BEA WebLogic Network Gatekeeper ™

Developer’s Guide for Parlay X

Version 2.2
Document Revised: May 15, 2006
Copyright

Copyright © 1995-2006 BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

This software is protected by copyright, and may be protected by patent laws. No copying or other use of this software is permitted unless you have entered into a license agreement with BEA authorizing such use. This document is protected by copyright and may not be copied photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form, in whole or in part, without prior consent, in writing, from BEA Systems, Inc.

Information in this document is subject to change without notice and does not represent a commitment on the part of BEA Systems. THE DOCUMENTATION IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, BEA SYSTEMS DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE DOCUMENT IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

Trademarks and Service Marks


All other names and marks are property of their respective owners.
Contents

1. Introduction and Roadmap
   Document Scope and Audience ................................................. 1-1
   Guide to this Document ......................................................... 1-1
   Terminology ........................................................................... 1-2
   Related Documentation ............................................................ 1-2

2. Introduction and Overall Workflow for Parlay X 1.0
   About WebLogic Network Gatekeeper Web Services applications .......... 2-1
   Architecture ........................................................................ 2-2
      Parlay X based applications ................................................ 2-2
   Development environment ...................................................... 2-3
   Information exchange ............................................................... 2-3
   Overall development workflows .............................................. 2-6
      Client-side Web Services ...................................................... 2-7
      Server-side Web Services ..................................................... 2-8
      Example: Server-side Web Service ....................................... 2-9
   Testing an application ............................................................. 2-10

3. Parlay X 1.0 Web Services API
   About Parlay X Web Services APIs ........................................... 3-2
   WSDL files ............................................................................. 3-3
   About the examples ............................................................... 3-5
   Workflow ................................................................................ 3-5
Login and retrieve login ticket ............................................................... 3-6
Define the security header ................................................................. 3-7
Get a handle to the Web Services port .............................................. 3-7
Add security header ........................................................................... 3-8
Invoke a method ................................................................................ 3-8
Logout ................................................................................................ 3-8

Overview of Supported Capabilities .................................................. 3-9

Access ................................................................................................. 3-9
Third Party Call .................................................................................. 3-9
  Call API ............................................................................................. 3-9
Network Initiated Call ........................................................................ 3-9
  Call API ............................................................................................. 3-9
SMS ...................................................................................................... 3-10
  Send SMS API ................................................................................ 3-10
  SMS Notification API ..................................................................... 3-11
  Receive SMS API .......................................................................... 3-11
Multimedia Message ........................................................................... 3-11
  Send Message API ......................................................................... 3-11
  Receive Message API ..................................................................... 3-11
  Message Notification API ............................................................... 3-12
Payment ............................................................................................... 3-12
  Amount Charging API ................................................................. 3-12
  Volume Charging API .................................................................... 3-12
  Reserved Amount Charging API .................................................... 3-13
  Reserved Volume Charging API ..................................................... 3-13
Terminal Location ............................................................................... 3-13
  Terminal Location API .................................................................... 3-13
User Status ........................................................................................ 3-13
4. Using the Parlay X 2.1 interfaces

Using the Access Web Service ........................................ 4-1
Parlay X 2.1 WSDL files .............................................. 4-2
Parlay X 2.1 interfaces ................................................ 4-2
  Part 1: Common .................................................. 4-2
    Data Types ..................................................... 4-3
    Exceptions ..................................................... 4-3
  Part 4: Short Messaging ............................................ 4-4
    Interface SendSms .............................................. 4-4
    Interface SmsNotification .................................. 4-4
    Interface ReceiveSms ......................................... 4-5
    SmsNotificationManager ..................................... 4-5
  Part 5: Multimedia Messaging .................................... 4-6
    Interface SendMessage ....................................... 4-6
    Interface ReceiveMessage ................................... 4-6
    Interface MessageNotification ................................ 4-7
    Interface MessageNotificationManager ..................... 4-7
  Part 9: Terminal Location ........................................ 4-8
    Interface TerminalLocation .................................. 4-8
    Interface TerminalLocationNotificationManager .......... 4-8
    Interface TerminalLocationNotification ................... 4-9
Support for dual senderName and senderAddress parameters ........ 4-9
5. Parlay X 1.0 Examples

About the examples ................................................................. 5-2
Send SMS ................................................................. 5-2
SMS Notifications ............................................................ 5-3
Send MMS ................................................................. 5-5
Poll for new MMSes .......................................................... 5-7
Receive notifications about new MMSes ............................... 5-9
Get an MMS by it’s message reference ID .............................. 5-10
Handling SOAP Attachments .............................................. 5-11
  Encoding a multipart SOAP attachment ............................. 5-11
  Retrieving and Decoding a multipart SOAP attachment ........ 5-13
Setting up a two-party call from an application ..................... 5-15
Handling network-initiated calls ......................................... 5-18
Get location ................................................................. 5-22
Get user status .............................................................. 5-24
Reserve and charge an account ......................................... 5-26

A. References
Introduction and Roadmap

The following sections describe the audience for and organization of this document:

- “Document Scope and Audience” on page 1-1
- “Guide to this Document” on page 1-1
- “Terminology” on page 1-2
- “Related Documentation” on page 1-2

Document Scope and Audience

This guide describes how to develop telecom-enabled applications based on the Parlay X 1.0 and 2.1 APIs and how to access and use the APIs/interfaces as exposed by WebLogic Network Gatekeeper.

This guide contains code fragments from example applications written in Java to illustrate different aspects of the usage of the interfaces.

The purpose of this guide is not to describe Web Service development in general, but rather how to use the specific interfaces.

All example code is Axis-specific.

Guide to this Document

- Chapter 1, “Introduction and Roadmap,” The structure and contents of this document, the writing conventions used, and related documentation.
Introduction and Roadmap

- Chapter 2, “Introduction and Overall Workflow for Parlay X 1.0,” An introduction to the two main types of WebLogic Network Gatekeeper Web services applications. The programming environment and development workflows.
- Chapter 3, “Parlay X 1.0 Web Services API,” Adding telecom functions to your Web Services Application using Parlay X 1.0.
- Chapter 4, “Using the Parlay X 2.1 interfaces,” An overview of the interfaces and operations supported in the Palray X 2.1 interface.
- Chapter 5, “Parlay X 1.0 Examples,” Code examples using Parlay X 1.0.

Terminology

The following terms and acronyms are used in the document:

- API — Application Programming Interface
- CORBA — Common Object Request Broker Architecture
- HTML — Hypertext Markup Language
- MMS — Multimedia Message Service
- RPC — Remote Procedure Call
- ORB — Object Request Broker
- SMS — Short Message Service
- SwA — SOAP with Attachments
- WSDL — Web Services Definition Language
- WSI-I — Web Services Interoperability
- SPA — Service Provider API
- XML — Extended Markup Language

Related Documentation

This Developer’s Guide is a part of WebLogic Network Gatekeeper documentation set. The following documents contain other types information:

- *API Description Parlay X 1.0 for WebLogic Network Gatekeeper*
The API description describes the Parlay X 1.0 API and its implementation in WebLogic Network Gatekeeper.

- **Parlay X 1.0 Specification, http://www.parlay.org**
  The Parlay X 1.0 specification describes the Parlay X 1.0 APIs available for programmers and applications.

  The Parlay X 2.1 specification describes the Parlay X 2.1 APIs available for programmers and applications. The 3GPP edition of the specification is used.
Introduction and Overall Workflow for Parlay X 1.0

The following sections provide an overview of developing applications using the Parlay X 1.0 APIs:

- “About WebLogic Network Gatekeeper Web Services applications” on page 2-1
- “Architecture” on page 2-2
- “Development environment” on page 2-3
- “Information exchange” on page 2-3
- “Overall development workflows” on page 2-6
- “Testing an application” on page 2-10

About WebLogic Network Gatekeeper Web Services applications

The WebLogic Network Gatekeeper Web Services interfaces make it possible for TCP/IP based applications to offer their users access to telecom functionality. The Network Gatekeeper offers three separate sets of Web Services APIs: Parlay X 1/0; Parlay X 2.1 (partial); and Extended Web Services. The choice of which API to use depends on the application’s needs for network functionality and means of access.

For applications that wish to provide only basic telecom functionality, the standardized Parlay X APIs may be a good choice. For applications with a need for more granular control, the Extended Web Services interfaces may be more appropriate.
This section describes how to develop Parlay X 1.0 applications that connect to an underlying telecom network using WebLogic Network Gatekeeper. Using the Parlay X APIs you can quickly develop powerful telecom-enabled applications using any programming environment that supports Web Services.

**Architecture**

Figure 2-1, “Parlay X Web Services interfaces,” on page 2-2 illustrates different ways of using the Web Services APIs as provided by WebLogic Network Gatekeeper, as well as examples of different execution environments for applications.

Applications using the Parlay X APIs can execute in any environment capable of handling Web Services, as the applications communicate with WebLogic Network Gatekeeper using SOAP/HTTP.
Parlay X based applications

A Web Services Parlay X application:

- Uses an API that is standardized
- Has a well-defined, small set of available methods
- Is stateless.

Development environment

This is a simple Web Services development environment. Integrated programming environments, like Visual Studio .Net can be used for development of Web Services applications, but this guide uses a minimalistic approach. For the purpose of this guide, the following will do:

- an ordinary text editor.
- Axis 1.1, see “Apache Axis, http://ws.apache.org/axis”.
  - axis.jar
  - axis-ant.jar
  - commons-discovery.jar
  - commons-logging.jar
  - jaxrpc.jar
  - saaj.jar
  - wsdl4j.jar
- JavaMail API 1.2, see “JavaMail, http://java.sun.com”, for messaging applications handling multimedia messages.
  - mail.jar
  - activation.jar

Information exchange

Before an application is developed, the application developer/service provider and the Network Gatekeeper operator must exchange information regarding resources.
Decide what functionality is needed, and the Network Gatekeeper modules that support that functionality - Messaging, Call Control, Location, etc. - and map it to the appropriate Parlay X API. More information on the functionality supported by the Network Gatekeeper is available in *Architectural Overview - Web Logic Network Gatekeeper*, a separate document in this set.

