BEA WebLogic Network Gatekeeper™

Developer's Guide for Parlay X

Version 1.0
Document Revised: March 14, 2005

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Contents

4. Parlay X Examples

A. References

CHAPTER 1

Introduction and Roadmap

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

Document Scope and Audience

The purpose of this guide is to describe how to develop telecom-enabled applications based on the Parlay XAPIs and how to access and use the APIs/interfaces as offered by the 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," informs you about the structure and contents of this document, the used writing conventions, and related documentation.
- Chapter 2, "Introduction and Overall Workflow," gives an introduction to the two main types of WebLogic Network Gatekeeper Web services applications. It also tells you about the programming environment and development workflows.
- Chapter 3, "Parlay X Web Services API," tells you how to add telecom functions to your Web Services Application using Parlay X.
- Chapter 4, "Parlay X Examples," contains examples of usage of Parlay X.

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 for WebLogic Network Gatekeeper*

  The API description describes the Parlay X API and it's implementation in WebLogic Network Gatekeeper.

- *Parlay X Specification, http://www.parlay.org*

  The Parlay X specification describes the Parlay X APIs available for programmers and applications.

CHAPTER 2

Introduction and Overall Workflow

The following sections provide an overview of developing using the Parly X APIs:

About WebLogic Network Gatekeeper Web Services applications

WebLogic Network Gatekeeper Web Services applications are services offering their users access to telecom functionality. The applications can access the telecom functionality through two different Web services APIs. Which API to use depends on the application's needs for network functionality and means of access.

For applications that will provide standardized basic telecom functionality, it is recommended to use the Parlay X APIs.

For applications with a need for more granular control, the Extended Web Services interfaces can be used.

This guide describes how to develop Parlay X applications that connects to the WebLogic Network Gatekeeper. WebLogic Network Gatekeeper acts as aParlay X gateway to the underlying telecom network.

Using the Parlay X APIs you can quickly develop powerful telecom-enabled applications using any programming environment supporting Web Services.

Architecture

Figure 2-2, "Parlay X and JS2SE applications," on page 2-4 illustrates different ways of using the Web Services APIs as provided by the WebLogic Network Gatekeeper, as well as examples of different execution environments for applications.

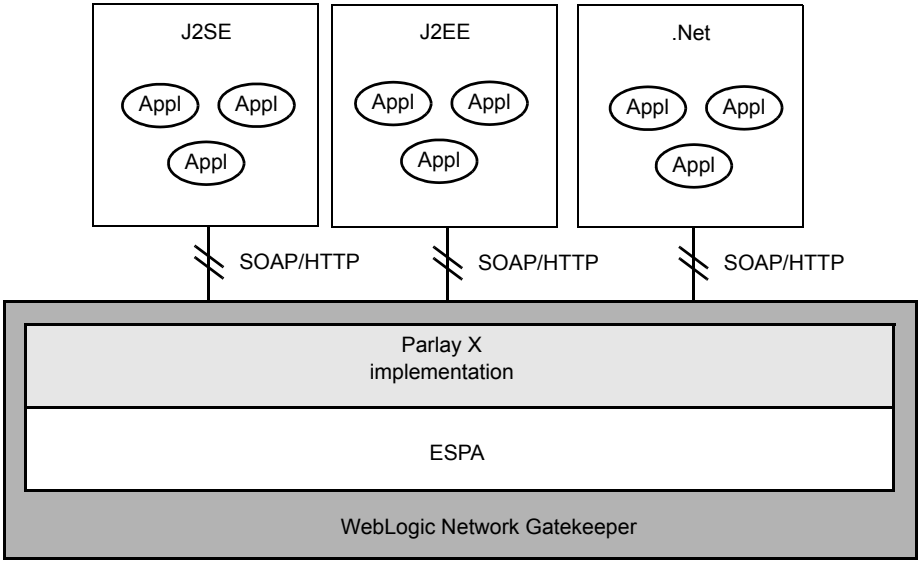
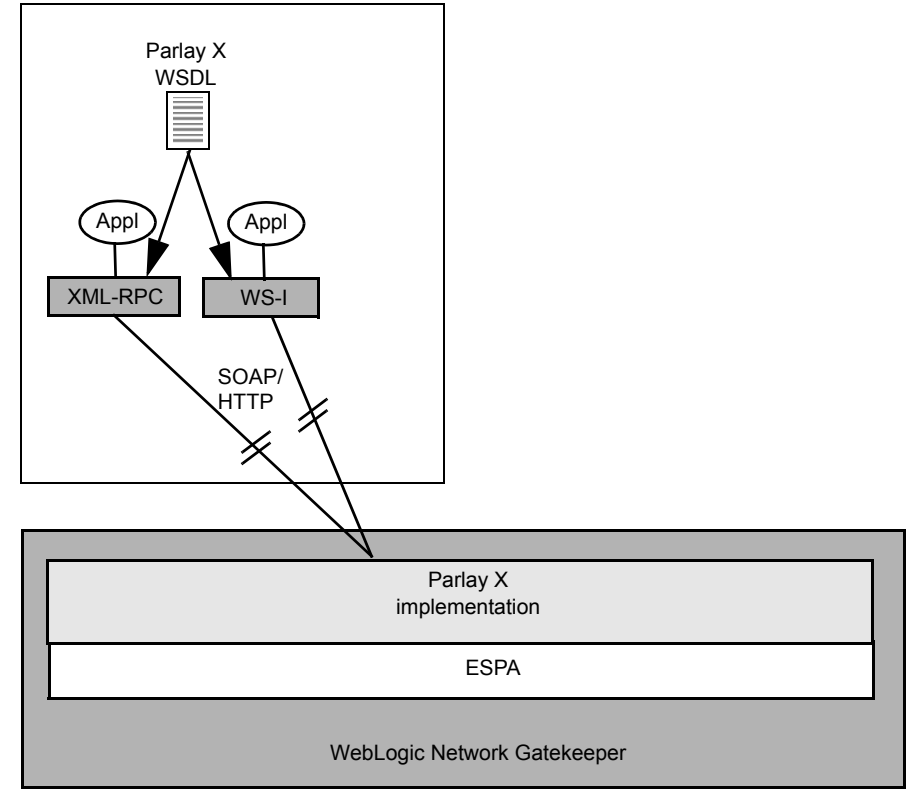


