
<td>getInfoReq</td>
<td>Requests call information when the call leg is disconnected.</td>
</tr>
<tr>
<td>release</td>
<td>Disconnects this call leg.</td>
</tr>
<tr>
<td>routeReq</td>
<td>Routes this call leg.</td>
</tr>
<tr>
<td>setAdviceOfCharge</td>
<td>Sets advice of charge information.</td>
</tr>
<tr>
<td>setCallback</td>
<td>Changes the callback object used for this call.</td>
</tr>
<tr>
<td>setChargePlan</td>
<td>Sets the call charge plan for this leg.</td>
</tr>
<tr>
<td>superviseReq</td>
<td>Sets granted connect time for this call leg.</td>
</tr>
</tbody>
</table>

**Use cases**

**Application-initiated two-party call**

The following sequence diagram shows a basic application initiated two-party call.

**Note:** The process by which the SC obtains the Call Control Manager is not shown here.
Details about the sequence diagram:

1. The client application requests that a new call be created. This object is created and returned.
2. A call leg is created for the first party by calling `createCallLeg`
3. Event reports and call information are requested. The charge plan is set and the call is routed. In general, the plug-in should buffer these calls and send everything when `routeReq` is called. The calls `getInfoReq` and `setChargePlan` are not mandatory.

4. When the call leg is answered, the answer event is sent to the application with `eventReportRes`. Other events could also be sent here, such as busy or no answer.

5. A second call leg is connected to this call, again using the `createCallLeg` call.

6. The same methods may be called as were called in creating the first call leg.

7. The `attachMediaReq` call is invoked on the first call leg. Note that all call legs created by an application are initially detached.

8. The first call leg is in an interrupted state, so call processing is on hold until `continueProcessing` is invoked on this leg. At this time any buffered events from the previous steps are sent.

9. The second leg is attached and `continueProcessing` is invoked on it.

10. The application releases the call.

11. Any information requested for each call leg is sent.

12. The end of the call notification is sent.

**Network-triggered call**

The following sequence diagram shows a basic network-triggered call. This might, for example, be a call to an enterprise support call center that is routed depending on the time of day the call comes in.

**Note:** For each new call the plug-in either uses the SC Manager to obtain a listener interface or, in the case of directly registered plug-in listeners, selects one to use.
Details about the sequence diagram:

1. Before the diagram starts, the ESPA SC, acting on behalf of the application, creates a listener interface and notifies the plug-in about it, either directly, or through the SC Manager.

2. The plug-in receives a network triggered call event and creates the call and call leg object that are sent to the plug-in listener. The call is now in an interrupted state.

3. The service capability uses the `setCallback` method to request event notifications for the first call leg.

4. The service capability creates a new call leg by calling `createCallLeg`.

5. Event reports are requested for this second leg.

6. The call leg is routed, but no operations are sent until `continueProcessing` is invoked.

7. The second call leg, the one to which the application has routed, is answered.
8. The second call leg is attached to the call and `continueProcessing` is invoked.

9. The call is de-assigned. This releases all objects related to this call. The call remains active in the network.
Call User Interaction

The following sections contain descriptions of the interface between the Call User Interaction service capability and network plug-ins of the Call User Interaction type:

- Call User Interaction interfaces
- Use case for Call User Interaction

Call User Interaction interfaces

All Call User Interaction interfaces are defined in two files. General level User Interaction interfaces are defined in `UserInteractionResource_IF.idl` and specific to Call User Interaction interfaces are found in `UserInteractionCallResource_IF.idl`. Both are located in `bea\w1ng22\esdk\idl\plugin_if\ui`.

Out of the box, Call User Interaction plug-ins work with the following northbound interfaces:

- Extended Web Services Call User Interaction
- Extended Web Services Messaging Interaction

in conjunction with the User Interaction Service Capability.
Implemented by the Call User Interaction SC

The following interfaces are implemented by the Call User Interaction SC.

Table 9-1 Implemented by Call User Interaction SC, used by a Call User Interaction plug-in

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IrAppUI</td>
<td>Callback interface for receiving responses to operations performed via the IrUI interface.</td>
</tr>
<tr>
<td>IrAppUICall</td>
<td>Callback interface for receiving responses to operations performed via the IrUICall interface.</td>
</tr>
</tbody>
</table>
IrAppUI

An object that implements the IrAppUI interface is used to receive responses performed via the IrUI interface.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendInfoRes</td>
<td>Receives the result of a successful sendInfoReq request.</td>
</tr>
<tr>
<td>sendInfoErr</td>
<td>Receives the result of a failed sendInfoRes request.</td>
</tr>
<tr>
<td>sendInfoAndCollectRes</td>
<td>Receives the result of a successful sendInfoAndCollectReq request.</td>
</tr>
<tr>
<td>sendInfoAndCollectErr</td>
<td>Receives the result of a failed sendInfoAndCollectReq request.</td>
</tr>
<tr>
<td>userInteractionFaultDetected</td>
<td>Indicates that a fault has been detected in the user interaction.</td>
</tr>
</tbody>
</table>

IrAppUICall

An object that implements the IrAppUICall interface is used to receive responses for calls using the IrUICall interface.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>recordMessageRes</td>
<td>Receives the result of a successful recordMessageReq request.</td>
</tr>
<tr>
<td>recordMessageErr</td>
<td>Receives the result of a failed recordMessageReq request.</td>
</tr>
<tr>
<td>abortActionRes</td>
<td>Receives the result of a successful request to abort a user interaction operation.</td>
</tr>
<tr>
<td>abortActionErr</td>
<td>Receives the result of a failed request to abort a user interaction operation.</td>
</tr>
</tbody>
</table>
Call User Interaction

**Implemented by the Network Plug-in**

The following interfaces are implemented by the network plug-in.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Resource</strong></td>
<td>The initial object obtained from Plug-in Manager. Base interface implemented by all plug-ins.</td>
</tr>
<tr>
<td><strong>UserInteractionCallResource</strong></td>
<td>Retrieves the Manager object that is used to create and release sessions using the Call User Interaction plug-in.</td>
</tr>
<tr>
<td><strong>IrUI</strong></td>
<td>Provides functions to send information (typically prompt messages) to the end user and to order collection of information from the end user. The information is typically collected from IVRs capable of collecting user input in the form of DTMF (Touch Tone tones). It also provides functions for ending, or releasing, a user information session.</td>
</tr>
<tr>
<td><strong>IrUICall</strong></td>
<td>Provides functions for ordering the recording of messages and for aborting a request.</td>
</tr>
<tr>
<td><strong>IrUICallManager</strong></td>
<td>Deprecated.</td>
</tr>
<tr>
<td><strong>IrUICallManagerExt</strong></td>
<td>Provides functions for creating objects representing a Call User Interaction session.</td>
</tr>
</tbody>
</table>

**UserInteractionCallResource**

UserInteractionCallResource inherits from the Resource interface and adds an additional method that is used to obtain the Call User Interaction manager.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getUICallManager</td>
<td>Retrieves a Call User Interaction manager. The returned object must be narrowed from an IrUICallManager to an IrUICallManagerExt object since IrUICallManager is deprecated.</td>
</tr>
</tbody>
</table>
**IrUI**

The User Interaction Service Interface provides functions to send information to, or gather information from, a user.

**Table 9-6  IrUI**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendInfoReq</td>
<td>Plays an announcement or sends other type of information to the end user.</td>
</tr>
<tr>
<td>sendInfoAndCollectReq</td>
<td>Plays an announcement or sends other type of information to the end user and collects input from the end user.</td>
</tr>
<tr>
<td>release</td>
<td>Releases an user interaction session. Releases all resources associated with the user interaction session and terminates the ongoing user interaction session.</td>
</tr>
</tbody>
</table>

**IrUICall**

The User Interaction Call Service Interface provides functions for requesting that responses from the end user be recorded and for aborting user interaction operations.

**Table 9-7  IrUICall**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>recordMessageReq</td>
<td>Requests that a message from the end user be recorded.</td>
</tr>
<tr>
<td>abortActionReq</td>
<td>Aborts a previously ordered operation.</td>
</tr>
</tbody>
</table>

**IrUICallManager**

IrUICallManager is deprecated.
**IrUICallManagerExt**

IrUICallManagerExt is used to create Call User Interaction sessions. Only one instance of this type is required.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>createUICallCtx</td>
<td>Creates a new Call User Interaction object.</td>
</tr>
</tbody>
</table>

**Use case for Call User Interaction**

A Call User Interaction plug-in only supports application-initiated operations. Network-triggered use is not supported.

A Call User Interaction session is established on an already existing call. That call has an ID that was created at the time the call was initiated. The call itself - on which the Call User Interaction session acts - can be either application or network originated.

**Application-initiated Call User Interaction Session**

Figure 9-2 shows the basic set of interactions between a Call User Interaction client (initially an application, represented in this sequence by the Call UI ESPA SC) and the Call User Interaction network plug-in that is followed in creating and releasing a UI session.

Figure 9-2  Application-initiated usage of Call User Interaction plug-in.
Details about the sequence diagram:

1. Before the diagram begins two things have already occurred:
   - The application client has requested and the Call UI ESPA SC has received a plug-in. See “Typical Traffic Path for Application-Initiated Requests” on page 5-14 for an overview of this process.
   - The Call UI ESPA SC has access to an IrAppCall object representing an ongoing call. See Table , “IrAppCall,” on page 8-3 for more information.

2. A User Interaction session is created by calling createUICallCtx on the implementation of the IrUICallManagerExt interface on the plug-in. The interface and a Call UI session identifier are returned.

3. The sendInfoReq method is called on the IrUI interface, requesting that the information making up the user interaction (pre-recorded messages, etc.) be sent to one or more participants (represented by call legs) in the ongoing call.

4. The status result of the sendInfoReq called is returned asynchronously to the IrAppUI interface as a sendInfoRes, if the information is successfully delivered to the ongoing call, or a sendInfoErr if there is a problem.

5. If desired, additional methods can be called on the IrUICall interface. The session is ended by calling release on that interface.

Termination of the ongoing call on which the Call UI session is created is handled separately elsewhere.
CHAPTER 10

SMS and MMS Messaging

The following sections contain descriptions of the interface between the Messaging service capability and network plug-ins of the Messaging (SMS and MMS) type:

- Messaging Interfaces
- Use cases for SMS

Messaging Interfaces

Although both SMS and MMS are handled by the Messaging SC, the interfaces required for handing them off to network plug-ins differ slightly. All messaging interfaces are defined in the package com.incomit.resources.messaging.

Out of the box, Messaging plug-ins work with the following northbound interfaces:

- Legacy SMPP (for SMS)
- Legacy MM7 (for MMS)
- Parlay X (1.0 and 2.1) SMS (for SMS)
- Parlay X (1.0 and 2.1) Multimedia Message (for MMS)
- Extended Web Services Messaging

in conjunction with the Messaging Service Capability.
SMS and MMS Messaging

SMS interfaces

Note: SMS Messaging Network Plug-ins also handle traffic for the Messaging User Interaction SC.

Figure 10-1 SMS interface
Implemented by the Messaging Service Capability

The following SMS interfaces are implemented by the Messaging SC:

Table 10-1 Implemented by the Messaging SC; used by an SMS Messaging plug-in

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCS</td>
<td>The base interface for Service Capabilities.</td>
</tr>
<tr>
<td>MessageListener</td>
<td>The base interface for Messaging Service Capabilities.</td>
</tr>
<tr>
<td>MessageListenerExt</td>
<td>Extended base interface for Messaging Service Capabilities. All methods are now deprecated.</td>
</tr>
<tr>
<td>AsyncSmsResourceListener</td>
<td>Listener for notifications for previously sent or deleted messages. A notification is sent when the request has been accepted or rejected by the network node. It can also be sent when the network node has failed to reach the network node.</td>
</tr>
<tr>
<td>AsyncSmsResourceNotification</td>
<td>Listener for result notifications for previously sent messages and network initiated messages. Status notifications are sent when the status of a message has been changed, for example it has been read by the terminal.</td>
</tr>
</tbody>
</table>

**MessageListener**

*MessageListener* listens for events originating in the plug-in. There may be several listeners registered for each plug-in in the SC Manager.