Based on the functionality you choose, exchange information with the Network Gatekeeper operator based on Table 2-1. The WebLogic Network Gatekeeper operator must also communicate which services and methods are supported by the deployment.

Information related to commercial and security considerations and privacy regulations must also be exchanged. For example:

- Charging plans
- Number of concurrent application instances you expect to be running.
- Projected usage amounts: for example, the number of `sendSMS` requests you expect to process per time period.
- Black/white lists of addresses.
- Allow/deny lists for User Status and User Location requests.
Table 2-1  Information exchange between Parlay X 1.0 application developer and WebLogic Network Gatekeeper operator

<table>
<thead>
<tr>
<th>Module</th>
<th>API</th>
<th>Information to be provided by the</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Application developer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WebLogic Network Gatekeeper Operator</td>
</tr>
<tr>
<td>Access</td>
<td>Access</td>
<td>Application ID.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Service provider ID.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Application Instance Group ID.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Password for the Application Instance Group.</td>
</tr>
<tr>
<td>Network-Initiated Third Party Call</td>
<td>Call</td>
<td>URL of the end-point. If the application is to be triggered when the calling party goes off-hook.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Access number to the application, if any. Can be a range of numbers.</td>
</tr>
<tr>
<td>SMS</td>
<td>Send SMS</td>
<td>Mailbox ID and corresponding password.</td>
</tr>
<tr>
<td>SMS Notification</td>
<td>URL of the end-point.</td>
<td>Access number to the application. Mailbox ID and corresponding password.</td>
</tr>
</tbody>
</table>
### Overall development workflows

There are two main types of workflow for the development of Web Services applications based on Parlay X APIs:

- The WebLogic Network Gatekeeper acts as a server and the application is the client. In this scenario, the application uses a Web Service provided by WebLogic Network Gatekeeper. For example, the application invokes the `sendSMS` operation on the Network Gatekeeper. This is the more common mode.

- The application acts as a server and WebLogic Network Gatekeeper is the client. In this scenario, the application itself is the provider of a Web Service and WebLogic Network Gatekeeper invokes methods on this Web Service.

Often, an application acts as both server and client.

---

### Table 2-1 Information exchange between Parlay X 1.0 application developer and WebLogic Network Gatekeeper operator

<table>
<thead>
<tr>
<th>Module</th>
<th>API</th>
<th>Information to be provided by the Application developer</th>
<th>WebLogic Network Gatekeeper Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive SMS</td>
<td></td>
<td>Mailbox ID and corresponding password.</td>
<td></td>
</tr>
<tr>
<td>Multimedia</td>
<td>Send Message</td>
<td>Mailbox ID and corresponding password.</td>
<td></td>
</tr>
<tr>
<td>Message</td>
<td>Receive Message</td>
<td>Mailbox ID and corresponding password.</td>
<td></td>
</tr>
<tr>
<td>Notification</td>
<td>URL of the end-point. Access number to the application. Mailbox ID and corresponding password.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The methods that the application invokes on Network Gatekeeper are defined in WSDL files, one for each service capability, where the name of the file reflects the capability. There are also WSDLs that define the methods that the application must implement to receive various notifications from WebLogic Network Gatekeeper. In Parlay X, all methods starting with “handle” or “notify,” such as handleBusy, are methods that WebLogic Network Gatekeeper invokes on the application’s server-part. The application must implement these methods. All WSDLs for Parlay X 1.0 use an RPC/encoded binding, while the WSDLs for Parlay X 2.1 use document/literal.

The method invocations are SOAP requests over HTTP, which means that if the application wishes to receive notifications, the server part of the application must be capable of handling SOAP requests. For example, while the Simple Axis Server could be used as a SOAP engine during the testing phase, in a production system, Axis in combination with Tomcat would be required.

Client-side Web Services

Below is a high level description of the work sequence for developing clients for telecom-enabled Web Services:

1. Retrieve the necessary IDs for the resources the application will use from the service provider. Examples are mailbox IDs, short numbers for network triggered applications and so on.
2. Retrieve the WSDL file that handles user login from the operator provided endpoint.
3. Retrieve WSDL files for the services the application is to use from the operator provided endpoint.

4. Generate stubs/proxy classes for the language in which you are implementing the application. The simplest way to do this is to use a tool that converts the WSDL into a proxy/stub for the preferred language. Examples of such tools include WSDL2Java and Soap Toolkit.

5. Compile and create jar-files from the Java stubs.

6. Use the generated APIs to add telecom functionality to the application.

7. Compile the application.

8. Test the application in a test environment like the WebLogic Network Gatekeeper Application Test Environment.

9. Connect the application to WebLogic Network Gatekeeper with a connection to a live telecom network.

Server-side Web Services

Below is a high level description of the work sequence for developing server-side Web Services for interacting with Network Gatekeeper:

1. Retrieve the necessary IDs for the resources the application will use from the Network Gatekeeper operator. These include mailbox IDs, short numbers for network-triggered applications and so on.

2. Retrieve the WSDL files for the listeners you want to use from the operator provided endpoint.

3. Generate skeleton classes for the language in which you are implementing the application. The simplest way to do this is to use a tool that converts the WSDL into a stub for the preferred language. Examples of such tools are WSDL2Java and Soap Toolkit.

4. Compile and create jar-files from the Java stubs.

5. Implement the generated interfaces, adding the ability to receive notifications and other requests from Network Gatekeeper to the application.

6. Adapt the generated WSDD file to bind the SOAP requests to the appropriate class.

7. Compile the application.
8. Deploy the application in an environment capable of decoding HTTP/SOAP messages, such as Axis.

9. Communicate the end-point of the new application to the Network Gatekeeper operator, who will register them using OAM.

10. Test the application in a test environment like the WebLogic Network Gatekeeper Application Test Environment.

11. Connect the application to WebLogic Network Gatekeeper with a connection to a live telecom network.

**Example: Server-side Web Service**

The example below shows how to create a Web Service to receive notification from Network Gatekeeper that a new SMS for the application has arrived from the network. The Web Service implements the the SMS notification API, containing the method `notifySmsReception()`.

For this example, the Simple Axis Server is used as deployment environment for the application.

1. Retrieve the WSDL files from the endpoint specified by the Network Gatekeeper operator.

2. Generate Java skeletons from the WSDL files (here using WSDL2Java):

   ```
   %java org.apache.axis.wsdl.WSDL2Java --server-side --skeletonDeploy true parlayx_sms_notification_service.wsdl
   ```

   **Note:** The Axis files must be in the classpath.

3. Compile and create jar-files from the skeletons.

4. Move the empty implementation of the generated interfaces to the source directory of the application.

   **Note:** In the example, the class is named `SmsNotificationBindingImpl`. When generating skeletons using WSDL2Java, the empty interface implementations are named `<Name of API>BindingImpl`. Other tools may have different naming conventions.

5. Adapt the generated Web Service Deployment Descriptor (WSDD) files to bind the SOAP request to the appropriate class.

   The WSDD files are used when deploying and undeploying services. Two files are generated: `deploy.wsdd` and `undeploy.wsdd`.

   In the example, the tag
<parameter name="className" value="org.csapi.www.wsdl.parlayx.sms.v1_0.notification.SmsNotification BindingSkeleton"/>

is replaced with

<parameter name="className" value="com.acme.apps.getSmsApp.SmsNotification"/>

in order to bind to the appropriate class.

6. Compile the application.

7. Verify that the application is deployed correctly by using a Web browser and pointing it to the URL of the web service. In the case of Simple Axis Server, the deployed Web Services can be found at the URL http://<host>::<port>/axis/services.

8. Provide the Network Gatekeeper operator with the URL to the Web Service, along with the data that identifies the application to Network Gatekeeper: the application ID and application instance group ID.

9. Run the application. The adapted file deploy.wsdd is used when instantiating the Simple Axis Server.

Testing an application

Figure 2-3, “Application test flow,” on page 2-11 shows the application test flow, from the application developer’s functional test to deployment in a live network. An application developer can perform functional tests using WebLogic Network Gatekeeper Application Test Environment. The other tests in the flow are performed in cooperation between the application provider and the Network Gatekeeper operator.
First the application is connected to the WebLogic Network Gatekeeper Application Test Environment (ATE), which emulates WebLogic Network Gatekeeper, using endpoints that belong to the ATE. It can then be attached to a Network Gatekeeper which is connected to a network simulator, using the endpoints in the Network Gatekeeper. Finally it connects to a Network Gatekeeper that is attached to a test network before it is put into a production system.

An overview of the relation between WebLogic Network Gatekeeper Application Test Environment and WebLogic Network Gatekeeper is shown in Figure 2-4, “ATE in relation to WebLogic Network Gatekeeper,” on page 2-12.
For applications based on Web Services, the application uses the endpoints provided by ATE during test. After successful verification, the application uses endpoints provided by WebLogic Network Gatekeeper.
Parlay X 1.0 Web Services API

The following sections describe the Parlay X Web Services API:

- “About Parlay X Web Services APIs” on page 3-2
- “WSDL files” on page 3-3
- “About the examples” on page 3-5
- “Workflow” on page 3-5
- “Overview of Supported Capabilities” on page 3-9
  - “Access” on page 3-9
  - “Third Party Call” on page 3-9
  - “Network Initiated Call” on page 3-9
  - “SMS” on page 3-10
  - “Multimedia Message” on page 3-11
  - “Payment” on page 3-12
  - “Terminal Location” on page 3-13
  - “User Status” on page 3-13
- “Addresses” on page 3-14
- “Data types and enumerations” on page 3-14
About Parlay X Web Services APIs

The Parlay X Web Services API offers a high-level abstraction of telecom network functionality for use in any application that can function in a Web Services environment.