Figure 2-1 Parkay X Web Services interfaces

There are two flavours of Web Services; XML-based RPC and WS-I.



Appl - Application
WS-I - Web Services Interoperability
XML RPC - Handles packaging of SOAP messages
WSDL - Web Services Definition Language

Figure 2-2 Parlay X and JS2SE applications

Web services applications

Web services applications executes in an environment capable of handling Web Services.

The Web Services applications communicates with the 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 methods available.
- is stateless.

Development environment

Below is a description of an example 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.
- Java 2 SDK 1.4.2, see J2SE SDK, http://java.sun.com.
- Axis 1.1, see "Apache Axis, http://ws.apache.org/axis".
- JavaMail API 1.2, see "JavaMail, http://java.sun.com", for messaging applications handling multimedia messages.

The following files, from the Axis distribution, are used:

- axis.jar
- axis-ant.jar
- commons-discovery.jar
- commons-logging.jar
- jaxrpc.jar
- saaj.jar
- wsdl4j.jar

The following JavaMail files are used:

- mail.jar
- activation.jar

Information exchange with the service provider

Before an application is developed, the application developer and the service provider must exchange information regarding resources.

The first step for the application developer is to define which resources to use, call, messaging, location, status, payment etc. and to map these requirements to an APIs that corresponds to these resources.

The next step is to exchange the information according to .

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

Module	API	Information to be provided by the	
		Application developer	WebLogic Network Gatekeeper Operator
Access	Access		Application ID. Service provider ID. Application Instance Group ID. Password for the Application Instance Group.
Network-Initiated Third Party Call	Call	URL of the end-point. If the application always shall be triggered when the calling party goes off-hook..	Access number to the application, if any. Can be a range of numbers.
SMS	Send SMS		Mailbox ID and corresponding password.
	SMS Notification	URL of the end-point.	Access number to the application. Mailbox ID and corresponding password.

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

Module	API	Information to be provided by the	
		Application developer	WebLogic Network Gatekeeper Operator
	Receive SMS		Mailbox ID and corresponding password.
Multimedia Message	Send Message		Mailbox ID and corresponding password.
	Receive Message		Mailbox ID and corresponding password.
	Message Notification	URL of the end-point.	Access number to the application. Mailbox ID and corresponding password.