This interface inherits from the SCS interface. It makes it possible to narrow an SC object retrieved from the SC Manager to a *MessageListener*.

Table 10-2 MessageListener

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>messageArrived</td>
<td>Deprecated.</td>
</tr>
<tr>
<td>messageResult</td>
<td>Deprecated.</td>
</tr>
<tr>
<td>messagingSessionAborted</td>
<td>Indicates that a previously created messaging session has been aborted</td>
</tr>
</tbody>
</table>
MessageListenerExt

MessageListenerExt listens for events originating in the plug-in. There may be several listeners registered for each plug-in in the SC Manager.

This interface inherits from the MessageListener interface. It makes it possible to narrow an SC object retrieved from the SC Manager to a MessageListenerExt.

Table 10-3  MessageListenerExt

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>messageArrivedCtx</td>
<td>Deprecated.</td>
</tr>
<tr>
<td>sendResultCtx</td>
<td>Deprecated</td>
</tr>
<tr>
<td>messageResultExt</td>
<td>Deprecated.</td>
</tr>
</tbody>
</table>

AsyncSmsResourceListener

AsyncSmsResourceListener is used with asynchronous methods. It receives results of requests to the plug-in.

Table 10-4  AsyncSmsResourceListener

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendSmsRes</td>
<td>Called by the plug-in when an SMS has successfully been sent to the network. Response to sendSmsReq.</td>
</tr>
<tr>
<td>sendSmsErr</td>
<td>Called by the plug-in when the SMS could not be delivered to the network. Response to sendSmsReq.</td>
</tr>
<tr>
<td>deleteSmsRes</td>
<td>Called by the plug-in when an SMS has successfully been deleted from storage in the network. Response to deleteSmsReq.</td>
</tr>
<tr>
<td>deleteSmsErr</td>
<td>Called by the plug-in when a SMS could not be deleted from the storage in the network. Response to deleteSmsReq.</td>
</tr>
</tbody>
</table>
AsyncSmsResourceNotification

AsyncSmsResourceNotification is used with asynchronous methods. It listens for events originating in the plug-in. It inherits from MessageListenerExt interface, which makes it possible to narrow a MessageListenerExt object to an AsyncSmsResource.

Table 10-5  AsyncSmsResourceNotification

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>smsArrivedReq</td>
<td>Called by the plug-in when a network initiated SMS has reached the plug-in.</td>
</tr>
<tr>
<td>smsDeliveryAckReq</td>
<td>Called by the plug-in when an delivery acknowledgement for a previously sent SMS has reached the plug-in.</td>
</tr>
</tbody>
</table>

Implemented by the SMS Network Plug-in

The following interfaces are implemented by an SMS plug-in:

Table 10-6  Implemented by an SMS Network Plug-in

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource</td>
<td>The base interface for plug-ins.</td>
</tr>
<tr>
<td>MessagingResource</td>
<td>The base interface for SMS Messaging plug-ins.</td>
</tr>
<tr>
<td>MessagingResourceExt</td>
<td>The extended base interface for SMS messaging plug-ins.</td>
</tr>
<tr>
<td>AsyncSmsResource</td>
<td>Sends messages and deletes previously sent messages from storage.</td>
</tr>
<tr>
<td>AsyncSmsResourceNotificationListener</td>
<td>Receives delivery results for network initiated messages and status notifications to be sent to the messaging SC.</td>
</tr>
</tbody>
</table>
**MessagingResource**

*MessagingResource* inherits from the generic *Resource* interface. The interface makes it possible to narrow a *Resource* object retrieved from the SC Manager to a *MessagingResource* object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>createMessagingSession</td>
<td>Creates a Message Based User Interaction session as a result of a call from the Message Based User Interaction SC.</td>
</tr>
<tr>
<td>releaseMessagingSession</td>
<td>Releases a previously created Message Based Interaction session as a result of a call from the Message Based User Interaction SC.</td>
</tr>
<tr>
<td>sendMessage</td>
<td>Deprecated.</td>
</tr>
<tr>
<td>addDefaultMessageListener</td>
<td>Deprecated.</td>
</tr>
<tr>
<td>addMessageListener</td>
<td>Deprecated.</td>
</tr>
<tr>
<td>removeMessageListener</td>
<td>Deprecated.</td>
</tr>
</tbody>
</table>

**MessagingResourceExt**

*MessagingResourceExt* is an extended version of the *MessagingResource* interface. The interface makes it possible to narrow a *MessagingResource* object retrieved from an SC Manager to a *MessagingResourceExt* object.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendMessageCtx</td>
<td>Deprecated.</td>
</tr>
<tr>
<td>deleteMessageCtx</td>
<td>Deprecated.</td>
</tr>
<tr>
<td>enableMessagingNotificationCtx</td>
<td>Enables message notification for a specified set of notification criteria to the Messaging SC. Used if the network protocol supports enabling of notifications.</td>
</tr>
<tr>
<td>enableMessagingUINotificationCtx</td>
<td>Enables message notification for a specified set of notification criteria to the Messaging User Interaction SC.</td>
</tr>
</tbody>
</table>
AsyncSmsResource

AsyncSmsResource is an extended version of the MessagingResourceExt interface designed to support asynchronous methods. The interface makes it possible to narrow a MessagingResourceExt object to an AsyncSmsResource.

Table 10-9  AsyncSmsResource

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendSmsReq</td>
<td>Sends an SMS.</td>
</tr>
<tr>
<td>deleteSmsReq</td>
<td>Deletes an SMS from the underlying storage.</td>
</tr>
</tbody>
</table>

AsyncSmsResourceNotificationListener

AsyncSmsResourceNotificationListener is used with asynchronous methods.

Table 10-10  AsyncSmsResource

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>smsArrivedRes</td>
<td>Notifies the plug-in that a network initiated SMS was successfully processed by the Service Capability.</td>
</tr>
<tr>
<td>smsArrivedErr</td>
<td>Notifies the plug-in that a network initiated SMS was received by the Service Capability, but it could not be processed.</td>
</tr>
<tr>
<td>smsDeliveryAckRes</td>
<td>Notifies the plug-in that a delivery acknowledgement was successfully processed by the Service Capability.</td>
</tr>
<tr>
<td>smsDeliveryAckErr</td>
<td>Notifies the plug-in that a delivery acknowledgement was received by the Service Capability, but it could not be processed.</td>
</tr>
</tbody>
</table>
MMS interfaces

Figure 10-2  MMS interface
Implemented by the Messaging Service Capability.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MmsListener</td>
<td>Deprecated. Call back interface for handling network initiated MMS messages.</td>
</tr>
<tr>
<td>MmsListenerExt</td>
<td>Deprecated. Extended call back interface for handling network initiated MMS messages. Also provides functionality for receiving notifications about a sent MMS.</td>
</tr>
<tr>
<td>MmsResourceExt2</td>
<td>Deprecated. Extended call back interface for handling network initiated MMS messages.</td>
</tr>
<tr>
<td>AsyncMmsResourceNotification</td>
<td>Delivers network triggered MMSes or delivery acknowledgements to the Messaging SC.</td>
</tr>
<tr>
<td>AsyncMmsResourceListener</td>
<td>Reports success or failure of send MMS or delete MMS operations.</td>
</tr>
</tbody>
</table>

**MmsListener**

*MmsListener* listens for events originating in the plug-in. There may be several listeners registered for each plug-in in the SCS manager.

This interface inherits from the SCS interface. It makes it possible to narrow a SCS object retrieved from the SC Manager to an *MmsListener*.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmMessageArrived</td>
<td>Deprecated.</td>
</tr>
<tr>
<td>mmMessageResult</td>
<td>Deprecated.</td>
</tr>
</tbody>
</table>

**MmsListenerExt**

*MmsListenerExt* listens for events originating in the plug-in. There may be several listeners registered for each plug-in in the SCS Manager.
This interface inherits from the SCS interface. It makes it possible to narrow a SC object retrieved from the SC Manager to a MmsListenerExt. It also extends the MmsListener interface.

Table 10-13 MmsListenerExt

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmMessageArrivedExt</td>
<td>Deprecated.</td>
</tr>
<tr>
<td>sendResultCtx</td>
<td>Deprecated.</td>
</tr>
</tbody>
</table>

MmsResourceExt2

MmsResourceExt2 is an extended interface to for sending MMS messages. It extends the MmsResourceExt interface, which makes it possible to narrow a MmsResourceExt object to an MmsResourceExt2.

Table 10-14 MmsResourceExt2

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendMmMessageExtCtx</td>
<td>Deprecated.</td>
</tr>
</tbody>
</table>

AsyncMmsResourceNotification

Receives messages from an MMS plug-in. It extends the MmsListenerExt2 interface, which makes it possible to narrow an MmsListenerExt2 object to an AsyncMmsResourceNotification.

Table 10-15 AsyncMmsResourceNotification

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmMessageArrivedReq</td>
<td>Called by the plug-in when a network initiated MMS has reached the plug-in.</td>
</tr>
<tr>
<td>mmMessageDeliveryAckReq</td>
<td>Called by the plug-in when a delivery acknowledgement for a previously sent MMS has reached the plug-in.</td>
</tr>
</tbody>
</table>
**AsyncMmsResourceListener**

Receives responses to the outcome of send or delete MMS operations. Passed in with request.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendMMMessageRes</td>
<td>Called by the plug-in when an MMS has successfully been sent to the network. Response to sendMMMessageReq.</td>
</tr>
<tr>
<td>sendMMMessageErr</td>
<td>Called by the plug-in when the MMS could not be delivered to the network. Response to sendMMMessageReq.</td>
</tr>
<tr>
<td>deleteMMMessageRes</td>
<td>Called by the plug-in when an MMS has successfully been deleted from the storage in the network. Response to deleteMMMessageReq.</td>
</tr>
<tr>
<td>deleteMMMessageErr</td>
<td>Called by the plug-in when an MMS could not be deleted from the storage in the network. Response to deleteMMMessageReq.</td>
</tr>
</tbody>
</table>

**Implemented by the MMS Network Plug-in**

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource</td>
<td>Initial object obtained from Plug-in Manager. Base interface implemented by all plug-ins.</td>
</tr>
<tr>
<td>MmsResource</td>
<td>Deprecated. Used for sending MM Messages.</td>
</tr>
<tr>
<td>MmsResourceExt</td>
<td>The extended MMS messaging interface enables notifications.</td>
</tr>
<tr>
<td>MmsResourceExt2</td>
<td>Deprecated. The extensions to the extended MMS messaging interface provides functions for sending messages.</td>
</tr>
</tbody>
</table>
MmsResource

**MmsResource** is the base interface for sending MMS messages. It extends the Resource interface, which makes it possible to narrow a Resource object to an MmsResource.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendMmMessage</td>
<td>Deprecated.</td>
</tr>
<tr>
<td>addDefaultMmsListener</td>
<td>Deprecated.</td>
</tr>
<tr>
<td>removeDefaultMmsListener</td>
<td>Deprecated.</td>
</tr>
</tbody>
</table>

MmsResourceExt

**MmsResourceExt** is an extended interface to for sending MMS messages. It extends the MmsResource interface, which makes it possible to narrow an MmsResource object to an MmsResourceExt.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendMmMessageCtx</td>
<td>Deprecated.</td>
</tr>
</tbody>
</table>
MmsResourceExt2

MmsResourceExt2 is an extended interface to for sending MMS messages, now all deprecated. It extends the MmsResourceExt interface, which makes it possible to narrow an MmsResourceExt object to an MmsResourceExt2

<table>
<thead>
<tr>
<th>Table 10-19 MmsResourceExt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
</tr>
<tr>
<td>deleteMessageCtx</td>
</tr>
<tr>
<td>enableMmMessagingNotificationCtx</td>
</tr>
</tbody>
</table>

AsyncMmmResource

AsyncMmmResource is used for sending MMS messages.