The API is designed for rapid application development. From an architectural point of view, the implementation of the API resides on top of the service capability modules in WebLogic Network Gatekeeper. For more information on the structure of WebLogic Network Gatekeeper, see *Architectural Overview - WebLogic Network Gatekeeper*, a separate document in this set.

All applications accessing WebLogic Network Gatekeeper through the Web Services interfaces use a Kerberos type of service token-based authentication. The application is provided with a user name (the application instance group ID) and a password. When an application wants access to WebLogic Network Gatekeeper the application instance logs in using the user name and password together with the application account ID and service provider ID to retrieve a service token. These IDs are established by the Network Gatekeeper operator, and provided either to the application developer directly, or through the service provider group to which the application belongs. This mechanism may be extended, using, for example Passport or other extended Kerberos Key Distribution Centre (KDC) authentication solutions, according to the WSSE (Web Services Security) standard.

To develop an application using the Parlay X 1.0 APIs, the developer needs:

- Access to either an instance of WebLogic Network Gatekeeper or WebLogic Network Gatekeeper Application Test Environment
- The WSDL files that define the interfaces for the desired services
- Login credentials and IDs of resources to use, which must be provided by the Network Gatekeeper operator.
The interfaces are separated into different modules. Each main type of telecom service is contained in a specific module. The modules are:

<table>
<thead>
<tr>
<th>Module</th>
<th>Defines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Third party call</td>
<td>Methods for handling application initiated calls.</td>
</tr>
<tr>
<td>Network-initiated third party call</td>
<td>Methods for handling network initiated calls.</td>
</tr>
<tr>
<td>SMS</td>
<td>Methods for handling sending and reception of SMSes.</td>
</tr>
<tr>
<td>Multimedia Message</td>
<td>Methods for handling sending and reception of MMSes.</td>
</tr>
<tr>
<td>Payment</td>
<td>Methods for handling charging based on content.</td>
</tr>
<tr>
<td>Terminal location</td>
<td>Methods for retrieving the geographical position of a mobile terminal.</td>
</tr>
<tr>
<td>User status</td>
<td>Methods for retrieving information on the status of mobile terminals.</td>
</tr>
</tbody>
</table>

In addition there is a Network Gatekeeper specific API, Access, that the application must use to log into Network Gatekeeper.

All Parlay X APIs except Account Management are supported by WebLogic Network Gatekeeper. For detailed information on individual methods and WebLogic Network Gatekeeper specifics not covered by the standard, see API Description Parlay X 1.0 for WebLogic Network Gatekeeper. For information on the standard itself, see Parlay X 1.0 Specification, http://www.parlay.org.

WSDL files

The interfaces for the Network Gatekeeper implementation of Parlay X 1.0 interfaces are published in WSDL files using the RPC/encoded binding:

- By default the service endpoints and the WSDLs for setting up client side implementations can be found at http://<URL to WebLogic Network Gatekeeper>/parlayx/servlet/AxisServlet
Also by default, the WSDL files for the northbound (Listener) interfaces can be fetched from
http://<URL to WebLogic Network Gatekeeper>/parlayx/wsd1

<table>
<thead>
<tr>
<th>Module</th>
<th>WSDL-file to download</th>
<th>API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>Access</td>
<td>Access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This API is specific for WebLogic Network Gatekeeper, and not part of the Parlay X specification.</td>
</tr>
<tr>
<td>Third Party Call</td>
<td>parlayx_third_party_calling_service</td>
<td>Call</td>
</tr>
<tr>
<td>Network Initiated Third Party Call</td>
<td>parlayx_network_initiated_call</td>
<td>Call</td>
</tr>
<tr>
<td>SMS</td>
<td>parlayx_sms_service</td>
<td>Send SMS</td>
</tr>
<tr>
<td></td>
<td>parlayx_sms_notification_service</td>
<td>SMS Notification</td>
</tr>
<tr>
<td>Multimedia Message</td>
<td>parlayx_mm_service</td>
<td>Send Message</td>
</tr>
<tr>
<td></td>
<td>parlayx_mm_service</td>
<td>Receive Message</td>
</tr>
<tr>
<td></td>
<td>parlayx_mm_notification_service</td>
<td>Message Notification</td>
</tr>
<tr>
<td>Payment</td>
<td>parlayx_payment_service</td>
<td>Amount Charging</td>
</tr>
<tr>
<td></td>
<td>parlayx_payment_service</td>
<td>Volume Charging</td>
</tr>
<tr>
<td></td>
<td>parlayx_payment_service</td>
<td>Reserved Amount Charging</td>
</tr>
<tr>
<td></td>
<td>parlayx_payment_service</td>
<td>Reserved Volume Charging</td>
</tr>
<tr>
<td>Terminal Location</td>
<td>parlayx_terminal_location_service</td>
<td>Terminal Location</td>
</tr>
<tr>
<td>User Status</td>
<td>parlayx_user_status_service</td>
<td>User Status</td>
</tr>
</tbody>
</table>
For a description of the methods in each API, see API Description Parlay X 1.0 for WebLogic Network Gatekeeper.

**About the examples**

The examples in this chapter use Java and Axis. The invocation techniques used is JAX-RPC using Dynamic Invocation Interface. The WSDL files describing the services are used to generate stubs and skeletons in Java.

**Workflow**

The main program control flow when executing applications based on Parlay X Web Services is described in pseudo code in Figure 3-1, “Application execution workflow,” on page 3-5.

![Figure 3-1 Application execution workflow](image-url)
a. See “Login and retrieve login ticket” on page 3-6.
b. See “Define the security header” on page 3-7.
c. See “Get a handle to the Web Services port” on page 3-7.
d. See “Add security header” on page 3-8.
e. See “Invoke a method” on page 3-8.
f. See “Logout” on page 3-8

Login and retrieve login ticket

Before your application can make a request to any telecom service using WebLogic Network
Gatekeeper, it must acquire a login ticket. This login ticket identifies the session and is included
in the SOAP header of every subsequent request during the session. The login ticket is valid either
until a logout is performed or until an operator-established time period has elapsed. If the latter
occurs, the application may be able to request a refreshed ticket, or it may need to re-login,
depending on the way the operator has configured the Network Gatekeeper.

Details about locators, endpoints, so on are explained later in this section.

Listing 3-1 Login

```java
AccessService accessService = new AccessServiceLocator();
java.net.URL endpoint = new java.net.URL(wsdlUrl);
Access access = accessService.getAccess(endpoint);
String sessionId = access.applicationLogin(spID,
          appID,
          appInstGroupID,
          appInstGroupPassword);
```

The login ticket ID retrieved when invoking applicationLogin is used in each consecutive
invocation towards WebLogic Network Gatekeeper. See “Define the security header” on
page 3-7.
The login credentials spID, appID, appInstGroupId, and appInstGoupPassword are generated by the Network Gatekeeper operator and are provided either directly to the application developer by the operator or via of the service provider group to which the application belongs.

Define the security header

Once the login ticket is acquired, it is sent in the SOAP header together with a username/password combination each time a Web Service method is invoked.

Listing 3-2  Define the security header

```java
org.apache.axis.message.SOAPHeaderElement header =
new org.apache.axis.message.SOAPHeaderElement(wsdlUrl, "Security", "");
header.setActor("wsse:PasswordToken");
header.addAttribute(wsdlUrl, "Username", ""+userName);
header.addAttribute(wsdlUrl, "Password", ""+sessionId);
header.setMustUnderstand(true);
```

Note that the login ticket is supplied in the Password attribute. The userName attribute is defined by the operator, normally in the format `<myUserName>@<myapplication>`. Axis 1.1 does not contain WSSE helper classes, so this must be performed manually.

The header is added to the Web Services request. Also see “Add security header” on page 3-8.

Get a handle to the Web Services port

Next you must get a handle for invoking a method, using, in this example, the SendSms interface.

Listing 3-3  Get hold of a port

```java
SendSmsService sendSmsService = new SendSmsServiceLocator();
java.net.URL endpoint = new java.net.URL(sendSmsWsd1Url);
SendSmsPort sendSms = sendSmsService.getSendSmsPort(endpoint);
```
The details on the parameters of the send SMS API are described in API Description Parlay X 1.0 for WebLogic Network Gatekeeper.

Add security header

Adding the security header to a request is straightforward, as illustrated below. For information on creating the header, see “Define the security header” on page 3-7.

Listing 3-4  Add security header

```java
((org.apache.axis.client.Stub)sendSms).setHeader(header);
```

Invoke a method

Using the handle you acquired above, invoke a method - in this case sendSms.

Listing 3-5  Invoke a method

```java
String sendID = sendSms.sendSms(eui, myMailbox, "CP_FREE", myMessage );
```

The details on the parameters of the send SMS API are described in API Description Parlay X 1.0 for WebLogic Network Gatekeeper.

Logout

Close the session.

Listing 3-6  Logout

```java
access.applicationLogout(sessionId);
```
The login Ticket is destroyed and cannot be used in consecutive method invocations.

Overview of Supported Capabilities

Access
This API is specific to WebLogic Network Gatekeeper, and is not part of the Parlay X specification. It provides the following functionality:

- Login an application to WebLogic Network Gatekeeper.
- Logout an application from WebLogic Network Gatekeeper.
- Change password.