WebLogic Network Gatekeeper operator must also communicate which services and methods that are supported by the deployment.

In addition to this information, other information related to commercial, security, and privacy regulations must also be exchanged. Examples includes:

- Charging plans to use
- Number of concurrent application instances.
- Amount of usage of the different resources, for example allowed number of send SMS requests.
- Black/white listed addresses.
- Allow/deny lists for user status and user location requests.

Overall development workflows

Below you find overall workflows for development of Web Services applications based on Parlay X.

Two main scenarios are identified:

- 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.
- the application acts as a server and WebLogic Network Gatekeeper is the client. In this scenario, the application in 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.

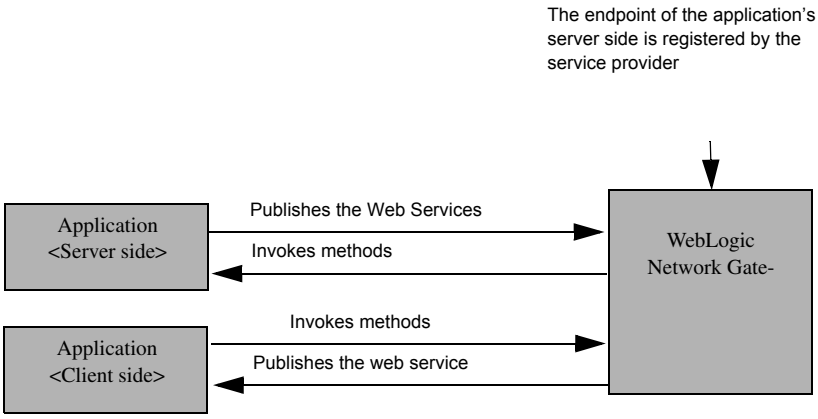


Figure 2-3 Subscribing for notifications

In Parlay X, all methods starting with "handle" or "notify", for example "handleBusy" are methods that WebLogic Network Gatekeeper invokes on the application's server-part. The application's server side implements the interface.

The method invocations are SOAP requests over HTTP, which means that the server part of the application must be capable of handling SOAP requests.

When using Axis, the Simple Axis Server can be used as a SOAP engine during test. In a production system can, for example, Axis in combination with Tomcat be used.

Client-side Web Services using XML based RPC

Below is an overall work sequence for developing telecom enabled Web Services using XML-based RPC:

1. Make sure to 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.
3. Retrieve WSDL files for the telecom services to use.
4. Generate stubs/proxy classes for the language to implement the application in. Use a tool that converts the WSDL into stubs for the preferred language. Examples of such tools are 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, for example the Application Test Environment.
9. Connect the application to WebLogic Network Gatekeeper with a connection to a live telecom network.

Server-side Web Services using XML based RPC

Below is an overall work sequence for developing telecom enabled Web Services using XML-based RPC:

1. Make sure to 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 WSDL files for the telecom services to use.
3. Generate skeleton classes for the language to implement the application in. Use a tool that converts the WSDL into stubs 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 and add telecom functionality 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. Examples of such environments includes Axis.
9. Communicate the end-point of the new application to the service provider.
10. Test the application in a test environment, for example the 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 define a web service that takes care of notifications on new SMS:es from to the applications' server side. The web service is the SMS notification API, containing the method notifySmsReception().

The Simple Axis Server is used as deployment environment for the application.

Below is an outline on the procedure:

1. Generate Java skeletons from the WSDL files:

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

   Note: The Axis files must be in the classpath.

2. Compile and create Jar-files from the skeletons.
3. Move the empty implementation of the generated interfaces to the source directory of the application.

   In the example, the class is named `SmsNotificationBindingImpl.`. When generating skeletons using WSDL2Java, the empty interface implementations are named `<Name of API>BindingImpl`.

4. 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.

5. Compile the application.
6. Verify that the application is deployed correctly by using a Web browser and point 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
7. Provide the service provider with the URL to the Web Service, and the data about the application; application ID and application instance group ID.
8. Run the application. The adapted file deploy.wsdd is used when instantiating the Simple Axis Server.

Testing an application

Figure 2-4, "Application test flow," on page 2-13 shows the application test flow, from the application developers' functional test to deployment in a live network. An application developer can perform functional tests using ATE. The other tests in the flow are performed in cooperation between the application provider and the service provider.