<table>
<thead>
<tr>
<th>Table 10-21 AsyncMmmResource</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
</tr>
<tr>
<td>sendMMMessageReq</td>
</tr>
<tr>
<td>deleteMMMessage</td>
</tr>
</tbody>
</table>
AsyncMmsResourceNotificationListener

AsyncMmsResourceNotificationListener is used for reporting successful or unsuccessful delivery of network triggered MMSes or delivery acknowledgements to the Messaging SC. Passed in with initial request to the service capability.

Table 10-22 AsyncMmsResourceNotificationListener

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mmMessageArrivedRes</td>
<td>Notifies the plug-in that a network initiated MMS was successfully processed by the Service Capability.</td>
</tr>
<tr>
<td>mmMessageArrivedErr</td>
<td>Notifies the plug-in that a network initiated MMS was received by the Service Capability, but it could not be processed.</td>
</tr>
<tr>
<td>mmMessageDeliveryAckRes</td>
<td>Notifies the plug-in that a delivery acknowledgement was successfully processed by the Service Capability.</td>
</tr>
<tr>
<td>mmMessageDeliveryAckErr</td>
<td>Notifies the plug-in that a delivery acknowledgement was received by the Service Capability, but it could not be processed.</td>
</tr>
</tbody>
</table>

Use cases for SMS

Registering the Service Capability

The following sequence diagram shows the Messaging SC registering itself in the SC Manager. If you are creating only a messaging plug-in, you do not need to perform this task, but you will use its results. The information is included simply to illustrate the process.
Details about the sequence diagram:

1. First, the AsyncSmsResourceNotification implementation of the Messaging Service Capability fetches the SLEESCSRegistration interface from the SLEE’s SC Manager.

2. It then registers the AsyncSmsResourceNotification interface with the SC Manager (on SLEESCSRegistration). The class implementing AsyncSmsResourceNotification is used by the plug-in to notify the Messaging SC about events related to the outcome of sendSmsReq operations (see Application-initiated send message) and smsArrivedReq operations (see Network-initiated messages).

Application-initiated send message

The following sequence diagram shows a basic send message operation.

Note: The plug-in uses the SC Manager to get a Messaging SC to which it can send the delivery receipts, via the AsyncSMSResourceNotification interface. This is because a long time may elapse before a delivery report is sent back to the plug-in from the network. In effect the Messaging SC itself acts as a listener for delivery reports.
Details about the sequence diagram:

1. Before the diagram starts, the Messaging SC (using an asynchronous interface) has acquired an SMS plug-in.

2. The Messaging SC calls `sendSmsReq` on the implementation of the plug-in’s `AsyncSmsResource`. The call provides an `AsyncSmsResourceListener` object to the plug-in.

3. When the SMS message has been passed on to the network node, the plug-in calls `sendSmsRes` on the listener object.

4. At some later time, when a delivery report for the SMS is received from the network, the plug-in fetches a Messaging SC by calling a `getSCSCtx` operation on the `SLEESCDiscovery` interface. This SC implements the `AsyncSmsResourceNotification` interface, which allows it to act as a listener.

5. The returned interface returned is narrowed to the appropriate object by using the `AsyncResourceNotificationHelper` class. This class is auto-generated when the Java stubs are created from the IDL interface.

6. The plug-in instantiates its implementation of the `AsyncSmsResourceNotificationListener`, used to receive acknowledgements that the delivery report has been received.
7. The plug-in calls sendSMSDeliveryAck on the AsyncSmsResourceNotification implementation.

8. The Messaging SC calls smsDeliveryAckRes when the delivery acknowledgement has been received through the asynchronous interface. The Messaging SC creates a CDR.

Note: The AsyncSmsResourceNotificationListener CORBA object must be explicitly deactivated.

Network-initiated messages

The following sequence diagram shows handling a basic network-initiated message.

Note: An application must register its interest in receiving notifications of network-initiated messages before they can be processed. The application registers listeners on mailboxes in the Messaging SC. The listeners report events - like the arrival of messages. (For more information on mailboxes, see “Messaging” in “Supported Service Capabilities” in the Architectural Overview.) When a network-initiated message arrives, the plug-in uses the SC Manager to get a listener interface on the Messaging SC. It is not necessary to enable specific listeners for incoming traffic on the Messaging SC, because the Messaging SC listens to all incoming traffic and distributes the message to a mailbox.

Figure 10-5 Network-initiated messages

Details about the sequence diagram:

1. The plug-in receives a network-initiated message from the underlying telecom network.
2. The plug-in fetches an appropriate SC based on a set of properties and a type identifier it supplies in the `getSCSCtx(...)` call. In this case the type identifier is `SCS.MESSAGING_TYPE` for Messaging (defined in `resource_common.idl`).

3. A list of appropriate Messaging SCs are returned. Because the SLEESCSDiscovery interface provides load balancing, the plug-in should use the first item in the list.

4. The `AsyncSmsResourceNotification` interface on the Messaging SC is acquired by calling `narrow` on the `AsyncSmsResourceNotificationHelper` class. This class is auto-generated when the Java stubs are created from the IDL interface.

5. The plug-in calls `smsArrivedReq` on the `AsyncSmsResourceNotification` interface.

6. The `smsDeliveryAckRes` is called on the plug-in's `AsyncSmsResourceNotificationListener` interface by the Messaging SC.

7. The `AsyncSmsResourceNotificationListener` CORBA object must be explicitly deactivated.
CHAPTER 11

Message Sender

The following sections contain descriptions of the interface between the Message Sender service capability and network plug-ins of the Message Sender type. Message Sender is used for *sending* messages only, and does not include the mailbox functionality in the Messaging service capability. For more information on the distinction between Messaging and Message Sender, see the *WebLogic Network Gatekeeper Architectural Overview*.

- Message Sender Interfaces
- Use case

**Message Sender Interfaces**

All Message Sender interfaces are defined in the package `com.bea.wlcp.wlng.plugin.messagesender`.

Out of the box, Message Sender plug-ins work with the following northbound interfaces:

- Legacy PAP
- Extended Web Services PAP

in conjunction with the *Message Sender Service Capability*. 
The following interfaces are implemented by the Message Sender SC.

**Table 11-1 Implemented by the Message Sender SC; used by a Message Sender plug-in**

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IrStatusNotificationReqListener</td>
<td>Listens for status notifications for previously sent messages. Status</td>
</tr>
<tr>
<td></td>
<td>notifications are sent when the status of a message has been changed; for</td>
</tr>
<tr>
<td></td>
<td>example it has been read by the terminal.</td>
</tr>
<tr>
<td>IrMessageResListener</td>
<td>Listener for response notification for previously sent messages. A</td>
</tr>
<tr>
<td></td>
<td>response notification is sent when the message has been accepted by the</td>
</tr>
<tr>
<td></td>
<td>network node.</td>
</tr>
</tbody>
</table>

**IrStatusNotificationReqListener**

The `IrStatusNotificationReqListener` interface listens for events originating in the plug-in concerning a change in the status of previously sent messages. This interface inherits from the SCS.
interface, which makes it possible to narrow an SC object retrieved from the SC Manager to a 
IrStatusNotificationReqListener.

Table 11-2  IrStatusNotificationReqListener

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendStatusNotificationReq</td>
<td>Listens for change in status notifications from the plug-in for previously sent messages.</td>
</tr>
</tbody>
</table>

**IrMessageResListener**

An object implementing this interface is sent to the plug-in in the sendMessageReq() call. It is used to inform the SC if the plug-in was able to successfully send the message to the underlying network node.

Table 11-3  IrMessageResListener

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendMessageRes</td>
<td>Returns a success response to the SC for a sendMessageReq.</td>
</tr>
<tr>
<td>sendMessageErr</td>
<td>Returns an error response to the SC for a sendMessageReq.</td>
</tr>
</tbody>
</table>

**Implemented by a Message Sender Network Plug-in**

The following interfaces are implemented by the Message Sender plug-in

Table 11-4  Implemented by a plug-in for Message Sender

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource</td>
<td>Initial object obtained from Plug-in Manager. Base interface implemented by all plug-ins.</td>
</tr>
<tr>
<td>IrMessageSenderPlugin</td>
<td>Used for sending messages.</td>
</tr>
<tr>
<td>IrStatusNotificationResListener</td>
<td>Used for receiving results of change of status notifications sent to the Message Sender SC.</td>
</tr>
</tbody>
</table>
IrMessageSenderPlugin

The IrMessageSenderPlugin interface extends the base Resource plug-in interface which makes it possible to narrow a Resource object to an IrMessageSenderPlugin.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendMessageReq</td>
<td>Sends a message. Returns immediately.</td>
</tr>
</tbody>
</table>

IrStatusNotificationResListener

An object implementing this interface is passed to the Message Sender service capability in the call sendStatusNotificationReq. It is used to inform the plug-in whether the service capability successfully received the change in message status notification sent to it by the plug-in.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>statusNotificationRes</td>
<td>Returns a success result to a change in status notification.</td>
</tr>
<tr>
<td>statusNotificationErr</td>
<td>Returns an error result to a change in status notification.</td>
</tr>
</tbody>
</table>

Use case

The following sequence diagram shows a basic message sending interaction between the Message Sender SC and the Message Sender Network Plug-in.

**Note:** The plug-in uses the SC manager to get a Message Sender SC to which it can send change of status notification, via the IrStatusNotificationReqListener interface. This is because it may take long time before the network notifies the plug-in that the status of the message has changed (for example, it has been displayed on the receiver's terminal). In effect, the Message Sender SC itself acts as a listener for status notifications.
Figure 11-2  Application-initiated send message

Details about the sequence diagram:

1. Before the diagram starts, the Message Sender SC has:
   - Registered the implementation of its `IrStatusNotificationReqListener` interface in the SLEESCSManager
   - Asked for and received a Message Sender plug-in from the Plug-in Manager

2. The Message Sender SC instantiates the implementation of the `IrMessageResListener` interface


4. When the plug-in has successfully passed the message on to the network, it calls a `sendMessageRes` on the `IrMessageResListener` object. This CORBA object must be explicitly deactivated after this call.

5. When a change in status notification reaches the plug-in from the network, it asks the SC Manager for a Message Sender SC, which it then narrows to an `IrStatusNotificationReqListener` on which it performs a `sendStatusNotificationReq`.

6. The Message Sender SC calls `statusNotificationRes` on the `IrStatusNotificationResListener` object it received with the `sendMessageRes`. This CORBA object must be explicitly deactivated after this call.
CHAPTER 12

Content Based Charging

The following sections contain descriptions of the interface between the Content Based Charging service capability and network plug-ins of the Content Based Charging type.

- Content Based Charging Interfaces
- Use case

For more information on Content Based Charging, see the Charging section of the “Supported Service Capabilities” chapter in the WebLogic Network Gatekeeper Architectural Overview.