For a description on using this API see “Login and retrieve login ticket” on page 3-6 and “Logout” on page 3-8. There is also support for changing the password.

Third Party Call

Call API
This API contains Web Services methods for handling application initiated, two-party telephony calls. The following functionality is provided:

- Connect two parties in a call.
- End the call.
- Get information about an ongoing call.
- Cancel call setup.

Network Initiated Call

Call API
This API contains Web Services methods for handling network initiated calls. It is a listener interface, which is implemented on the server-side of the application. The following functionality is provided:

- Check to see if the terminal of a called party is busy (off-hook before the call attempt).
• Check to see if the terminal of a called party is not reachable (for example, if it is turned off.

• Check to see if a called party does not go off-hook (does not answer) within a certain time-interval which has been defined by the telecom network.

• Check to see if a called party goes off-hook (answers)

Several possible actions can be taken when any of the above listed information reaches the application. It can:

• Continue handling the call as it would normally be performed in the network.

• Re-route the call to a destination specified by the application.

• End the call.

In each of the above scenarios, Network Gatekeeper invokes a Web Service method on an endpoint on the server-side of the application in order to learn what action the application wishes Network Gatekeeper to perform. The endpoint of the server-side application must be communicated to the operator by the application provider, as the information must be entered into Network Gatekeeper manually, using OAM methods.

For more information on implementing Web Services, see “Server-side Web Services” on page 2-8 and “Example: Server-side Web Service” on page 2-9.

**SMS**

**Send SMS API**

This API contains Web Services methods for sending SMSes.

The following functionality is provided:

• Send SMS.

• Send SMS logo.

• Send SMS ringtone.

• Get the delivery status of an SMS.
Overview of Supported Capabilities

SMS Notification API
This API contains Web Services methods for being notified on the arrival of new incoming SMSes. It is a listener interface, which is implemented on the server-side of the application. The following functionality is provided:

- Receive notification that a new SMS for the application has arrived at WebLogic Network Gatekeeper.

In this case, Network Gatekeeper invokes a Web Service method on an endpoint on the server-side of the application in order to inform the application that the SMS has arrived. The endpoint of the server-side application must be communicated to the operator by the application provider, as the information must be entered into Network Gatekeeper manually, using OAM methods.

For more information on implementing Web Services, see “Server-side Web Services” on page 2-8 and “Example: Server-side Web Service” on page 2-9.

Receive SMS API
This API contains Web Services methods for fetching SMSes.

The following functionality is provided:

- Get SMSes for the application that have arrived at WebLogic Network Gatekeeper.

Multimedia Message

Send Message API
This API contains Web Services methods for sending multimedia messages. MMS and e-mail are supported. The following functionality is provided:

- Send message.
- Get delivery status of a sent message.

Receive Message API
This API contains Web Services methods for fetching multimedia messages. The following functionality is provided:

- Poll for new messages.
- Fetch individual messages.

**Message Notification API**
This API contains Web Services methods for receiving notifications on new incoming multimedia messages. It is a listener interface, which is implemented on the server-side of the application. The following functionality is provided:
- Receive notification when a new multimedia message for the application has arrived at Network Gatekeeper.

In this case, Network Gatekeeper invokes a Web Service method on an endpoint on the server-side of the application in order to inform the application that the MMS has arrived. The endpoint of the server-side application must be communicated to the operator by the application provider, as the information must be entered into Network Gatekeeper manually, using OAM methods.

For more information on implementing Web Services, see “Server-side Web Services” on page 2-8 and “Example: Server-side Web Service” on page 2-9.

**Payment**

**Amount Charging API**
This API contains Web Services methods for handling charging in amount units based on content. The following functionality is provided:
- Charge an amount from a user’s account.
- Refund an amount to a user’s account.

**Volume Charging API**
This API contains Web Services methods for handling charging in volume units based on content. The following functionality is provided:
- Charge a volume from a user’s account.
- Refund a volume to a user’s account.
- Convert a volume to an amount.
Reserved Amount Charging API
This API contains Web Services methods for reservation of amount units based on content. The following functionality is provided:

- Reserve an amount from a user’s account.
- Reserve and unreserved an additional amount.
- Charge a reservation.
- Refund funds left in a reservation and release the reservation.

Reserved Volume Charging API
This API contains Web Services methods for reservation of volume units based on content. The following functionality is provided:

- Get the amount that corresponds to a given volume.
- Reserve a volume from a user’s account.
- Reserve an additional volume.
- Charge a reservation.
- Refund funds left in a reservation and release the reservation.

Terminal Location

Terminal Location API
This API contains Web Services methods to get the geographical position of a mobile terminal. The following functionality is provided:

- Get the location of a mobile terminal.

User Status

User Status API
This API contains Web Services methods to get the status of a terminal, for example busy and off hook. The following functionality is provided:
Get the status of a user. Status can be one of the following: online, offline, busy, and other.

**Addresses**

Addresses are specified as `EndUserIdentifiers`. This is a datatype defined by Parlay X.

The `EndUserIdentifier` is defined as an Uniform Resource Identifier specified as a URI: `[scheme]://[schemeSpecificPart]` (RFC 2396, amended by RFC 2732).

Where `scheme` is one of the following: `[tel | sip | mailto]` and `schemeSpecificPart` is the actual address.

**Examples**

If the address is a telephone number, the `EndUserIdentifier` is as follows: “tel:+461234567”

If the address is an e-mail address, the `EndUserIdentifier` is as follows: “mailto:someone@somecompany.com”

If the address is a sip telephone number, the `EndUserIdentifier` is as follows: “sip:someone@somecompany.com”

**Data types and enumerations**

Some datatypes, like, for example, `EndUserIdentifier`, are defined by the Parlay X standard. These datatypes are defined using WSDL. Some other datatypes are enumerations of values. Different programming languages use different approaches to handling variables or classes of these types.

Using Java, an `EndUserIdentifier` which holds a telephone number would look like this:

```java
EndUserIdentifier eu = new EndUserIdentifier();
eu.setValue(new URI("tel", "4654176700"));
```

For a information on all datatypes, see API Description Parlay X 1.0 for WebLogic Network Gatekeeper.
CHAPTER 4

Using the Parlay X 2.1 interfaces

The Parlay X 2.1 interfaces provided by Network Gatekeeper follows the same architecture and structure as the Parlay X 1.0 interfaces. This section provides a description of the Parlay X 2.1 interfaces and how they are supported by Network Gatekeeper.

- “Using the Access Web Service” on page 4-1
- “Parlay X 2.1 WSDL files” on page 4-2
- “Parlay X 2.1 interfaces” on page 4-2

The Parlay X 2.1 interfaces exposed are compliant with the 3GPP Specification (TS 29.199 2005-12).

In section “Parlay X 2.1 interfaces” on page 4-2, the endpoints of the web services and which operations supported are stated. For a full description of the interfaces, refer to the specification.

Using the Access Web Service

All applications using Parlay X 2.1 must login to WebLogic Network Gatekeeper using the non-Parlay X Access interface in the same manner as applications based on Parlay X 1.0.

This interface returns a login ticket, which must be included in the SOAP header of all subsequent requests. More information on using this interface and on login tickets in general can be found in the section “Workflow” on page 3-5.

The service endpoint for Access used with Parlay X 2.1 is:

http://<IP-address>:<port>/parlayx2/services/Access
Using the Parlay X 2.1 interfaces

Parlay X 2.1 WSDL files

To acquire the WSDL file for each service, add ?wsdl to the service endpoint URL. For example:

http://<IP-address>:<port>/parlayx2/services/SendSms?wsdl

The notification WSDLs can be found at:
http://<IP-address>:<port>/parlayx2/wsdl/

Where <IP-address> and <port> are the locations at which the Network Gatekeeper is configured to expose the Web Services interfaces.

The WSDL binding style is WS-I basic profile, doc/lit compliant.

Parlay X 2.1 interfaces

Network Gatekeeper exposes the following Parlay X 2.1 interfaces:

- Common according to 3GPP TS 29.199-1 V6.2.0 (2005-06)
- Short Messaging according to 3GPP TS 29.199-4 V6.3.0 (2005-06)
- Multimedia Messaging according to 3GPP TS 29.199-5 V6.3.0 (2005-06)
- Terminal Location according to 3GPP TS 29.199-9 V6.2.0 (2005-06)

See the sections below for detailed information about methods and parameters.

Part 1: Common

The following datatypes an exceptions are supported.

Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Compliant</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimeMetric</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>ChargingInformation</td>
<td>Yes</td>
<td>See note below.</td>
</tr>
<tr>
<td>SimpleReference</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>
Exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Compliant</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ServiceException</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>PolicyException</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

Note: In some situations, applications sending SMS or MMS messages need to include charging data as an inline message part. According to specification, this data is to be presented in a ChargingInformation structure.

<table>
<thead>
<tr>
<th>Element name</th>
<th>Element type</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>description</td>
<td>xsd:string</td>
<td>No</td>
<td>Description text to be used for information and billing text</td>
</tr>
<tr>
<td>currency</td>
<td>xsd:string</td>
<td>Yes</td>
<td>Currency identifier as defined in ISO 4217[12]</td>
</tr>
<tr>
<td>amount</td>
<td>xsd:string</td>
<td>Yes</td>
<td>Amount to be charged</td>
</tr>
<tr>
<td>code</td>
<td>xsd:string</td>
<td>Yes</td>
<td>Charging code, referencing a contract under which the charge is applied</td>
</tr>
</tbody>
</table>

The Parlay X 2.1 implementation supports this structure, but has a total string length (description + currency + amount + code) limit of 85 characters. If the data length exceeds this limit, a ServiceException, id SVC1005, is thrown.