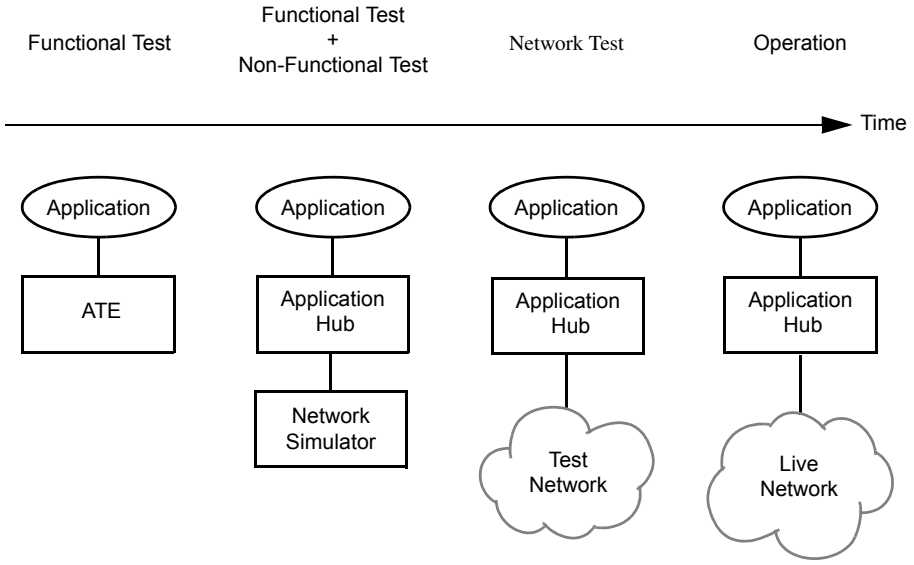


Figure 2-4 Application test flow

When an application shall be tested using the ATE, the application is connected to ATE, which emulates the WebLogic Network Gatekeeper. Before testing in a test telephony network, a network simulator can be used.

An overview of the relation between ATE and WebLogic Network Gatekeeper is shown in Figure 2-5, “ATE in relation to WebLogic Network Gatekeeper,” on page 2-14.

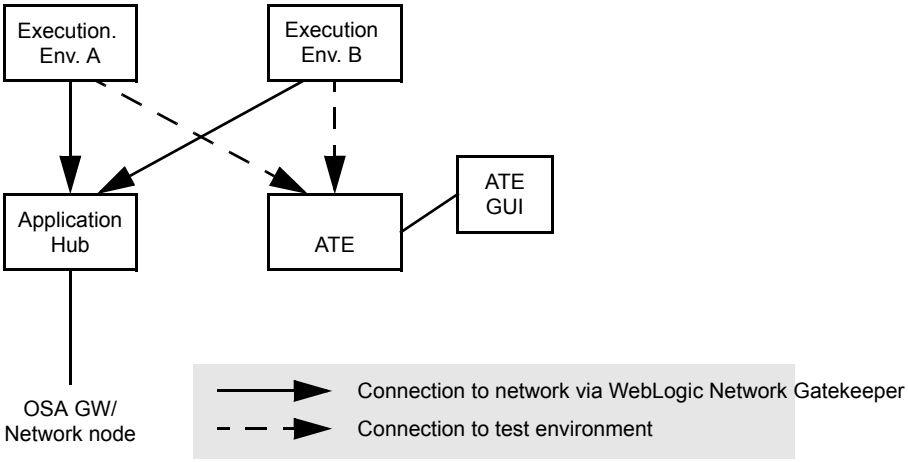


Figure 2-5 ATE in relation to WebLogic Network Gatekeeper

For a applications based on Web Services, the applications uses the endpoints provided by ATE during test. After successful verification, the application uses endpoints provided by WebLogic Network Gatekeeper.

CHAPTER 3

Parlay X Web Services API

The following sections describe the Parlay X Web Services API:

About Parlay X Web Services APIs

The Parlay X Web Services API offers a set of methods that adds telecom functionality to any application that uses Web Services.

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.

All applications, accessing the WebLogic Network Gatekeeper through the Web Services interfaces uses 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 the 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. This mechanism may be extended, as an option using, for example Passport or other extended Kerberos Key Distribution Centre (KDC) authentication solutions.