Content Based Charging Interfaces

All Content Based Charging interfaces are defined in the file bea\w1ng22\esdk\idl\plugin_if\charging\ChargingResource_IF.idl

Out of the box, Content Based Charging plug-ins work with the following northbound interfaces:

- Parlay X 1.0 Payment
- Extended Web Services Content Based Charging in conjunction with the Charging Service Capability.
**Figure 12-1  Content based charging interface**

Implemented by the Content Based Charging SC

The following interface is implemented in the Content Based Charging service capability

**Table 12-1  Implemented by the Content Based Charging SC, used by a Content Based Charging plug-in.**

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IrAppChargingSession</td>
<td>Callback interface for receiving responses to operations performed via the IrChargingSession interface.</td>
</tr>
</tbody>
</table>
IrAppChargingSession

The IrAppChargingSession receives the results of various calls on the IrChargingSession implementation in the plug-in.

Table 12-2 IrAppChargingSession

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>creditAmountErr</td>
<td>Result of a failed creditAmountReq request. No amount was credited.</td>
</tr>
<tr>
<td>creditAmountRes</td>
<td>Result of a successful creditAmountReq request. Contains the information requested.</td>
</tr>
<tr>
<td>creditUnitErr</td>
<td>Result of a failed creditUnitReq request. No units were credited.</td>
</tr>
<tr>
<td>creditUnitRes</td>
<td>Result of a successful creditUnitReq request.</td>
</tr>
<tr>
<td>debitAmountErr</td>
<td>Result of a failed debitAmountReq request. No amount was debited.</td>
</tr>
<tr>
<td>debitAmountRes</td>
<td>Result of a successful debitAmountReq request.</td>
</tr>
<tr>
<td>debitUnitErr</td>
<td>Result of a failed debitUnitReq request. No units were debited.</td>
</tr>
<tr>
<td>debitUnitRes</td>
<td>Result of a successful debitUnitReq request.</td>
</tr>
<tr>
<td>directCreditAmountErr</td>
<td>Result of a failed directCreditAmountReq request. No amount was debited.</td>
</tr>
<tr>
<td>directCreditAmountRes</td>
<td>Result of a successful directCreditAmountReq request.</td>
</tr>
<tr>
<td>directCreditUnitErr</td>
<td>Result of a failed directCreditUnitReq request. No units were credited.</td>
</tr>
<tr>
<td>directCreditUnitRes</td>
<td>Result of a successful directCreditUnitReq request.</td>
</tr>
<tr>
<td>directDebitAmountErr</td>
<td>Result of a failed directDebitAmountReq request. No amount was credited.</td>
</tr>
<tr>
<td>directDebitAmountRes</td>
<td>Result of a successful directDebitAmountReq request.</td>
</tr>
<tr>
<td>directDebitUnitErr</td>
<td>Result of a failed directDebitUnitReq request. No units were debited.</td>
</tr>
<tr>
<td>directDebitUnitRes</td>
<td>Result of a successful directDebitUnitReq request.</td>
</tr>
<tr>
<td>extendLifeTimeErr</td>
<td>Result of a failed extendLifeTimeReq request. The lifetime was not extended.</td>
</tr>
</tbody>
</table>


The following interface is implemented in the Content Based Charging network plug-in

Table 12-3 Implemented by a plug-in for Content Based Charging

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource</td>
<td>Initial object obtained from Plug-in Manager. Base interface implemented by all plug-ins.</td>
</tr>
<tr>
<td>ChargingResource</td>
<td>Creates Charging sessions.</td>
</tr>
<tr>
<td>IrChargingSession</td>
<td>Provides functions to reserve, debit and credit user accounts. It also provides functions for rating requests. There are separate functions that operate on units, volumes and amounts.</td>
</tr>
</tbody>
</table>
ChargingResource

The Charging Resource implementation is a Manager object that creates Charging Sessions.

Table 12-4  ChargingResource

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>createChargingSessionCtx</td>
<td>Creates a Content Based Charging session.</td>
</tr>
<tr>
<td>getChargingSessionCtx</td>
<td>Retrieves an already created instance of a Charging session.</td>
</tr>
</tbody>
</table>

IrChargingSession

Table 12-5  IrChargingSession

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>creditAmountReq</td>
<td>Credits an amount for the reservation associated with the session.</td>
</tr>
<tr>
<td>creditUnitReq</td>
<td>Credits one or more units for the reservation associated with the session.</td>
</tr>
<tr>
<td>debitAmountReq</td>
<td>Debits an amount for the reservation associated with the session.</td>
</tr>
<tr>
<td>debitUnitReq</td>
<td>Debits one or more units towards the reservation associated with the session.</td>
</tr>
<tr>
<td>directCreditAmountReq</td>
<td>Credits an amount directly without affecting the reservation.</td>
</tr>
<tr>
<td>directCreditUnitReq</td>
<td>Credits one or more units directly without affecting the reservation.</td>
</tr>
<tr>
<td>directDebitAmountReq</td>
<td>Debits an amount directly without affecting the reservation.</td>
</tr>
<tr>
<td>directDebitUnitReq</td>
<td>Debits one or more units directly without affecting the reservation.</td>
</tr>
<tr>
<td>extendLifeTimeReq</td>
<td>Extends the lifetime of a reservation.</td>
</tr>
<tr>
<td>getAmountLeft</td>
<td>Gets the remaining amount in a reservation.</td>
</tr>
<tr>
<td>getLifeTimeLeft</td>
<td>Gets the remaining lifetime of a reservation.</td>
</tr>
<tr>
<td>getUnitLeft</td>
<td>Gets the remaining units of a reservation.</td>
</tr>
<tr>
<td>rateReq</td>
<td>Rates a request.</td>
</tr>
<tr>
<td>release</td>
<td>Releases a Content Based Charging session and any reserved amount or units left in the associated reservation.</td>
</tr>
</tbody>
</table>
Table 12-5  IrChargingSession

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reserveAmountReq</td>
<td>Reserves an amount from an account</td>
</tr>
<tr>
<td>reserveUnitReq</td>
<td>Reserves one or more units from an account.</td>
</tr>
</tbody>
</table>

Use case

The following sequence diagram shows a basic interaction between a Content Based Charging client (usually the Content Based Charging service capability) and a CBC plug-in.

**Note:** Content Based Charging only supports operations originating from an application. Network triggered is not supported.
Use case

Introduction and Developer Guide for the WebLogic Network Gatekeeper Extension SDK

Figure 12-2 Application-initiated usage of Content based charging plug-in.

Details about the sequence diagram:

1. Before the diagram starts, the Charging client (for example the Content Based Charging SC) has asked for and received a Content Based Charging plug-in. See “Typical Traffic Path for Application-Initiated Requests” on page 5-14 for more information on this process.

2. The Charging client calls `createChargingSessionCtx` on the implementation of the `ChargingResource` interface in the plug-in.

3. The plug-in creates an instance of the `IrChargingSession` interface and returns it to the client. The return also includes:
   - An identifier for the session
– A sequence number that is increased with each new request. This is used to make sure that the client’s requests are performed in the correct order.

4. The Content Based Charging client calls rateRequest on the Charging Session object. The returned rating lets the client know how much to reserve.

5. The reservation is performed via reserveAmountReq (not shown).

6. A successful reservation is reported via reserveAmountRes (not shown).

7. An amount is debited from the reservation using debitAmountReq.

8. The successful debit is reported via debitAmountRes.

9. If the debited amount is too large, the account is credited using creditAmountReq, and the successful credit is reported using creditAmountRes.

10. The lifetime of the reservation is extended via extendLifeTimeReq, and the successful request is reported via extendLifeTimeRes. If additional funds need to be reserved, reserveAmountReq can be called again.

11. When all activities for the session have been completed, the session is released and sessionEnded is returned.
Subscriber Profile

The following sections contain descriptions of the interface between the Subscriber Profile service capability and plug-ins of the Subscriber Profile type:

- **Subscriber Profile Interfaces**
- **Use case**

### Subscriber Profile Interfaces

All subscriber profile interfaces are defined in the file

```
bea\w1ng22\esdk\idl\plugin_if\sp\sp_interfaces.idl
```

Out of the box, Subscriber Profile plug-ins work with the following northbound interface:

- Extended Web Services Subscriber Profile

in conjunction with the Subscriber Profile Service Capability.
Implemented by the Subscriber Profile SC

The following interface is implemented in the Subscriber Profile service capability.

Table 13-1 Implemented by the Subscriber profile SC, used by a Subscriber Profile plug-in

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IrAppSubscriberProfile</td>
<td>Callback interface for receiving responses to operations performed via the</td>
</tr>
<tr>
<td></td>
<td><code>IrSubscriberProfile</code> interface.</td>
</tr>
</tbody>
</table>
**IrAppSubscriberProfile**

The **IrAppSubscriberProfile** interface listens for results from getter and setter operations on the Subscriber Profile object which can communicates, for example, with an external database.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getInfoPropertyRes</td>
<td>Result of a successful getInfoPropertyReq request. Contains the information requested.</td>
</tr>
<tr>
<td>getInfoPropertyErr</td>
<td>Result of a failed getInfoPropertyReq request.</td>
</tr>
<tr>
<td>setInfoPropertyRes</td>
<td>Verification of a successful setInfoPropertyReq request.</td>
</tr>
<tr>
<td>setInfoPropertyErr</td>
<td>Result of a failed setInfoPropertyReq request.</td>
</tr>
<tr>
<td>queryBalanceRes</td>
<td>Result of a successful queryBalanceReq request.</td>
</tr>
<tr>
<td>queryBalanceErr</td>
<td>Result of a failed setqueryBalanceReq request.</td>
</tr>
</tbody>
</table>

## Implemented by the Subscriber Profile plug-in

The following interfaces are implemented by the Subscriber Profile plug-in.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource</td>
<td>Initial object obtained from Plug-in Manager. Base interface implemented by all plug-ins.</td>
</tr>
<tr>
<td>IrSubscriberProfileResource</td>
<td>Creates sessions on the Subscriber Profile plug-in</td>
</tr>
<tr>
<td>IrSubscriberProfile</td>
<td>Gets and sets subscriber data and releases Subscriber Profile sessions.</td>
</tr>
<tr>
<td>IrSubscriberProfileSubscriptionExt</td>
<td>Provides extended functions for handling subscriptions. The Subscriber Profile SC does not operate on this interface, but other -custom- clients to a subscriber profile plug-in can use this; for example, a policy utility class might be a client.</td>
</tr>
</tbody>
</table>
**IrSubscriberProfileResource**

This manager interface is used to create a Subscriber Profile session.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getSubscriptionProfileCtx</td>
<td>Requests a subscriber profile. Creates the session.</td>
</tr>
</tbody>
</table>

**IrSubscriberProfile**

This interface gets and sets information about a subscriber.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getSubscriberId</td>
<td>Gets a subscriber ID based on the session id.</td>
</tr>
<tr>
<td>getInfoPropertyReq</td>
<td>Gets one or more properties for the subscriber.</td>
</tr>
<tr>
<td>setInfoPropertyReq</td>
<td>Sets one or more properties for the subscriber.</td>
</tr>
<tr>
<td>release</td>
<td>Releases a session.</td>
</tr>
<tr>
<td>queryBalanceReq</td>
<td>Gets the available balance for a subscriber.</td>
</tr>
</tbody>
</table>