Part 4: Short Messaging

Interface SendSms

Service Endpoint found at: http://<IP-address>:<port>/parlayx2/services/SendSms
Using the Parlay X 2.1 interfaces

### Interface SmsNotification

Service Endpoint is given by the application

<table>
<thead>
<tr>
<th>Operation</th>
<th>Compliant</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendSms</td>
<td>Yes</td>
<td>See “Support for dual senderName and senderAddress parameters” on page 4-9.</td>
</tr>
<tr>
<td>sendSmsLogo</td>
<td>Yes</td>
<td>Logos must be in either SmartMessaging or EMS format.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The image is not scaled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The SmsFormat parameter is required.</td>
</tr>
<tr>
<td>sendSmsRingtone</td>
<td>Yes</td>
<td>See “Support for dual senderName and senderAddress parameters” on page 4-9.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ringtones must be in either SmartMessaging or EMS (iMelody) format.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The SmsFormat parameter is required</td>
</tr>
<tr>
<td>getSmsDeliveryStatus</td>
<td>Yes</td>
<td>See “Support for dual senderName and senderAddress parameters” on page 4-9.</td>
</tr>
</tbody>
</table>

### Interface ReceiveSms

Service Endpoint found at:

http://<IP-address>:\<port>/parlayx2/services/ReceiveSms
### SmsNotificationManager

Service Endpoint found at:
http://<IP-address>:<port>/parlayx2/services/SmsNotificationManager

<table>
<thead>
<tr>
<th>Operation</th>
<th>Compliant</th>
<th>Comments</th>
</tr>
</thead>
</table>
| getReceivedSms             | Yes       | The format of the parameter registrationIdentifier is `<mailbox ID>` `<mailbox password>`  
Mailbox ID and password are supplied by the service provider.  
Example:  
"tel:50000\apassword"  |
| startSmsNotification       | Yes       | The format of the parameter SmsServiceActivationNumber is tel:`<mailbox ID>` ;mboxPwd=`<mailbox password>`  
Mailbox ID and password are supplied by the service provider.  
Example:  
"tel:50000;mboxPwd=apassword"  
Also see  
http://www.ietf.org/rfc/rfc3966.txt  |
| stopSmsNotification        | Yes       |                                                                                                                                                                                                          |
Part 5: Multimedia Messaging

Interface SendMessage

Service Endpoint found at:
http://<IP-address>:<port>/parlayx2/services/SendMessage

<table>
<thead>
<tr>
<th>Operation</th>
<th>Compliant</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>sendMessage</td>
<td>Yes</td>
<td>Messages sent as Attachments. Only MMS applicable; Email not supported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See “Support for dual senderName and senderAddress parameters” on page 4-9.</td>
</tr>
</tbody>
</table>

Interface ReceiveMessage

Service Endpoint found at:
http://<IP-address>:<port>/parlayx2/services/ReceiveMessage

<table>
<thead>
<tr>
<th>Operation</th>
<th>Compliant</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>getMessageURIs</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>getMessage</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>
**Interface MessageNotification**
Service Endpoint is provided by the application

<table>
<thead>
<tr>
<th>Operation</th>
<th>Compliant</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>notifyMessageReception</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

**Interface MessageNotificationManager**
Service Endpoint found at:
http://<IP-address>:<port>/parlayx2/services/MessageNotificationManager

<table>
<thead>
<tr>
<th>Operation</th>
<th>Compliant</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>startMessageNotification</td>
<td>Yes</td>
<td>The format of the parameter MessageServiceActivationNumber is tel:&lt;mailbox ID&gt;;mboxPwd=&lt;mailbox password&gt; Mailbox ID and password are supplied by the service provider. Example: &quot;tel:50000;mbxPwd=apassword&quot; Also see <a href="http://www.ietf.org/rfc/rfc3966.txt">http://www.ietf.org/rfc/rfc3966.txt</a></td>
</tr>
<tr>
<td>stopMessageNotification</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

**Part 9: Terminal Location**

**Interface TerminalLocation**
Service Endpoint found at:
http://<IP-address>:<port>/parlayx2/services/TerminalLocation
Using the Parlay X 2.1 interfaces

Interface TerminalLocationNotificationManager
Service Endpoint found at:
http://<IP-address>:<port>/parlayx2/services/TerminalLocationNotificationManager

<table>
<thead>
<tr>
<th>Operation</th>
<th>Compliant</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>getLocation</td>
<td>Yes</td>
<td>Charging based on Requestor not supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AcceptableAccuracy not supported</td>
</tr>
<tr>
<td>getTerminalDistance</td>
<td>Yes</td>
<td>Charging based on Requestor not supported</td>
</tr>
<tr>
<td>getLocationForGroup</td>
<td>Yes</td>
<td>Charging based on Requestor not supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AcceptableAccuracy not supported</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Compliant</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>startGeographicalNotification</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>startPeriodicNotification</td>
<td>Yes</td>
<td>Duration not supported</td>
</tr>
<tr>
<td>endNotification</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

Interface TerminalLocationNotification
Service Endpoint provided by the application

<table>
<thead>
<tr>
<th>Operation</th>
<th>Compliant</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>locationNotification</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>locationError</td>
<td>Yes</td>
<td>Address not supported</td>
</tr>
<tr>
<td>locationEnd</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>
Support for dual senderName and senderAddress parameters

The Parlay X 1.0 implementation in WebLogic Network Gatekeeper uses a non-standard structure for the senderName parameter in the sendSms, sendSmsLogo and SendSmsRingtone operations of the SendSMS interface and the senderAddress parameter in the sendMessage operation of the SendMessage interface. The format for this string is as follows:

tel: <mailbox ID>\ <mailbox password>\ tel: <originator address>.

This format continues to be supported for Parlay X 2.1 (It must be used if the application also uses Parlay X 1.0 interfaces.).

The Network Gatekeeper Parlay X 2.1 implementation also supports the specification-defined format for the senderName parameter.

To use this format the mailbox ID and mailbox password must be configured by the operator when the application account is created in the Network Gatekeeper.
Using the Parlay X 2.1 interfaces
Parlay X 1.0 Examples

The following sections describe the Parlay X Web Service examples:

- “About the examples” on page 5-2
- “Send SMS” on page 5-2
- “SMS Notifications” on page 5-3
- “Send MMS” on page 5-5
- “Poll for new MMSes” on page 5-7
- “Receive notifications about new MMSes” on page 5-9
- “Get an MMS by it’s message reference ID” on page 5-10
- “Handling SOAP Attachments” on page 5-11
- “Setting up a two-party call from an application” on page 5-15
- “Handling network-initiated calls” on page 5-18
- “Get location” on page 5-22
- “Get user status” on page 5-24
- “Reserve and charge an account” on page 5-26
About the examples

Below are a set of examples given that illustrates how to use of the Parlay X Web services using AXIS and Java.

Send SMS

Get hold of the Send SMS Web Service.

**Listing 5-1  Get hold of the SMS Service**

```java
SendSmsService sendSmsService = new SendSmsServiceLocator();
java.net.URL endpoint = new java.net.URL(sendSmsWsdlUrl);
SendSmsPort sendSms = sendSmsService.getSendSmsPort(endpoint);
```

The security header is created as outlined in Listing 3-2, “Define the security header,” on page 3-7 and the header is added to the port. The destination address parameter is defined, and the method is invoked. The second parameter of the sendSms method is a combination of the mailbox ID as given by the service provider, the corresponding password and the originator address. The format is "<mailboxID>\<mailboxPassword>\<originator>", for example "tel:50001\thepassword\tel:+46547600". The third parameter is operator-specific, it is used for charging purposes. The message is an ordinary String. An ID is returned. This ID can be used to retrieve delivery status information for the SMS.

**Listing 5-2  Add the security header and send the SMS**

```java
header.setMustUnderstand(true);
((org.apache.axis.client.Stub)sendSms).setHeader(header);
EndUserIdentifier[] eui = new EndUserIdentifier[1];
eui[0] = new EndUserIdentifier();
eui[0].setValue(new org.apache.axis.types.URI("tel:" + destAddress));
String sendID = sendSms.sendSms(eui, myMailbox, "CP_FREE", myMessage );
```
Below is outlined how the ID is used to get hold of delivery status information.

**Listing 5-3  Get delivery status**

```java
DeliveryStatusType[] status = sendSms.getSmsDeliveryStatus(sendID);
System.out.println("Delivery status:");
+status[0].getDeliveryStatus().toString());
```

**SMS Notifications**

SMS notifications are sent asynchronously from WebLogic Network Gatekeeper. This means that the application must implement a Web Service. The initial thing is to start the Web Service server and deploy the implementation of the Web service into the server. The deployment is made using a deployment descriptor that is automatically generated when the Web Service java skeletons are generated.

**Listing 5-4  Start SimpleAxis server**

```java
// start SimpleAxisServer
System.out.println("Opening server on port: "+ port);
ServerSocket ss = new ServerSocket(port);
server.setServerSocket(ss);
server.start();
System.out.println("Starting server... ");
// Read the deployment description of the service
InputStream is = new FileInputStream(deploymentDescriptorFileName);
// Now deploy our  web service
```
The deployment descriptor (deploy.wsdd) was modified to refer to the class that implements the Web Service interface. This class is outlined in Listing 5-5, “Implementation of the smsNotification Web Service,” on page 5-4. The class is based on the auto-generated file SmsNotificationBindingImpl.java.