To run a Parlay X Web Services application, access to either a WebLogic Network Gatekeeper or an ATE is needed, together with a set of WSDL files defining the API, login credentials and IDs of resources to use. These are provided by the service provider.

The Parlay X Web Services APIs are separated in different APIs. Each main component is contained in a specific module. The modules are:

Module	Defines
Third party call	Methods for handling application initiated calls.
Network-initiated third party call	Methods for handling network initiated calls.
SMS	Methods for handling sending and reception of SMS:es.
Multimedia Message	Methods for handling sending and reception of MMS:es.
Payment	Methods for handling charging based on content.
Terminal location	Methods for retrieving the geographical position of a mobile terminal.
User status	Methods for retrieving information on the status of mobile terminals.

In addition there is a an API, Access, that handles login-sessions towards WebLogic Network Gatekeeper. This API is specific for WebLogic Network Gatekeeper, and is not defined in the Parlay X standard. All Parlay X based applications that use the WebLogic Network Gatekeeper as a Parlay X Gateway use this API as an entry point.

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

WSDL files

There are two sets of WSDL files describing the Parlay X API

- SOAP-encoded files. These are known as rpc/encoded files. These files do not use "import" statements. The files can be used in platforms such as .Net.
- WS-I Basic Profile compliant files. These are known as rpc/literal files. This is the expected future-proof set of files. The drawback is that, currently, there are few tools supporting this approach.

The WebLogic Network Gatekeeper supports the Parlay X API using the SOAP encoding (RPC encoding) approach.

By default, the WSDL files can be fetched from:

- http://<URL to WebLogic Network Gatekeeper>/parlayx/servlet/AxisServlet (The Web Services, also the endpoints)
- The WSDL files for the northbound (Listener) interfaces can be fetched from http://<URL to WebLogic Network Gatekeeper>/parlayx/wsdl (definitions of the listener interfaces)

Module	WSDL-file to download	API
Access	Access	Access This API is specific for WebLogic Network Gatekeeper, hence not part of the Parlay X specification.
Third Party Call	ThirdPartyCallPort	Call
Network Initiated Third Party Call	parlayx_network_initiated_call_service	Call
SMS	SendSmsPort	Send SMS
	parlayx_sms_notification_service	SMS Notification
Multimedia Message	SendMessagePort	Send Message
		Receive Message
	parlayx_mm_notification_service	Message Notification
Payment	AmountChargingPort	Amount Charging
	VolumeChargingPort	Volume Charging
	ReserveAmountChargingPort	Reserved Amount Charging
	ReserveVolumeChargingPort	Reserved Volume Charging
Terminal Location	MobileTerminalLocationPort	Terminal Location
User Status	UserStatusPort	User Status

For a description of the methods in each API, see API Description Parlay X for WebLogic Network Gatekeeper.

About the examples

The examples in this chapter are using 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.

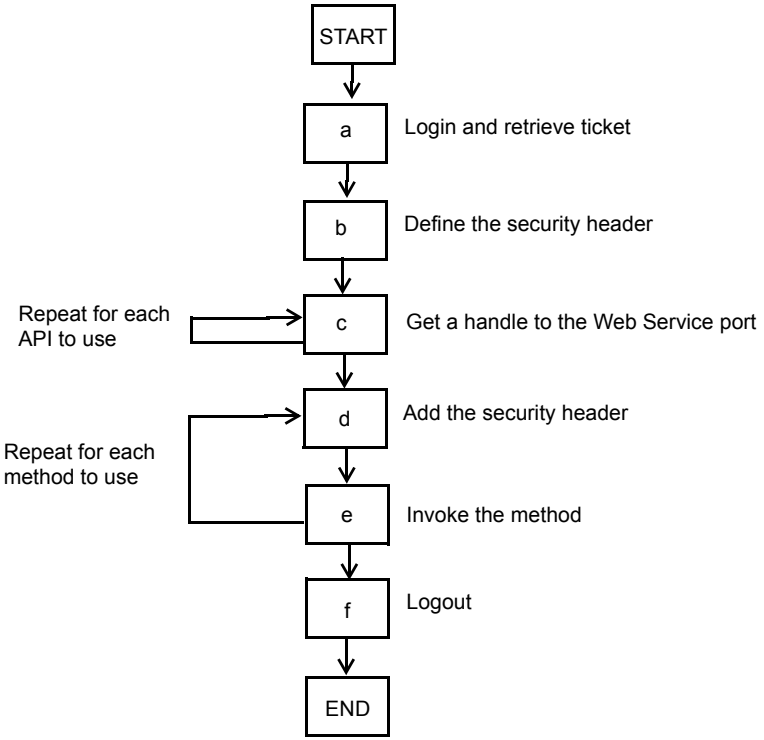


Figure 3-1 Application execution workflow

a. See “Login and retrieve login ticket” on page 3-6.