**IrSubscriberProfileSubscriptionExt**

This interface offers a custom, non-SC client of the Subscriber Profile plug-in methods to get and set information about a subscription and to add and remove subscriptions.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getSubscriptionReq</td>
<td>Fetches data about a subscription.</td>
</tr>
<tr>
<td>setSubscriptionStateReq</td>
<td>Sets data about a subscription.</td>
</tr>
<tr>
<td>addSubscriptionReq</td>
<td>Adds a subscription.</td>
</tr>
<tr>
<td>removeSubscriptionReq</td>
<td>Removes a subscription.</td>
</tr>
</tbody>
</table>
Use case

Implemented by a customized non-SC client of the Subscriber Profile plug-in

The following interface is implemented by a non-SC client of the Subscriber Profile plug-in that wishes to use the extended functionality interface.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IrAppSubscriberProfileSubscriptionExt</td>
<td>Callback interface for receiving responses to operations performed via the IrSubscriberProfileSubscriptionExt interface.</td>
</tr>
</tbody>
</table>

IrAppSubscriberProfileSubscriptionExt

This interface offers non-SC clients methods to receive information it requests of the Subscriber Profile plug-in

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getSubscriptionRes</td>
<td>Result of a successful getSubscriptionReq request.</td>
</tr>
<tr>
<td>getSubscriptionErr</td>
<td>Result of a failed getSubscriptionReq request.</td>
</tr>
<tr>
<td>setSubscriptionStateRes</td>
<td>Result of a successful setSubscriptionStateReq request.</td>
</tr>
<tr>
<td>setSubscriptionStateErr</td>
<td>Result of a failed setSubscriptionStateReq request.</td>
</tr>
<tr>
<td>addSubscriptionRes</td>
<td>Result of a successful addSubscriptionReq request.</td>
</tr>
<tr>
<td>addSubscriptionErr</td>
<td>Result of a failed addSubscriptionReq request.</td>
</tr>
<tr>
<td>removeSubscriptionRes</td>
<td>Result of a successful removeSubscriptionReq request.</td>
</tr>
<tr>
<td>removeSubscriptionErr</td>
<td>Result of a failed removeSubscriptionReq request.</td>
</tr>
</tbody>
</table>

Use case

The following sequence diagram shows a basic interaction between a client of the Subscriber Profile service capability and a plug-in of the Subscriber Profile type. It also shows an additional set of interactions using the Extended interfaces.
**Note:** Subscriber Profile plug-ins only supports operations originating from an application. Network triggered operations are not supported.

**Figure 13-2 Application-initiated usage of Subscriber Profile plug-in.**

Details about the sequence diagram:

1. Before the diagram starts, the Subscriber Profile client (for example, the Subscriber Profile SC) has acquired a Subscriber Profile plug-in. See “Typical Traffic Path for Application-Initiated Requests” on page 5-14 for more information on this process.

2. The client calls the `getSubscriberProfileCtx` method of the object that implements the `IrSubscriberProfileResource` interface on the plug-in. The `IrSubscriberProfile` object is created, and an identifier for the session, along with the interface, is returned to the client.

3. The client requests information about the subscriber using `getInfoPropertyReq`. The result is returned asynchronously using the `IrAppSubscriberProfile` interface implemented by the client, using the `getInfoPropertyRes` method. If the plug-in experiences an error when getting the requested properties, the method `getInfoPropertyErr` is invoked instead. The data are given as name-value pairs.
4. The client sets information about the subscriber using `setInfoPropertyReq`. The data are sent as name-value pairs. The result is returned asynchronously using the `IrAppSubscriberProfile` interface implemented by the client, using the `setInfoPropertyRes` method. If the plug-in experiences an error when setting the requested properties, the method `setInfoPropertyErr` is invoked instead.

5. Custom Subscriber Profile clients can also get and set subscription information using the `IrSubscriberProfileSubscriptionExt` interface.

6. The client requests information on the subscriptions to which a user may be subscribed by calling `getSubscriptionReq` on the plug-in’s `IrSubscriberProfileSubscriptionExt` interface.

7. The response is sent back asynchronously using the `getSubscriptionRes` on the `IrAppSubscriberProfileSubscriptionExt` interface implemented by the custom client. If the plug-in experiences an error, `getSubscriptionErr` is invoked instead.

8. When the client’s interaction with the plug-in is finished, the client invokes `release` on the `IrSubscriberProfile` object and the session is destroyed. The plug-in has the responsibility for removing all objects.
User Location

The following sections contain descriptions of the interface between the User Location service capability and network plug-ins of the User Location type:

- User Location Interfaces
- Use cases

User Location Interfaces

All User Location interfaces are defined in the file `UlResource_IF.idl` in `bea\wlng22\esdk\idl\plugin_if\mobility`. The interfaces also use definitions in the files `UlResource_data.idl` and `MobilityResource_data.idl`.

Out of the box, User Location plug-ins work with the following northbound interfaces:

- Parlay X (1.0 and 2.1) Terminal Location
- Extended Web Services User Location Interaction

in conjunction with the User Location Service Capability.
Implemented by the User Location SC

The following interfaces are implemented by the User Location service capability:

Table 14-1 Implemented by the User Location SC; used by a User Location plug-in

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IrAppUserLocation</td>
<td>Callback interface for positioning results.</td>
</tr>
<tr>
<td>IrAppUserLocationExt</td>
<td>Extended callback interface for positioning results.</td>
</tr>
</tbody>
</table>
**IrAppUserLocation**

An object that implements `IrAppUserLocation` listens for events originating in the plug-in. There may be several listeners registered for each plug-in. To implement this functionality, however, you should use the `IrAppUserLocationExt` interface below.

This interface inherits from the SCS interface, which makes it possible to narrow a SC object retrieved from the SC Manager to an `IrAppUserLocation`.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>locationReportRes</td>
<td>Delivers a report containing locations for one or several terminals to the Service Capability implementation.</td>
</tr>
<tr>
<td>locationReportErr</td>
<td>Informs the Service Capability implementation that a location report request has failed.</td>
</tr>
<tr>
<td>extendedLocationReportRes</td>
<td>Delivers a report containing extended location information about one or several users to the Service Capability implementation.</td>
</tr>
<tr>
<td>extendedLocationReportErr</td>
<td>Informs the Service Capability implementation that an extended location report request has failed.</td>
</tr>
<tr>
<td>triggeredLocationReport</td>
<td>Delivers a report containing triggered location information about one or several terminals to the Service Capability implementation.</td>
</tr>
<tr>
<td>triggeredLocationReportErr</td>
<td>Informs the Service Capability implementation that a triggered location report request has failed.</td>
</tr>
</tbody>
</table>

**IrAppUserLocationExt**

An object that implements `IrAppUserLocationExt` listens for events originating in the plug-in. There may be several listeners registered for each plug-in. Because this interface extends the `IrAppUserLocation` interface, you should use this one for the functionality of both.
This interface inherits from the SCS interface, which makes it possible to narrow a SC object retrieved from the SC Manager to an \texttt{IrAppUserLocationExt}.

### Table 14-3 \texttt{IrAppUserLocationExt}

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>geoLocationReportRes</td>
<td>Delivers a report containing geo-locations for one or several terminals.</td>
</tr>
<tr>
<td>triggeredGeoLocationReport</td>
<td>Delivers a report containing geographical information on the position of one or several terminals. The position data is geographical data such as address, zip code and so on.</td>
</tr>
<tr>
<td>triggeredGeoLocationReportErr</td>
<td>Informs the Service Capability implementation that a triggered geographical location report request has failed.</td>
</tr>
<tr>
<td>periodicLocationReport</td>
<td>Delivers a periodic report containing locations for one or several terminals.</td>
</tr>
<tr>
<td>periodicLocationReportErr</td>
<td>Informs the Service Capability implementation that a periodic location report request has failed.</td>
</tr>
</tbody>
</table>

### Implemented by the User Location plug-in

The following interfaces are implemented by a User Location plug-in:

### Table 14-4 Implemented by a plug-in for User Location

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource</td>
<td>Initial object obtained from Plug-in Manager. Base interface implemented by all plug-ins.</td>
</tr>
<tr>
<td>\texttt{IrUserLocation}</td>
<td>Gets a terminal position and starts triggered and/or periodic location requests. Most of its methods are deprecated.</td>
</tr>
<tr>
<td>\texttt{IrUserLocationExt}</td>
<td>Gets the position for a terminal, starting triggered and period location request.</td>
</tr>
</tbody>
</table>
**IrUserLocation**

An object implementing `IrUserLocation` inherits from the Resource interface. Most methods are deprecated. Use `IrUserLocationExt` instead.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>locationReportReq</td>
<td>Deprecated.</td>
</tr>
<tr>
<td>extendedLocationReportReq</td>
<td>Deprecated.</td>
</tr>
<tr>
<td>triggeredLocationReportingStartReq</td>
<td>Deprecated.</td>
</tr>
<tr>
<td>triggeredLocationReportingStop</td>
<td>Stops a previously started triggered location report request.</td>
</tr>
</tbody>
</table>

**IrUserLocationExt**

`IrUserLocationExt` is an extended version of the `IrUserLocationExt` interface. Use this interface for the functionality of both.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>locationReportCtxReq</td>
<td>Requests a report on the location of one or several terminals</td>
</tr>
<tr>
<td>extendedLocationReportCtxReq</td>
<td>Requests a report on the location of one or several terminals. More information, such as altitude, can be provided in the response to the request that in the response to <code>locationReportCtxReq</code>.</td>
</tr>
<tr>
<td>triggeredLocationReportingStartCtxReq</td>
<td>Requests a triggered report - when a terminal enters or exits a specific location.</td>
</tr>
<tr>
<td>geoLocationReportReq</td>
<td>Requests a report on the location for one or several terminals where the result is delivered as geographical data such as address, zip code and so on.</td>
</tr>
<tr>
<td>periodicLocationReportingStartReq</td>
<td>Starts a periodic report on the location for one or several terminal. The desired interval between positioning reports is defined.</td>
</tr>
<tr>
<td>periodicLocationReportingStop</td>
<td>Stops a previously started periodic report.</td>
</tr>
</tbody>
</table>


Use cases

Application-initiated User Location Request

The following sequence diagram shows a basic positioning request and an extended user location request between a User Location client (for example ESPA User Location SC) and a plug-in.

Figure 14-2 Application-initiated send message

Details about the sequence diagram:

1. Before the diagram begins, the User Location client (usually the ESPA User Location SC) has acquired a User Location plug-in. See “Typical Traffic Path for Application-Initiated Requests” on page 5-14 for more information.

2. The User Location client makes a locationReportCtxReq call to the implementation of the IrUserLocationExt interface, which is implemented in the plug-in. The call-back interface, IrAppUserLocation, is provided in call.

3. The plug-in reports the location via locationReportRes. If the plug-in experiences problems getting the position, locationReportErr is used instead.

4. The User Location client makes an extendedLocationReportCtxReq call to the plug-in’s implementation of the IrUserLocationExt interface. The call-back interface, IrAppUserLocationExt, is provided in call. The response to extendedLocationReportCtxReq provides more detailed information such as altitude, assuming the underlying network supports such information.
5. The plug-in reports the location via extendedLocationReportRes. If the plug-in experienced problems getting the position, extendedLocationReportErr is used.

**Network-triggered user location request**

The standard Network Gatekeeper implementation of the ESPA User Location SC does not support network triggered events. A custom User Location implementation could, however, benefit from the network triggered parts of the User Location plug-in interface.

The network triggered part of User Location uses the IrUserLocationExt interface, and starts a triggered location report using triggeredLocationReportingStartCtxReq. Supplied with this request is a callback interface and the area of interest, together with information on whether the information should be supplied when the terminal enters or leaves the specified area.

**Note:** The area can be either defined by longitude and latitude or by an abstract geographical area, such as an a street or city. The latter assumes access to a geographic information system.