Listing 5-5   Implementation of the smsNotification Web Service

```java
public class SmsNotification implements org.csapi.www.wsdl.parlayx.sms.v1_0.notification.SmsNotificationPort{
    public void notifySmsReception(String registrationIdentifier,
        String smsServiceActivationNumber,
        EndUserIdentifier senderAddress,
        String message)
        throws java.rmi.RemoteException, ApplicationException {
        System.out.println("->New SMS arrived");
        System.out.println(" Registration Indentifier " + registrationIdentifier );
        System.out.println(" Service activation number " + smsServiceActivationNumber );
        System.out.println(" Sender adress " + senderAddress.getValue() );
        System.out.println(" Message " + message );
    }
}
```

```java
org.apache.axis.client.AdminClient adminClient;
adminClient = new org.apache.axis.client.AdminClient();
System.out.println("Deploying receiver server web service...");
adminClient.process(new org.apache.axis.utils.Options(new String[]
{"-ddd","-tlocal"}), deploymentDescriptorStream);
System.out.println("Server started. Waiting for connections on: " + port);
```
Send MMS

First is a handle to the send MMS service retrieved.

Listing 5-6  Get hold of the Send MMS service

```java
SendMessageServiceLocator sendMmsService = new SendMessageServiceLocator();
java.net.URL endpoint = new java.net.URL(sendMmsWsdlUrl);
SendMessagePort sendMms = sendMmsService.getSendMessagePort(endpoint);
```

The security header is created as outlined in Listing 3-2, “Define the security header,” on page 3-7 and the header is added to the port.

Listing 5-7  Add security header

```java
((org.apache.axis.client.Stub)sendMms).setHeader(header);
```

The contents of the MMS are sent as SOAP attachment in MIME format, consisting of several attachment parts. The method defineattachmentPart described in Listing 5-18, “Define an attachment part,” on page 5-12. Each attachment part is added to the header of the object representing the call.

Listing 5-8  Creating two attachment parts.

```java
int index = 1;
AttachmentPart ap = new AttachmentPart();
ap = defineAttachmentPart("file:../img/afile.jpg",
    "image/jpeg",
    "afile",
    index++);
```
((org.apache.axis.client.Stub)sendMms).addAttachment(ap);
ap = defineAttachmentPart("file:../img/anotherfile.jpg",
   "image/jpeg",
   "anotherfile",
   index++);
((org.apache.axis.client.Stub)sendMms).addAttachment(ap);

When the attachment parts have been defined and added to the call object, the method sendMms is invoked. The second parameter of the sendMessage method is a combination of the mailbox ID as given by the service provider, the corresponding password and the originator address. The format is "<mailboxID>:<mailboxPassword>:<originator>", for example "tel:50001:thepassword:tel:+46547600".

Listing 5-9  Send the MMS

EndUserIdentifier[] eui = new EndUserIdentifier[1];
eui[0] = new EndUserIdentifier();
eui[0].setValue(new org.apache.axis.types.URI("tel:" + destAddress));
String sendID = sendMms.sendMessage(eui, myMailbox, "A subject line",
   MessagePriority.Default,"CP_FREE");
System.out.println("Send ID:" + sendID);

The delivery status of the message can be retrieved as outlined in Listing 5-10, “Check the delivery status,” on page 5-6.

Listing 5-10  Check the delivery status

DeliveryStatusType[] status = sendMms.getMmsDeliveryStatus(sendID);
System.out.println("Delivery status:
   +status[0].getDeliveryStatus().toString());
Poll for new MMSes

An application can poll for new messages. A list of references to the unread messages are returned. The messages are retrieved using these references.

Listing 5-11  Get hold of Receive Message service

```java
receiveMmsService = new ReceiveMessageServiceLocator();
java.net.URL endpoint = new java.net.URL(ReceiveMmsWsdlUrl);
ReceiveMessagePort receiveMms =
    receiveMmsService.getReceiveMessagePort(endpoint);
```

The security header is created as outlined in Listing 3-2, “Define the security header,” on page 3-7 and the header is added.

Listing 5-12  Add the security header

```java
((org.apache.axis.client.Stub)receiveMms).setHeader(header);
```

In Listing 5-13, “Poll and receive new messages,” on page 5-8, the mailbox is polled for new messages. A MessageRefId is retrieved for each new message. The first parameter in getReceivedMessages is specified as &lt;mailboxID&gt;\&lt;mailboxpassword&gt;, for example “tel:10000\thepassword”. The mailboxID and the corresponding password is defined by the service provider.

For each new message, the MessageContext is retrieved. The MessageContext makes it possible to retrieve the response message, where the contents of the MMS is found. The MMS message is found in the SOAP header of the HTTP response of the request to getMessage. The number of attachments are retrieved and also the number of attachment parts. The method
extractAttachment can be implemented as described in Listing 5-21, “Extract the attachments,” on page 5-14.

Listing 5-13  Poll and receive new messages

```java
String[] messageRef;
messageRef = receiveMms.getReceivedMessages(myMailbox,
       MessagePriority.Default);
if (messageRef.length != 0) {
    int i=0;
    // For each new message
    while(i< messageRef.length){
        System.out.println("MessageRef: " + messageRef[i]);
        receiveMms.getMessage(messageRef[i].getMessageRefIdentifier());
        System.out.println("getMessage returned OK");
        // Get the context of the SOAP message
        MessageContext context;
        context = receiveMmsService.getCall().getMessageContext();
        // Get the last response message.
        org.apache.axis.Message reqMsg = context.getResponseMessage();
        // Get the SOAP attachments
        Attachments attachments = reqMsg.getAttachmentsImpl();
        System.out.println("Number of attachments: " +
                           attachments.getAttachmentCount());
        // Get the actual SOAP attachment
        java.util.Collection c = attachments.getAttachments();
        extractAttachments(c);
        i++;
    }
}
```
Receive notifications about new MMSes

Notifications about new MMS are handled in the same manner as for notifications about new SMSes. See Listing , “SMS Notifications,” on page 5-3, the wen service environment is started in the same way as outlined in Listing 5-4, “Start SimpleAxis server,” on page 5-3, and the deployment descriptor is read in the same manner. The deployment descriptor to use is auto generated from the WSDL file for Multimedia Message Notifications. The implementation of the interface is also based on the auto generated class MmNotificationBindingImpl. The adapted implementation of this class file is outlined in Listing 5-14, “Implementation of listener interface,” on page 5-9.

Listing 5-14   Implementation of listener interface

```java
public class MmsNotification implements MmNotificationPort{

    public void notifyMessageReception(String registrationIdentifier,
                               MessageRef messageRef)
        throws java.rmi.RemoteException, ApplicationException {
        System.out.println("->New Message arrived");
        System.out.println("Registration Indentifier  " +
                               registrationIdentifier);
        System.out.println(" Message reference " + messageRef );

        GetMms aMessage = new GetMms();
        aMessage.get(messageRef);
    }
}
```
The parameter messageref can be used to fetch the actual MMS. In the example above this is performed using the class GetMMS. For simplicity this call is not threaded, which it should be in a live system.

**Get an MMS by it’s message reference ID**

The message reference ID can be retrieved notifications from WebLogic Network Gatekeeper, as outlined in “Receive notifications about new MMSes” on page 5-9.

The example below illustrates how to fetch the actual MMS.

A handle to the receive message web service is retrieved as outlined in Listing 5-11, “Get hold of Receive Message service,” on page 5-7.

The security header is created as outlined in Listing 3-2, “Define the security header,” on page 3-7 and the header is added to the port.

The getMessage method is invoked using the message reference ID as an inparameter, see Listing 5-15, “Get Message,” on page 5-10. This ID can be retrieved as outlined in “Receive notifications about new MMSes” on page 5-9.

**Listing 5-15  Get Message**

```java
receiveMms.getMessage(messageRef.getMessageRefIdentifier());
System.out.println("getMessage returned OK");
```

The method returns void, and the contents of the MMS is returned as attachments in the SOAP header of the HTTP response. In Listing 5-16, “Get the context of the SOAP message,” on page 5-10, the SOAP header and the attachments are retrieved.

**Listing 5-16  Get the context of the SOAP message**

```java
MessageContext context = receiveMmsService.getCall().getMessageContext();
// Get the last response message.
org.apache.axis.Message reqMsg = context.getResponseMessage();
// Get the SOAP attachments
```
Attachments attachments = reqMsg.getAttachmentsImpl();
System.out.println("Number of attachments: " + attachments.getAttachmentCount());

The different parts of the attachments are extracted as outlined in Listing 5-13, “Poll and receive new messages,” on page 5-8.

Handling SOAP Attachments

When sending and receiving multimedia messages, the content is handled as attachments in MIME or DIME using SwA, SOAP with Attachments. This technique combines SOAP with MIME, allowing any arbitrary data to be included in a SOAP message.

An SwA message is identical with a normal SOAP message, but the header of the HTTP request contains a Content-Type tag of type “multipart/related”, and the attachment block(s) after the termination tag of the SOAP envelope.

Axis and Java Mail classes can be used to construct and deconstruct MIME/DIME SwA messages.

Encoding a multipart SOAP attachment

Listing 5-17, “Create an attachment,” on page 5-11 gives an example on how to create an attachment and to add it to the SOAP header. Two attachment parts are created.

Listing 5-17  Create an attachment

SendMessageServiceLocator sendMmsService = new SendMessageServiceLocator();
java.net.URL endpoint = new java.net.URL(sendMmsWsdlUrl);
SendMessagePort sendMms = sendMmsService.getSendMessagePort(endpoint);
AttachmentPart ap = new AttachmentPart();
ap = defineAttachmentPart("file:../img/img1.jpg",
    "image/jpeg",
    "img1",
    index++);
((org.apache.axis.client.Stub)sendMms).addAttachment(ap);
ap = defineAttachmentPart("file:../img/img2.jpg",
   "image/jpeg",
   "img2",
   index++);
((org.apache.axis.client.Stub)sendMms).addAttachment(ap);

The method `defineAttachmentPart` is illustrated Listign 5-18, “Define an attachment part,” on page 5-12. The method creates an attachment part. The method is invoked with the following parameters:

- String `mmsInfo`, the full URL to the attachment.
- String `contentType`, the mime type.
- String `contentId`, ID of attachment part, unique within the attachment.
- int `index`, ID of attachment part, unique within the attachment.