b. See “Define the security header” on page 3-7.

c. See “Get hold of a 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

To use any Web Service provided by WebLogic Network Gatekeeper, a login ticket is needed. A login is needed to retrieve the ticket. The login ticket identifies the login session. This ticket is valid until a logout is performed. The ticket is sent in each consecutive method invocation to identify the originator of the invoker.

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

Listing 3-1 Login

```
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 provided by the service provider.

Define the security header

The login ticket ID, as retrieved when logging in, is sent in the SOAP header together with a username/password combination for each invocation of a Parlay X web service method.

Listing 3-2 Define the security header

```
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);
```

The login ticket is supplied in the Password attribute. The userName attribute is defined by the service provider, normally in the format <myUserName>@<myapplication>.

The header is defined upon the object representing the Web Service port to use. Also see "Add security header" on page 3-8.

Get hold of a Port

Below is the code for getting hold of a port. The example is using the Send SMS API.

Listing 3-3 Get hold of a port

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

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

Add security header

Adding the security header to a request is straightforward, as illustrated below. For information on how create the header, see "Define the security header" on page 3-7.

Listing 3-4 Add security header

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

Invoke a method

Below it is illustrated how to get hold of a port. The example is using the Send SMS API.

Listing 3-5 Invoke a method

```
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 for WebLogic Network Gatekeeper.

Logout

The logout destroys the login ticket.

Listing 3-6 Logout

```
access.applicationLogout(sessionId);
```

The login Ticket is destroyed and cannot be used in consecutive method invocations.

Access

Access API

This API is specific for WebLogic Network Gatekeeper, and is hence not part of the Parlay X specification.

The following functionality is provided:

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

For a description on how to use 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:

- Get information on if the terminal of a called party is busy (off-hook before the call attempt).
- Get information on if the called party is not reachable. For example, if a mobile terminal is switched off.
- Get information on if a called party does not goes off-hook (No answer). If the called party has not answered after a certain time-interval. The time-interval is defined in the telecom network.
- Get information on if a called party goes off hook.

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

- Continue the call handling with as it normally would be performed in the network.
- Re-route to a called party specified by the application.
- End the call.

All the above events originates from WebLogic Network Gatekeeper. The events comes in form of Web Service method invocations from WebLogic Network Gatekeeper to the end-point of the server-side of the application. The endpoint of the server-side application is defined off-line. The end-point details must be communicated to the service provider.

For more information on implementing Web Services, see "Server-side Web Services using XML based RPC" on page 2-10 and "Example: Server-side Web Service" on page 2-11.

SMS

Send SMS API

This API contains Web Services methods for sending SMS:es.

The following functionality is provided:

- Send SMS.
- Send SMS logo.
- Send SMS ringtone.
- Get the delivery status of an SMS.

SMS Notification API

This API contains Web Services methods for getting notifications on new incoming SMSes.

It is a listener interface, which is implemented on the server-side of the application.

The following functionality is provided:

- Get information on when a new SMS has arrived to WebLogic Network Gatekeeper.

All the above events originates from WebLogic Network Gatekeeper. The events comes in form of Web Service method invocations from WebLogic Network Gatekeeper to the end-point of the server-side of the application. The endpoint of the server-side application is defined off-line. The end-point details must be communicated to the service provider.

For more information on implementing Web Services, see "Server-side Web Services using XML based RPC" on page 2-10 and "Example: Server-side Web Service" on page 2-11.

Receive SMS API

This API contains Web Services methods for fetching SMSes.

The following functionality is provided:

- Get SMSes that has arrived to WebLogic Network Gatekeeper.