The responses to the triggered user location request are sent using the triggeredGeoLocationReport method of the IrAppUserLocationExt interface, or the triggeredLocationReport method of the IrAppUserLocation interface.
User Location
Plug-ins that execute as both a SLEE service and a web application

The following sections contain descriptions of plug-ins that uses or exposes Web Services:

- Overview
- The Stages of Interaction

Overview

Because many underlying network nodes expose Web Services (SOAP/HTML) or other HTTP based protocols to communicate with external requests, Network Gatekeeper supports using the Tomcat servlet engine to handle the HTTP part of such interactions. This section describes setting up the relationship between the Tomcat instance and other SLEE services, including normal plug-in functionality. A plug-in that uses the web application functions of the Tomcat servlet engine must be divided into two parts, one part that executes as a regular SLEE service and one part that executes as a web application in Tomcat. Tomcat is itself deployed as the SLEE service, Embedded_Tomcat.

A southbound, network-facing layer that uses HTTP behaves in a similar manner to a northbound, application-facing HTTP based layer - interaction between the Web Services aspect of the layer and the regular SLEE Service part is mediated using the registration mechanism of the SLEE Common Loader. (For more on the use of the SLEE Common Loader in the northbound layer, see “Web Services layer to SESPA” on page 5-2.) The web specific functionality for the plug-in resides in a web application in the form of a servlet or Web Service that is deployed in the Embedded_Tomcat instance. The AXIS classes necessary for SOAP based processing are a part of the Embedded_Tomcat classpath rather than a part of the SLEE classpath.
Plug-ins that execute as both a SLEE service and a web application

Because of the classloader hierarchy in the Network Gatekeeper, the SLEE service part of the plug-in must register the appropriate interfaces on both sides in the SLEE Common Loader, after which they can be accessed, each by the other.

**Figure 15-1  Classloader hierarchy**

```
+-----------------------------+
| SLEE                       |
+-----------------------------+
| SLEE Common Loader         |
| +---------------------------+
| | SLEE Services             |
| | +--------------------------+
| | | Embedded Tomcat           |
| | | +--------------------------+
| | | Web Applications/  |
| | | | Web Services            |
```

**The Stages of Interaction**

**Interface class registration**

The SLEE service part of the plug-in must register all the appropriate class interfaces with the SLEE Common Loader before they can be used. If one part of the plug-in attempts to use some aspect of the other side of the plug-in before this occurs, a ClassCastException will occur.

In Listing 15-1 the jar files that contain the interface definitions of both the “receiver” (in the regular SLEE service part) and the “sender” (in the Embedded_Tomcat part) and their fully qualified classnames are registered in the SLEE Common Loader. In this example, m_sc is the Service Context object provided by the SLEE when the plug-in enters the Started state. (For more on the process by which a SLEE service acquires a Service Context object, see “Implementing SLEE Service Behavior” on page 4-5)
Listing 15-1  Registering the class interfaces

```java
String mySOAPReceiver_if = “com.acme.MySOAPReceiver_if”;  
String JarName = m_sc.getJarName;  
SleeCommonLoader.getInstance().registerClass(JarName, mySOAPReceiver_if);  
String mySOAPSender_if = “com.acme.MySOAPSender_if”;  
String JarName = m_sc.getJarName;  
SleeCommonLoader.getInstance().registerClass(JarName, mySOAPSender_if);
```

Processing incoming requests

For requests that originate in the network, the plug-in must implement a two part server. In this case, the regular SLEE service part of the plug-in must be instantiated and added to the SLEE Common Loader so that the web application part of the plug-in can use it.

SLEE Service part

As the plug-in SLEE service enters the Activated state, the SLEE service part of the plug-in is instantiated and then added to the SLEE Common Loader as in Listing 15-2. An ID is provided when registering the object. This ID is what is used by the web application part to interact with the regular SLEE service part of the plug-in.

**Note:** The object must be added to the SLEE Common Loader immediately after it has been instantiated.

Listing 15-2  Add the implementation of the regular SLEE service interface to the SLEE Common Loader

```java
soapReceiver = new MySOAPReceiver_impl(TheContext.getServiceContext(), this);
```
Plug-ins that execute as both a SLEE service and a web application

SleeCommonLoader.getInstance().addObject(MY_SLEESERVICE_OBJECT_ID, soapReceiver);

Web Application part

The web application part of the plug-in gets the object that implements the SLEE service part of the plug-in from the SLEE Common Loader, using the ID with which the SLEE service part was registered. The fetched object is then cast to the correct class, as in Listing 15-3. The object must be retrieved for each request.

Listing 15-3 Fetch the interface from the SLEE Common Loader

Object obj = SleeCommonLoader.getInstance().getObject(MY_SLEESERVICEOBJECT_ID);
MySOAPReciever_if mySOAPReciever_if = (MySOAPReciever_if) obj;

Because the web application part of the plug-in must fetch the object from the SLEE Common Loader, the object must have been instantiated and registered in the SLEE Common Loader before the getInstance call. This means that the web application part of the plug-in must always start after the part that executes as a SLEE Service. This is done automatically as regular SLEE services always are started prior to the Embedded_Tomcat instance.

Processing outgoing requests

For requests that originate in the Network Gatekeeper, the plug-in must implement a two part client. In this case, the Embedded_Tomcat part of the plug-in must be instantiated and added to the SLEE Common Loader so that the regular SLEE services part of the plug-in can use it.

Web Application part

The web application part of the plug-in is responsible for constructing and performing the HTTP requests. If the particular network element requires SOAP, this part of the plug-in is also responsible for creating the SOAP message, using the AXIS classes available for applications running in the Embedded_Tomcat instance.

When the web application is instantiated, it adds itself along with an ID to the SLEE Common Loader, as in Listing 15-4.
The Stages of Interaction

Listing 15-4  Add the implementation of the interface to the SLEE Common Loader

```java
soapSender = new MySOAPSender_impl(TheContext.getServiceContext(), this);
SleeCommonLoader.getInstance().addObject(MY_SERVLET_OBJECT_ID, soapSender);
```

The web application part of the plug-in must be registered in the SLEE Common Loader before the SLEE service part tries to use the interface. Because the normal order has SLEE services started before the Embedded_Tomcat instance, specific code must be added to insure that the web app has been registered before calling `getObject`. One way to achieve this is to add a method in the interface used by the web application part (and implemented in the SLEE Service part of the plug-in) that notifies the SLEE service part of the plug-in that the web app has started, and have the web application part of the plug-in always call this method when it goes into the Active state.

**SLEE service part**

For outgoing requests, the SLEE service part of the plug-in uses the interface class registered by the web application part of the plug-in.

This means that it should fetch the object via the SLEE Common Loader and cast it to the correct class, as in **Listing 15-5**. The object must be retrieved for each request.

Listing 15-5  Fetch the interface from the SLEE Common Loader

```java
Object obj = SleeCommonLoader.getInstance().getObject(MY_SERVLET_OBJECT_ID);
MySOAPSender_if mySOAPSender_if = (MySOAPSender_if) obj;
```

This also means that the SLEE service part of the plug-in must wait until the part executing in Embedded_Tomcat has registered the implementing class in the SLEE Common loader.
Plug-ins that execute as both a SLEE service and a web application
Policy rules and Policy Utilities

For most installations of WebLogic Network Gatekeeper, the ability to rapidly and accurately evaluate the status of requests in terms of Policy, or rules governing access and other service characteristics, is one of the most important features that the system offers. If you extend the Network Gatekeeper, particularly if you add a new traffic path, you may also need to make changes in the Policy system to cover new functionality that you have added. This chapter extends the information covered in “Using the PolicyManager” on page 4-36 by taking a look at the process as it takes place within the Policy Engine, with a very high level description of the process by which policy requests are processed and though which new rules can be added. It covers:

- “PolicyRequest data” on page 16-1
- “Adding a new rule to Policy Decision Point” on page 16-3
- “Extending Service Level Agreements” on page 16-7
- “Defining a Policy Utility class” on page 16-11

PolicyRequest data

Data arrives at the Policy Engine from the Policy Enforcement Point (PEP) in the ESPA SC bundled in the form of a PolicyRequest object. These values must be mapped to the variables in the Policy Rule that will be used to evaluate them. The Policy Request object can contain subsets of this standard data:

- applicationID: The Application ID of the requesting party
Policy rules and Policy Utilities

- serviceProviderID: The Service Provider ID of the requesting party
- nodeID: Used internally by Network Gatekeeper - ignore
- serviceName: The name of the software module in which the policy request originates. Used in the rules to match to service contracts in the SLAs and to look-up any rules specific to the service.
- methodName: The name of the method which the request wishes to have executed. Access to this method is what is being evaluated. To see the standard mapping between northbound interface method requests and their PEP equivalents, see “Appendix F: Policy Enforcement Points” in the System Administrator’s Guide.
- serviceGroup: This value must be “ espa”
- serviceCode: The service code provided by the application, which is written to CDRs for tracking purposes
- requesterID: An additional ID that may be provided by the application for tracking purposes. (dependent on the northbound interface being used)

All of the above are Strings.

- transactionID: Used internally by Network Gatekeeper - ignore
- noOfActiveSessions: Used internally by Network Gatekeeper - ignore
- timeStamp: The time the request was fed to the Policy Engine. Use the SLEE Time manager to get this timestamp.
- reqCounter: The number of target addresses in the request. If only one target address is used in the request set this value to 1. If using multiple target addresses in the request, use the number of target addresses.

All of these are Longs.

In addition to these standard values, Policy Request objects can contain AdditionalParameters, which is an array of AdditionalDataValue. This data type provides a mechanism for the transferring request data other than that which is predefined for the Policy Decision Point. An AdditionalDataValue consist of a name-value pair. The following data types can be defined in an AdditionalDataValue object.

- intValue(int val): Integer values
- longValue(long val): Long values
- `stringValue(String val)`: Strings.
- `stringArrayValue(String[] val)`: Arrays of String values.
- `booleanValue(boolean val)`: Boolean values.
- `shortValue(short val)`: Short values.
- `charValue(char val)`: Char values.
- `floatValue(float val)`: Float values.
- `doubleValue(double val)`: Double values.
- `intArrayValue(int[] val)`: Arrays of int values.

The name of the name-value pair is defined in the `dataName` member variable in the `AdditionalData` object. See Listing 16-1

**Listing 16-1  Defining AdditionalData**

```java
AdditionalData adArray[] = new AdditionalData[1];
AdditionalDataValue targetAddressValue = new AdditionalDataValue();
AdditionalData adTargetAddressString = new AdditionalData();
targetAddressValue.stringValue(address);
adTargetAddressString.dataName = "targetAddress";
adTargetAddressString.dataValue = targetAddressValue;
adArray[0] = adTargetAddressString;
policyRequest.additionalParameters = adArray;
```

See “Adding a new rule to Policy Decision Point” on page 16-3 for information on using the `PolicyRequest` object in the rule.

**Adding a new rule to Policy Decision Point**

New rules can be added to the Policy Decision Point (PDP) in the Policy Service, using the Ilog IRL language. The rule must have a name and a priority.
High priority rules are evaluated before low priority rules. There are a set of pre-defined priority levels, which are mapped to a numerical value:

- minimum, where the value is $-1 \times 10^9$
- low, where the value is $-1 \times 10^6$
- high, where the value is $1 \times 10^6$
- maximum, where the value is $1 \times 10^9$

Listing 16-2 shows the basic structure of a rule:

```
Listing 16-2  Skeleton of a rule

rule DenySubscriberNotExists
{
    priority = high;
    when
    {
        // fetch the policy request data and perform evaluations.
    }
    then
    {
        // Take action on
    }
};
```

Getting data defined in the PolicyRequest

In order to perform an evaluation, the data in the PolicyRequest object must be fetched by the rule in the Policy Engine and mapped to the equivalent variable name in the rule. The standard types of request data in the Policy Request are associated with variables of the same name in the
rules. Below is an example of a rule assigning the PolicyRequest member variable serviceName to the rule variable sname via the Policy Request object. The rule object pr is assigned to the PolicyRequest object.