Listing 5-18  Define an attachment part

private AttachmentPart defineAttachmentPart(String mmsInfo,
      String contentType,
      String contentId,
      int index){
     AttachmentPart apPart = new AttachmentPart();
     try {
         URL fileurl = new URL(mmsInfo);
         BufferedInputStream bis =
             new BufferedInputStream(fileurl.openStream());
         apPart.setContent(bis, contentType);
         apPart.setMimeHeader("Ordinal", String.valueOf(index));
         //reference the attachment by contentId.
         apPart.setContentId(contentId);
Retrieving and Decoding a multipart SOAP attachment

In order to get a SOAP attachment, the response message is necessary since the SOAP attachment is returned in as an attachment in the SOAP header of the HTTP response. In Listing 5-19, “Get a response message,” on page 5-13, the response message is retrieved.

Listing 5-19  Get a response message

```java
// Get the context of the SOAP message
MessageContext context = receiveMmsService.getCall().getMessageContext();
// Get the last response message.
org.apache.axis.Message reqMsg = context.getResponseMessage();
```

When a handle to the response message is retrieved, the SOAP attachments can be fetched.

Listing 5-20  Get the SOAP attachments

```java
Attachments attachments = reqMsg.getAttachmentsImpl();
java.util.Collection c = attachments.getAttachments();
```
Each attachment, and each attachment part, is traversed and decoded. In the example the attachments are saved to file.

**Listing 5-21  Extract the attachments**

```java
java.util.Collection c = attachments.getAttachments();
Iterator it = c.iterator();

// For each attachment
while( it.hasNext()){
    org.apache.axis.attachments.AttachmentPart p =
    (org.apache.axis.attachments.AttachmentPart)it.next();
    javax.activation.DataHandler dh = p.getDataHandler();
    BufferedInputStream bis = new BufferedInputStream(dh.getInputStream());
    ByteArrayOutputStream bos = new ByteArrayOutputStream();
    while (bis.available() > 0) {
        bos.write(bis.read());
    }
    byte[] pmsg = bos.toByteArray();
    System.out.println("Message Length: "+pmsg.length);
    System.out.println("Content Type: "+p.getContentType());
    System.out.println("Content ID: "+p.getContentId());
    // Convert mime identifier to file extension
    String type =
    p.getContentType().substring(1+p.getContentType().lastIndexOf("/",
    p.getContentType().length()));
    // Save attachment as file
    FileOutputStream fos = new FileOutputStream("ContentID_"
    +p.getContentId()+
    "."+ type);
    fos.write(pmsg);
```
Setting up a two-party call from an application

A two-party call can be set up and controlled from an application using the Parlay X Third Party Call API.

The mechanism is, simplified, as follows:

1. The application orders the call to be set up between a calling party and a called party.
2. The first call leg is set up between the calling party and the MSC or the local exchange.
3. A call attempt is performed to the calling party. At this stage, no action has been performed towards the called party.
4. When the calling party answers, a call attempt is performed to the called party.
5. When the called party answers, the two call legs are connected and the call is processed.

Naturally, this is the ideal situation. A number of other scenarios can be identified, where either the calling party or the called party:

- does not answer.
- is busy because of an already ongoing call.
- is unreachable because of a switched-off mobile terminal.

Using the method `cancelCallRequest`, the request to set up the call can be cancelled. This method is valid until both parties have answered.

When the call has been set up, the status of the call can be monitored using the method `getCallInformation`. The call can also be ended by the application using the method `endCall`.

Below is an example of how to set up a call between two parties.

First, a handle to the Third Party Call service is retrieved.

```java
fos.close();
}
```
ThirdPartyCallService thirdPartyCallService =
    new ThirdPartyCallServiceLocator();
java.net.URL endpoint = new java.net.URL(thirdPartyCallWsdlUrl);
setupCall = thirdPartyCallService.getThirdPartyCallPort(endpoint);

The security header is created as outlined in Listing 3-2, “Define the security header,” on page 3-7 and the header is added to the port.

Listing 5-23   Add security header

((org.apache.axis.client.Stub)setupCall).setHeader(header);

The addresses, in URI-format (tel:<telephone number>) of the calling and the called party are defined and the call is set up. An identifier of the call is returned.

The method makeACall returns before the actual call is setup.

Listing 5-24   Setup the call

String callingParty = "+461111111";
String calledParty = "+4622222222";
EndUserIdentifier[] eui = new EndUserIdentifier[2];
eui[0] = new EndUserIdentifier();
eui[0].setValue(new org.apache.axis.types.URI("tel:" + callingParty));
eui[1] = new EndUserIdentifier();
eui[1].setValue(new org.apache.axis.types.URI("tel:" + calledParty));
m_callID = m_setupCall.makeACall(eui[0], eui[1], "cp_FREE");
The status of the call can be retrieved in order to let the application survey the call processing. The status is one of the following:

- CallInitial
- CallConnected
- CallTerminated

Listing 5-25  Retrieve the status of the call

```java
CallInformationType status;
status = setupCall.getCallInformation(callID);
System.out.println("Call status: " + status.getCallStatus().toString());
```

When the call has terminated, information about the call can be retrieved as illustrated below.

Listing 5-26  Retrieve call information

```java
System.out.println("Call Information");
System.out.println("-start time (YYYY-MM-DD HH:MM:SS:MSMS) " +
status.getStartTime().get(Calendar.YEAR) + "-" +
status.getStartTime().get(Calendar.MONTH) + "-" +
status.getStartTime().get(Calendar.DAY_OF_MONTH) + " " +
status.getStartTime().get(Calendar.HOUR_OF_DAY) + ":" +
status.getStartTime().get(Calendar.MINUTE) + ":" +
status.getStartTime().get(Calendar.SECOND) + ":" +
status.getStartTime().get(Calendar.MILLISECOND));
System.out.println(" -Call duration (s): " +status.getDuration());
System.out.println(" -Call termination cause: " +
status.getTerminationCause().toString());
```
Handling network-initiated calls

A call originating from the telecom network hand be controlled from an application using the Parlay X Network Initiated Third Party Call API.

The mechanism is, simplified, as follows:

1. A call attempt is performed from a calling party to a called party.
2. The application is notified about the call attempt.
3. The application can:
   - reroute, that is change the destination address of the called party.
   - continue, that is leave the call as is and hand over the control of the call to the network.
   - end the call.

Naturally this is the ideal situation. A number of other scenarios can be identified, where the called party:

- does not answer.
- is busy because of an already ongoing call.
- is unreachable because of a switched-off mobile terminal.

It is also possible to route the calling party directly to the application, when he or she goes off-hook, even before a destination number is dialled.

In all above scenarios it is possible for the application to define an operator-specific charging parameter.

The be able to handle network-initiated calls from an application, the calls must be routed to WebLogic Network Gatekeeper. This is normally done by provision data into an MSC or a local exchange.

Notifications on network-initiated calls are sent asynchronously from WebLogic Network Gatekeeper. This means that the application must implement a Web Service. The initial task is to start the Web Service server and deploy the implementation of the Web service into the server. The deployment is made using a deployment descriptor that is automatically generated when the Web Service java skeletons are generated. See Listing 5-4, Start SimpleAxis server on page 3 for an example on how the application is deployed into the Simple Axis Server.

The deployment descriptor (deploy.wsdd) is modified to refer to the class that implements the Web Service interface. This class is outlined in Listing 5-27, “Implementation of the
Handling network-initiated calls

NotifyNwInitCallHandler Web Service,” on page 5-19. The class is based on the auto-generated file NetworkInitiatedCallBindingImpl.java, examples of implementations of the different methods are given below.

Listing 5-27  Implementation of the NotifyNwInitCallHandler Web Service

```java
public class HandleNwInitCall implements NetworkInitiatedCallPort{
    public Action handleCalledNumber(EndUserIdentifier callingParty,
                                      EndUserIdentifier calledParty)
        throws java.rmi.RemoteException, InvalidArgumentException,
               UnknownEndUserException, ApplicationException {

        // See example code for handleCalledNumber below.
    }

    public Action handleOffHook(EndUserIdentifier callingParty)
        throws java.rmi.RemoteException, InvalidArgumentException,
               UnknownEndUserException, ApplicationException {

        // See example code for handleOffHook below.
    }

    public Action handleBusy(EndUserIdentifier callingParty,
                             EndUserIdentifier calledParty) throws
        java.rmi.RemoteException, InvalidArgumentException,
               UnknownEndUserException, ApplicationException {

        // See example code for handleBusy below.
    }

    public Action handleNotReachable(EndUserIdentifier callingParty,
                                      EndUserIdentifier calledParty)
        throws java.rmi.RemoteException, InvalidArgumentException,
               UnknownEndUserException, ApplicationException {

        // Handling of the scenario when the called party is busy.
    }
}
```
public Action handleNoAnswer(EndUserIdentifier callingParty,
   EndUserIdentifier calledParty)
   throws java.rmi.RemoteException, InvalidArgumentException,
           UnknownEndUserException, ApplicationException {
   // Handling of the scenario when there is no answer from the called party.
}

Below is an example of an implementation of the method handleCalledNumber. In the example, the action returned is defined to “Route” and the routing address is set to routeNumber. This means that when the application is triggered, the number dialled by the calling party always is replaced with 12345678. The charging parameter is also set.