**Listing 16-3  Policy Request data is fetched**

```java
?pr: event PolicyRequest(?sname: serviceName);
```

If the Policy Engine has evaluated the request and made the decision to deny it, the Policy Engine’s representation of the PolicyRequest object (pr) must be retracted. Retracting the PolicyRequest object aborts further rule enforcement.

**Listing 16-4  Retract a request**

```java
retract (?pr);
```

If the Policy Engine has evaluated the request and made the decision to allow it, the Policy Engine’s representation of the request (pr) must still be retracted, but in the last rule of the execution flow. For example, this could be achieved by adding a general finalizing allow rule that retracts the request. This rule should have priority minimum.

**Listing 16-5  General finalizing allow rule that retracts a request**

```java
rule AllowServiceRequest
{
    priority = minimum;
    when
    {
        ?pr: event PolicyRequest();
    }
```
Data that is defined as AdditionalValues must fetched as shown in Listing 16-6. The Additional Value named targetAddress is stored in the variable addDataValue. The PolicyRequest object is pr.

**Listing 16-6  Fetching AdditionalValue data**

```java
bind ?addDataValue = ?pr.getAdditionalDataStringValue("targetAddress");
```

The particular signature of the fetching method depends on the type of data:

- `getAdditionalDataIntValue(...)`, for int values
- `getAdditionalDataLongValue(...)`, for long value.
- `getAdditionalDataStringValue(...)`, for String values
- `getAdditionalDataStringArrayValue(...)`, for arrays of String values
- `getAdditionalDataBooleanValue(...)`, for boolean values
- `getAdditionalDataShortValue(...)`, for short values
- `getAdditionalDataCharValue(...)`, for char values
- `getAdditionalDataFloatValue(...)`, for float values
getAdditionalDataDoubleValue(...), for double values

getAdditionalDataIntArrayValue(...) for arrays of int values.

If the data type is unknown, it can be determined by invoking the discriminator method on the AdditionalDataValue object.

Listing 16-7  Determine the type of an AdditionalDatavalue

bind ?type = ?pr.getAdditionalData.dataValue.discriminator().value();

Where type is one of the following:

- AdditionalDataType._P_ADDITIONAL_INT
- AdditionalDataType._P_ADDITIONAL_LONG
- AdditionalDataType._P_ADDITIONAL_STRING
- AdditionalDataType._P_ADDITIONAL_STRING_ARRAY
- AdditionalDataType._P_ADDITIONAL_BOOLEAN
- AdditionalDataType._P_ADDITIONAL_SHORT
- AdditionalDataType._P_ADDITIONAL_CHAR
- AdditionalDataType._P_ADDITIONAL_FLOAT
- AdditionalDataType._P_ADDITIONAL_DOUBLE
- AdditionalDataType._P_ADDITIONAL_INT_ARRAY

See also the JavaDoc for PolicyRequest.

Extending Service Level Agreements

Service Level Agreements (SLAs) are XML files that contain the data which is enforced by Policy Engine using the rules. To understand more about the types of SLAs, see “Managing Application Service Providers” in WebLogic Network Gatekeeper Architectural Overview. The Service Level Agreements are created and loaded into the Policy Engine for these levels:

- Service Provider
- Application
Policy rules and Policy Utilities

- Service Provider traffic
- Total traffic

There are separate Policy rules that enforce these SLAs, one on the service provider level and one on application level. The traffic SLAs are both enforced using one rule.

To extend Service Level Agreements, the following steps must be taken:

6. “Update the SLA Schema” on page 16-8
7. “Load the new SLA schema into the Policy Service” on page 16-8
8. “Update and reload the rule files” on page 16-9
9. “Update the appropriate SLAs” on page 16-10
10. “Load the updated SLAs” on page 16-10

Update the SLA Schema

The SLA schema files are located in

<installation directory>/bin/policy/sla_schema.

There are three different schema files, based on the SLA type:

- app_sla_file.xsd: Defines the SLA schema for the application level SLAs.
- sp_sla_file.xsd: Defines the SLA schema for the service provider level SLAs.
- node_sla_file.xsd: Defines the SLA schema for the service provider traffic and total traffic SLA.

Update the appropriate schema file by adding appropriate elements. For example, if the service provider SLA schema needs a new element to define a String value listed in the service provider SLA as `<additionalData>mydata</additionalData>`, update the `serviceContract` in the schema file with the following element:

```xml
<xs:element name="additionalData" minOccurs="0" maxOccurs="1"
type="xs:int"/>
```

Load the new SLA schema into the Policy Service

Load the updated SLA schema into the Policy Service using the Network Gatekeeper Management Tool. The following methods in the Policy service are used:

reloadApplicationXmlDriver: Reloads application level SLA schemas.
**reloadServiceProviderXmlDriver**: Reloads service provider level SLA schemas.

**reloadNodeXmlDriver**: Reloads service provider traffic and total traffic SLA schemas.

### Update and reload the rule files

In order to enforce the new data in the SLA, the rule files must also be updated. The data in the SLA is fetched from an object model the Policy Engine creates from the data defined in the SLAs. The data in an SLA is fetched from the Policy Rules by name. For example, if a tag in the SLA is `<additionalData>`, the data is fetched using the same name, as described below.

The rule gets the parameter `aParam` from the Policy Request object, and puts it in the local variable `pr`. The parameter `aParam` is compared with the data fetched from the SLA. The request is denied if the parameter given in the SLA is larger than the parameter provided in the Policy Request.

#### Listing 16-8  Get SLA data and compare with a parameter in a Policy Request

```java
rule denyAParamValueNotAllowed
{
    priority = high;
    when
    {
        ?pr: event PolicyRequest(?serviceName: serviceName;
            ?aParam: aParam);
        ?sc: ServiceContract(?scs: scs;
            ?scs equals ?serviceName;
            ?aParam > additionalData);
    }
    then
    {
        retract (?pr);
        ?pr.deny("The parameter is not allowed!");
    }
```

The updated rules must also be loaded into the Policy Engine: this is performed with the Network Gatekeeper Management Tool using the following methods in the Policy Service:

- **loadApplicationRules**: Reloads application level rules.
- **loadServiceProviderRules**: Reloads service provider level rules.
- **loadNodeRules**: Reloads service provider traffic and total traffic rules.

**Update the appropriate SLAs**

If the schema is updated, the SLAs themselves must also be updated to contain any new elements and data. For example:

```xml
<additionalData>mydata</additionalData>
```

**Load the updated SLAs**

The SLAs holding the new parameters must be loaded into the Policy Engine. See “Enabling Service Providers and Applications” in the Network Gatekeeper System Administrator’s Guide for information on loading updated SLAs.

**Using a Policy Utility object**

The rule language, Ilog IRL, makes it possible to perform basic Policy evaluation and parameter substitution, but if your updated SLAs require more complex processing, call-outs to Java object may be necessary. A Policy Utility is a Java object that is used from a rule. Java objects can also be used for interaction with external systems such as databases or prepaid systems.

**Listing 16-9** shows a Policy Utility object being called from a rule file.

**Listing 16-9   Invoking a Policy Utility from a rule**

```java
if (MyPolicyUtility.getInstance().subscriberExists(?address,
    ?sp,
```
Defining a Policy Utility class

A Policy utility class has the following characteristics:

- It executes as a SLEE service
- It is a Singleton class

In order to be callable from the rules, a Policy Utility class must execute as a singleton. The Policy Utility registers its classes in the Policy Service, where they are instantiated and executed. As a consequence of the classloader hierarchy, two consecutive restarts of the SLEE in which the Policy utility is installed are required to load the Policy Utility - in the first one the Policy Utility installs itself in the SLEE and in the second the Policy Utility is loaded into the Policy Service.

When the Policy Utility is instantiated, it registers itself in the Policy Service. The constructor must be private as it is a singleton class. A static public class is created to get the Policy Utility Class from the rule, and to instantiate the class if necessary.

Since the Policy Utility class executes in the Policy service, it fetches the Service Context for the Policy via reflection.

Listing 16-10  Registering the Policy Utility class

```java
Class internalPolicyContextClass =
Class.forName("com.incomit.policy.InternalPolicyContext");
java.lang.reflect.Method getServiceContextMethod =
```
internalPolicyContextClass.getMethod("getServiceContext", null);
java.lang.Object serviceContextResultObj =
getServiceContextMethod.invoke(null, null);
m_sc = (ServiceContext) serviceContextResultObj;

The Policy Utility uses the same POA as the Policy Service. The POA is only necessary when the Policy Utility creates new CORBA objects such as, for example, listener objects when connecting to a plug-in.

Listing 16-11  Getting the POA in a Policy Utility class

java.lang.reflect.Method getChildPOAMethod =
internalPolicyContextClass.getMethod("getChildPOA", null);
java.lang.Object poaResultObj = getChildPOAMethod.invoke(null, null);
m_poa = (org.omg.PortableServer.POA) poaResultObj;

The rule calls a method in the singleton class via a static method that checks if the class is already instantiated, instantiates it if necessary, and returns the object. In the example in Listing 16-9 this method is named getInstance().

An Example of a Policy Utility

To see an example of a Policy Utility, look in the example template modules in the \module_templates\policy_utility directory. This shows a Policy Utility class, invoked from a rule, which checks if a given address parameter is present in an external subscriber database.
CHAPTER 17

Hints and Tips

The following section gives a summary of hints, tips, and recommendations for creating extensions to Network Gatekeeper that are based on practical experience in the field. They are linked to further discussions of the concepts involved elsewhere in this document, as needed.

- “General SLEE Managers” on page 17-1
  - “Using the SLEE Task Manager” on page 17-2
  - “Using the SLEE ID Managers” on page 17-2
- “Synchronous Interface/Asynchronous Implementation” on page 17-2
- “Time-outs” on page 17-4
- “The Tracing and Alarm Services” on page 17-5
- “SLEE Service and Web Archive (WS layer) interaction using the SLEE common loader” on page 17-6
- “Web Services and WSDD/WSDL” on page 17-8
- “Networking” on page 17-10

**General SLEE Managers**

Two important notions are connected to services fetched from the SLEE Context object.
Hints and Tips

### Using the SLEE Task Manager

Good practices regarding the SLEE Task Manager:

- Use the `scheduleSLEETask` method of the SLEE Task Manager to fetch threads from the SLEE thread pool. Do not create Java threads directly. See “SLEE Task Manager” on page 4-14 for more information.

- Do not schedule a task from another task unless this is explicitly required.

- Do not use one SLEE task for several purposes.

### Using the SLEE ID Managers

Good practices regarding creating unique IDs.

- Use the two SLEE ID Managers whenever you need to create an ID. There are two ID managers: the SLEE ID Manager, which returns a `long` value, and the SLEE Cyclic ID Manager, which returns an `int` value. For more information see “SLEE ID Manager” on page 4-17 and “SLEE Cyclic ID Manager” on page 4-16.

- Do not implement your own ID allocation logic.

- IDs retrieved from the SLEE ID managers are guaranteed to be unique within a cluster.

### Synchronous Interface/Asynchronous Implementation

Synchronous calls block until the result value is available to be returned as a return parameter. For example, in Figure 17-1, the thread making the synchronous call `getName` will be blocked until the result value (`name`) is retrieved.

**Figure 17-1** Synchronous call

[Diagram of synchronous call]
An asynchronous version of the same call delivers the result in a separate method call, as in Figure 17-2. The `getName` method now returns immediately, with either a void or a correlator of some sort as its result, while the `name` value is delivered in the `myNameIs` method at some later time.

**Figure 17-2  Asynchronous call**

Because asynchronous calls return immediately, they are a more efficient use of system resources, and are to be preferred, particularly because any given request must be handled by multiple layers within the Network Gatekeeper. So even if an interface is designed to be synchronous (with regard to the application or the network), the implementation of that interface should still process the request asynchronously internally, using multiple threads during the processing of the request. Only lock the top level thread.

This asynchronous implementation of a synchronous interface is accomplished by having the thread that receives the request hand over the processing of it to another thread, which may in turn hand over parts of it to yet other threads. The hand over from one thread to another is typically done either by using the SLEE Task Manager to schedule a SLEE Task or by calling a CORBA object with an asynchronous interface.

The thread receiving the initial request cannot, however, return until the result is available, so it must block. In order to block a thread it is possible to call `wait` on a java object. (See “Synchronous application-initiated sequence” on page 3-7 for an illustration). The `wait` call will block the calling thread until:

- the `wait` call times out (a `wait` call can and always should be associated with a configurable time out).
- another thread calls the `notify` method on the same object.
**Time-outs**

The default Web Service request time-out in Network Gatekeeper is set to 30 seconds. In some situations this value may be too large. It is possible to use a configurable time-out and override the default value of 30 seconds. This will be necessary in situations where it is not possible to wait up to 30 seconds.

**Note:** Using a request rate of 100 requests/second could result in (worst case scenario) 100 * 30 = 3000 requests being blocked in the Web Service call. This could potentially block a lot of threads.

The current AXIS libraries used in the Network Gatekeeper do not make it possible to specify separate Timeout/ConnectionTimeout values, so any configured time-out value will be used for both.

Listing 17-1 is an example of setting the time-out for a service named Foo.

**Listing 17-1  Set a time-out**

```java
FooService binding = locator.getFooService();
org.apache.axis.client.Stub s = (Stub) binding;
s.setTimeout(pluginContext.getWSTimeout());  // timeoutvalue in milliseconds
binding.doRequest("Hello");
```

Default HTTP related connection settings for Network Gatekeeper are:

- **MaximumTotalConnections** (For HTTP 1.1 keep-alive pool) = 200
- **MaximumConnectionsPerHost** (For HTTP 1.1 keep-alive pool) = 50

**Note:** These first two bullets refer to the AXIS generated Locator object. One Locator instance has one pool. Use only one Locator instance per interface.

- **ConnectionPoolTimeout** (For HTTP 1.1 keep-alive pool) = 0
- **DefaultConnectionTimeout = 3000 ms**
- **DefaultSoTimeout = 30 000 ms**
The above specified time-out values are used for all Web Service calls unless they are explicitly overridden.

Default CORBA related time-out values are defined in the Network Gatekeeper configuration file `slee_properties.xml`. The time-out values include:

- Connection time-out
- Request time-out

The above specified time-out values will be used for all CORBA calls unless they are explicitly overridden.

## The Tracing and Alarm Services

Hints for using the tracing and alarm SLEE services.

### Tracing

It is important to include information about the current context in trace. For example:

- Notification endpoint
- Session ID
- Transaction ID
- Message IDs (in the case of Messaging traffic)
- Any other relevant data that may be available in the current context.

Note that the following properties are automatically included, assuming it has been added to the Trace Context:

- Service provider ID
- Application ID
- Transaction ID
- Addresses
Hints and Tips

Alarms

Alarm IDs are specified in the OAM IDL file for each SLEE service. The alarm offset for a module is also specified in the IDL file.

If you develop a SLEE service, you must maintain a list of your alarms. For more information on Alarms, see “Using the Alarm Service” on page 4-34.

The severity of an alarm can be:

- CRITICAL - A severe error has occurred causing all services in a SLEE to fail.
  Can be generated only from core SLEE modules or customized supervision services. They are never generated from a plug-in. An example would be no access to the database.

- MAJOR - An error has occurred causing all requests to a specific SLEE service to fail.
  Can be generated from any type of SLEE service. An example would be a plug-in that cannot connect to the underlying network node.

- MINOR - An error has occurred affecting only a single request.
  Can be generated from any type of SLEE service. An example would be an inability to parse mandatory data element in incoming mobile originated request.

- WARNING - Something bad may be happening…
  Can be generated from any type of SLEE service. An example would be an inability to read “some data” from “somewhere”, falling back to default values.

It is important that you include information from the current context whenever an alarm is generated. Generating an alarm saying “Failed to send message.” is not particularly useful if there is a lot of traffic on the system from different applications. Always try to include information like message ID, session ID, application ID, service provider ID, target address, the URL that failed, and so on.

SLEE Service and Web Archive (WS layer) interaction using the SLEE common loader

The Network Gatekeeper uses a class loader hierarchy and you must understand the following implications of this:

- The class com.foo.MyClass loaded by class loader X it NOT the same as class com.foo.MyClass loaded by class loader Y.
The class loader that loads a class is important if you are passing an object instance of a particular class between different contexts, such as a SLEE service and a web archive (WS layer).

- All classes in a SLEE service will be loaded by a dedicated class loader for that SLEE service.
- It is possible to specify a parent class loader service. If a class can be found by the class loader for the parent SLEE service that class will be used.

If you are using the SLEE Common Loader to register classes and objects so that they can be shared between the web context and the SLEE service context, your SLEE service must specify `Slee_common_loader` as the parent class loader service. See Listing 17-2. For more information on this registration process, see “Web Services layer to SESPA” on page 5-2 and “Interface class registration” on page 15-2

Listing 17-2  Example deployment descriptor

```xml
<?xml version='1.0' encoding='us-ascii'?>
<SLEE_SERVICE
    name="Plugin_myplugin"
    version="1.0"
    max_alarms="3"
    company="BEA"
    trace="ON"
    parent_class_loader_service="Slee_common_loader">
    <SERVICE_DEPLOYABLE>
        com.accompany.plugin.myplugintype.MyPluginSleeService
    </SERVICE_DEPLOYABLE>
    <SERVICE_ACCESSIBLE>
        com.accompany.plugin.myplugintype.MyPluginSleeService
    </SERVICE_ACCESSIBLE>
    <SERVICE_MANAGEABLE>
```
Hints and Tips

com.acompany.plugin.myplugintype.MyPluginOAMImpl

<SERVICE_MANAGEABLE_IDL>
  MyPluginOAM.idl
</SERVICE_MANAGEABLE_IDL>
</SERVICE_MANAGEABLE>
</SLEE_SERVICE>

Version Handling and the SLEE Common Loader

When an installation upgrades a SLEE service that registers classes in the SLEE Common Loader, the SLEE that hosts the service should be restarted. This is also the case when a Policy Utility class is upgraded.

Web Services and WSDD/WSDL

When you implement the server side of a Web Service, you must specify a valid server-config.wsdd file. The class name of the class implementing the web service interface must be specified in that WSD file. The line specifying the class name typically looks something like:

<parameter name="className"
  value="com.bea.wlcp.wlng.resources.messaging.px2.sms.web.impl.SmsNotificationBindingImpl"/>

Note: The following assumes that you start with the plug-in template that comes with the Extension Toolkit.

To create the correct WSDD files:

1. Compile the web_impl module in the normal way. A server-config.wsdd file is generated in the generated_wsdd directory.
2. Update the generated server-config.wsdd file and save it in the deploy directory, which is the directory that contains the web.xml file.
3. Add the server-deploy.wsdd file to CVS (or whatever source control system being used).
4. Update the war target in the build.xml file. See Listing 17-3. It should be updated to include:
- The server-config.wsdd file from the deploy directory, not the generated_wsdd directory.
- The WSDL files.

**Listing 17-3  Example from war target of updated build.xml file**

```xml
<copy
todir="${module.lib.dir}/axis/WEB-INF"
    file="deploy/server-config.wsdd">
</copy>
<copy todir="${module.lib.dir}/axis/wsdl">
  <fileset dir="${pipe.root.dir}/parlayx2_if/wsdl">
    <include name="NetworkTriggeredEventListener.wsdl"/>
  </fileset>
</copy>
```

**Note:** While the build file will continue to generate a new server-config.wsdd to the generated_wsdd directory each time you build, the file in the deploy folder is the one that will be used.

You can verify that this works by browsing to the URL of your Web Service (for example, http://127.0.0.1:8080/wplugin) and then clicking on the view link. This should display a Web Page similar to that in **Listing 17-4**.

**Listing 17-4  Example of output when browsing to a Web Service**

And now... Some Services

AdminService (wsdl)
  - AdminService

Version (wsdl)
  - getVersion
NetworkTriggeredEventListener (wsdl)
  o myDeliverNetworkTriggeredEventMethod
  o myDeliverNetworkTriggeredEventMethod

You should be able to click the wsdl link and see the WSDL.

While it is possible to use the ns2pkg mapping file to map the namespace of the Web Service to the java package where you have implemented the Web Service (and have your implementing class be named accordingly), recommended practice is not to use this approach. We should keep the files generated from the standard WSDL files separated from the implementation files. It must be clear if the file is generated from the standard WSDL file or if it is an implementation file.

Networking
Additional information about networking.

Locator Classes
Never create more than one instance of your locator class. Use a static member variable. The locator class is thread safe. In HTTP 1.1 creating multiple instances of the locator creates new connections for each request, while old connections are kept alive. This is obviously not a good idea.

On the other hand, create a new instance of the stub for each request using the locator. These stubs are not thread safe.

CORBA Object Activation
When you need to create CORBA objects in your plug-in, these objects need to be associated with the correct POA. In the class that extends PluginSleeService there is a method doSetPOA. This method creates a POA that you should use (the code creating the POA is in the template already). Note that this POA is associated with a property called “implicit activation”, meaning that the CORBA object will be automatically activated when you do a myImplObject._this(m_sc.getSLEEContext().getORB()) call. The POA is stored in the m_context variable (i.e. your plug-in context). In order to associate an implementation of a CORBA object with a POA all objects implementing a CORBA object should have the following method:
public org.omg.PortableServer.POA _default_POA() {
    return m_context.getPOA();
}

Where m_context is the plug-in context.
For example, if you are implementing the IrCall interface with a class named IrCall_impl you would have something like Listing 17-5:

Listing 17-5   Getting the POA

public class IrCall_impl extends IrCallPOA{

    ...

    ...

    public org.omg.PortableServer.POA _default_POA() {
        return m_context.getPOA();
    }

}

Creating a CORBA object would then look something like Listing 17-6:

Listing 17-6   Creating the object

IrCall_impl callImpl = new IrCall_impl(some parameters);
IrCall callCORBAObject = callImpl._this(m_sc.getSLEEContext().getORB());
**Hints and Tips**

**Note:** CORBA objects associated with the POA from `m_context.getPOA()` will not have persistent IORs.

**CORBA Object Deactivation**

The ORB maintains an object reference to the CORBA object implementation (the servant) until it is explicitly told to release it. To make an object available for garbage collection by the JVM it must be deactivated as in **Listing 17-7**.

**Listing 17-7  Deactivating a CORBA object**

```java
byte[] id = m_poa.servant_to_id((org.omg.PortableServer.Servant) callImpl);
m_poa.deactivate_object(id);
```

**Note:** User written code must, of course, also release all references to the object in order for it to be garbage collected.