Listing 5-28  Example on implementation of handleCalledNumber

public Action handleCalledNumber(EndUserIdentifier callingParty,
   EndUserIdentifier calledParty)
   throws java.rmi.RemoteException, InvalidArgumentException,
           UnknownEndUserException, ApplicationException {
   System.out.println("handleCalledNumber()");
   System.out.println("Calling Party: " + callingParty.getValue());
   System.out.println("Calling Party: " + calledParty.getValue());
   String routeNumber = "tel:12345678";
   String charging_Param = "Free";
   Action action = new Action();
   try {
       EndUserIdentifier eui = new EndUserIdentifier();
       eui.setValue(new org.apache.axis.types.URI("tel:" + routeNumber));
       action.setActionToPerform(ActionValues.Route);
       action.setRoutingAddress(eui);
       action.setCharging(charging_Param);
   }
Below is an example of an implementation of the method handleOffHook. In the example, the action returned is defined to "Route" and the routing address is set to routeNumber. This means that when the calling party goes off hook, the application is triggered, and a call is setup to 12345678. The charging parameter is also set.

**Listing 5-29  Example on implementation of handleOffHook**

```java
public Action handleOffHook(EndUserIdentifier callingParty)
    throws java.rmi.RemoteException, InvalidArgumentException, UnknownEndUserException, ApplicationException {
    System.out.println("handleOffHook()");
    System.out.println("Calling Party: " + callingParty.getValue());
    String routeNumber = "tel:12345678";
    String charging_Param = "Free";
    Action action = new Action();
    try {
        EndUserIdentifier eui = new EndUserIdentifier();
        eui.setValue(new org.apache.axis.types.URI("tel:" + routeNumber));
        action.setActionToPerform(ActionValues.Route);
        action.setRoutingAddress(eui);
        action.setCharging(charging_Param);
    }
    catch (Throwable e){
        return null;
    }
    return action;
}
```
catch (Throwable e){
    return null;
}
return action;
}

Below is an example of an implementation of the method `handleBusy`. In the example, the action returned is defined to “Continue” which transfers the control of the call to the underlying telecom network, which acts on the call as any other call. The charging parameter is also set.

Listing 5-30   Example on implementation of handleBusy

```java
public Action handleBusy(EndUserIdentifier callingParty,
    EndUserIdentifier calledParty)
    throws java.rmi.RemoteException, InvalidArgumentException,
        UnknownEndUserException, ApplicationException {
    System.out.println("HandleBusy()");
    System.out.println("Calling Party: " + callingParty.getValue());
    System.out.println("Called Party: " + calledParty.getValue());
    Action action = new Action();
    action.setActionToPerform(ActionValues.Continue);
    String charging_Param = "Free";
    action.setCharging(charging_Param);
    return action;
}
```

Get location

Get hold of the Terminal location Web Service.
Get location

Listing 5-31  Get hold of the Terminal Location

MobileTerminalLocationService mobileTerminalLocationService = new
MobileTerminalLocationServiceLocator();
java.net.URL endpoint = new java.net.URL(mobileTerminalLocationWsdlUrl);
terminalLocation =
mobileTerminalLocationService.getMobileTerminalLocationPort(endpoint);

The security header is created as outlined in Listing 3-2, “Define the security header,” on page 3-7 and the header is added to the port.

Listing 5-32  Add the security header

header.setMustUnderstand(true);
((org.apache.axis.client.Stub)terminalLocation).setHeader(header);

In Listing 5-33, “Define parameters and get the location,” on page 5-23, the addresses of the requesting party and the requested party are defined, both in URI-format. The desired accuracy is defined and the method is invoked. An object of type LocationInfo is returned.

Listing 5-33  Define parameters and get the location

EndUserIdentifier requested = new EndUserIdentifier();
requested.setValue(new org.apache.axis.types.URI("tel:" +
requestedParty));

EndUserIdentifier requester = new EndUserIdentifier();
requester.setValue(new org.apache.axis.types.URI("tel:" +
requesterParty));

LocationAccuracy accuracy = LocationAccuracy.Medium;
locationInfo = terminalLocation.getLocation(requested, requester, accuracy);

The object holding the returned positioning information is used as outlined in Listing 5-34, “Retrieve the coordinates,” on page 5-24.

Listing 5-34  Retrieve the coordinates

System.out.println("Longitude: " + locationInfo.getLongitude() );
System.out.println("Latitude: " + locationInfo.getLatitude() );
System.out.println("Accuracy:" + locationInfo.getAccuracy().toString() );
java.text.SimpleDateFormat dateFormat =
    new java.text.SimpleDateFormat("EEE, d MMM yyyy HH:mm:ss Z");
java.util.Date theTime = new
    java.util.Date(locationInfo.getDateTime().getTimeInMillis());
System.out.println("Location data updated at: " +
    dateFormat.format(theTime));

Get user status

Get hold of the User status Web Service.

Listing 5-35  Get hold of the User status web service

UserStatusService userStatusService = new UserStatusServiceLocator();
java.net.URL endpoint = new java.net.URL(userStatusWsdlUrl);
userStatus  = userStatusService.getUserStatusPort(endpoint);
The security header is created as outlined in Listing 3-2, “Define the security header,” on page 3-7 and the header is added to the port.

Listing 5-36  Add the security header

```
header.setMustUnderstand(true);
((org.apache.axis.client.Stub)userStatus).setHeader(header);
```

In Listing 5-37, “Define parameters and get the status,” on page 5-25, the addresses of the requesting party and the requested party are defined, both in URI-format. An object of type UserStatusData is returned.

Listing 5-37  Define parameters and get the status

```
EndUserIdentifier requested = new EndUserIdentifier();
requested.setValue(new org.apache.axis.types.URI("tel:" +
   requestedParty));

EndUserIdentifier requester = new EndUserIdentifier();
requester.setValue(new org.apache.axis.types.URI("tel:" +
   requesterParty));

System.out.println("Before getstatus");
UserStatusData userStatusData = userStatus.getUserStatus(requested,
   requester);
```

The object holding the returned status information is used as outlined in Listing 5-38, “Retrieve the status,” on page 5-26. If supported by the network, additional status data is provided
Listing 5-38  Retrieve the status

```java
System.out.println("Status of the terminal is: " + 
userStatusData.getUserStatusIndicator().toString());
System.out.println("Extended Status information :" + 
userStatusData.getAdditionalUserStatusInformation());
```

**Reserve and charge an account**

Get hold of the Reserve Amount Charging Web Service, in order to make reservations and charge the reservation.

Listing 5-39  Get hold of the Reserve Amount Charging web service

```java
ReserveAmountChargingService reserveAmountChargingService = new 
ReserveAmountChargingServiceLocator();
java.net.URL endpoint = new java.net.URL(reserveAmountChargingWsdl1Url);
reserveAmountCharging = 
reserveAmountChargingService.getReserveAmountChargingPort(endpoint);
```

The security header is created as outlined in Listing 3-2, “Define the security header,” on page 3-7 and the header is added to the port.

Listing 5-40  Add the security header

```java
header.setMustUnderstand(true);
((org.apache.axis.client.Stub)reserveAmountCharging).setHeader(header);
```

Get hold of the Amount Charging Web Service, in order to directly charge an account.
Reserve and charge an account

Listing 5-41  Get hold of the Amount Charging web service

AmountChargingService amountChargingService = new
        AmountChargingServiceLocator();
        java.net.URL endpoint = new java.net.URL(amountChargingWsdlUrl);
        amountCharging = amountChargingService.getAmountChargingPort(endpoint);

The security header is created as outlined in Listing 3-2, “Define the security header,” on page 3-7 and the header is added to the port.

Listing 5-42  Add the security header

header.setMustUnderstand(true);
        ((org.apache.axis.client.Stub)amountCharging).setHeader(header);

In Listing 5-43, “Make reservations and charge the reservation,” on page 5-27, the addresses of the party to charge is defined, in URI-format. The amount to reserve (amount1) is defined and the reservation is performed. A reservation ID is returned. This ID is used to identify the charging session in the subsequent reservations via calls to reserveAdditionalAmount. Finally the reservation is charged (it may also be released).

Listing 5-43  Make reservations and charge the reservation

EndUserIdentifier partyToCharge = new EndUserIdentifier();
        partyToCharge.setValue(new org.apache.axis.types.URI("tel:" + endUser));
        java.math.BigDecimal amount1 = new java.math.BigDecimal(10.1);
        String billingText ="Initial reservation";
        String reservationID =
                reserveAmountCharging.reserveAmount(partyToCharge,
                        amount1,
                        billingText);
java.math.BigDecimal amount2 = new java.math.BigDecimal(7);
billingText = "Additional reservation";
reserveAmountCharging.reserveAdditionalAmount(reservationID, 
    amount2, 
    billingText);

java.math.BigDecimal amount = new java.math.BigDecimal(0);
amount.add(amount1);
amount.add(amount2);
billingText = "Charging the reservation";
java.lang.String referenceCode = "Unique referenceCode";
reserveAmountCharging.chargeReservation(reservationID, amount, 
    billingText, referenceCode);

As an alternative an amount can be charged directly without any prior reservations as outlined in

Listing 5-44  Directly charge an account

amount = new java.math.BigDecimal(10.1);
billingText = "Direct debit";
referenceCode = "Unique referenceCode";
amountCharging.chargeAmount(partyToCharge, amount, billingText, 
    referenceCode);
References

API Description Parlay X 1.0 for WebLogic Network Gatekeeper
JavaMail, http://java.sun.com