Multimedia Message

Send Message API

This API contains Web Services methods for handling sending of multimedia messages. MMS and e-mail is 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 getting 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:

- Get information on when a new multimedia message has arrived.

All the above events originates from WebLogic Network Gatekeeper. The events comes in form of Web Service method invocations from WebLogic Network Gatekeeper to the end-point of the server-side of the application. The endpoint of the server-side application is defined off-line. The end-point details must be communicated to the service provider.

For more information on implementing Web Services, see "Server-side Web Services using XML based RPC" on page 2-10 and "Example: Server-side Web Service" on page 2-11.

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 according to 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 a 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, for example EndUserIdentifier, are defined by the Parlay X standard. These datatypes are defined using WSDL. Also some datatypes are enumerations of values.

Different programming languages uses different approaches for handling variables or classes of these types.

Using Java, it would look like this to define an EnduserIdentifier which holds a telephone number:

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

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

CHAPTER 4

Parlay X Examples

The following sections describe the Parlay X Web Service examples:

About the examples

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

Send SMS

Get hold of the Send SMS Web Service.

Listing 4-1 Get hold of the SMS Service

```
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 4-2 Add the security header and send the SMS

```
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 4-3 Get delivery status

```
DeliveryStatusType[] status = sendSms.getSmsDeliveryStatus(sendID);

System.out.println("Delivery status:"
+status[0].getDeliveryStatus().toString());
```

SMS Notifications

SMS notifications are sent asynchronously from the 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 4-4 Start SimpleAxis server

```
 // start SimpleAxisServer

org.apache.axis.transport.http.SimpleAxisServerserver =
new org.apache.axis.transport.http.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 FileInputStrem(deploymenDescriptorFileName);

// Now deploy our  web service
```

```
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);
```

The deployment descriptor (deploy.wsdd) was modified to refer to the class that implements the Web Service interface. This class is outlined in Listing 4-5, "Implementation of the smsNotification Web Service," on page 4-4. The class is based on the auto-generated file SmsNotificationBindingImpl.java.

Listing 4-5 Implementation of the smsNotification Web Service

```
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 );

   }
}
```

Send MMS

First is a handle to the send MMS service retrieved.

Listing 4-6 Get hold of the Send MMS service

```
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 4-7 Add security header

```
((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 4-18, “Define an attachment part,” on page 4-12. Each attachment part is added to the header of the object representing the call.

Listing 4-8 Creating two attachment parts.

```
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 sendMessage 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 4-9 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 4-10, “Check the delivery status,” on page 4-6.

Listing 4-10 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 4-11 Get hold of Receive Message service

```
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 4-12 Add the security header

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

In Listing 4-13, “Poll and receive new messages,” on page 4-8, the mailbox is polled for new messages. A MessageRefIdentifier is retrieved for each new message. The first parameter in getRecievedMessages is specified as <mailboxID>\<mailboxpassword>, 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 4-21, “Extract the attachments,” on page 4-14.

Listing 4-13 Poll and receive new messages

```
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 attachmnents
      Attachments attachments = reqMsg.getAttachmentsImpl();
      System.out.println("Number of attachments: " +
      attachments.getAttachmentCount());
      // Get the actual SOAP attachmnent
      java.util.Collection c = attachments.getAttachments();
      extractAttachments(c);
      i++;
   }
```

```
    } else {
        System.out.println("No messages found in Mailbox " + myMailbox);
    }
}
```

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 4-3, the wen service environment is started in the same way as outlined in Listing 4-4, "Start SimpleAxis server," on page 4-3, 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 4-14, "Implementation of listener interface," on page 4-9.

Listing 4-14 Implementation of listener interface

```
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 the WebLogic Network Gatekeeper, as outlined in "Receive notifications about new MMSes" on page 4-9.

The example below illustrates how to fetch the actual MMS.

A handle to the receive message web service is retrieved as outlined in Listing 4-11, "Get hold of Receive Message service," on page 4-7.

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 4-15, "Get Message," on page 4-10. This ID can be retrieved as outlined in "Receive notifications about new MMSes" on page 4-9.

Listing 4-15 Get Message

```
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 HTPP response. In Listing 4-16, "Get the context of the SOAP message," on page 4-10, the SOAP header and the attachments are retrieved.

Listing 4-16 Get the context of the SOAP message

```
MessageContext context = receiveMmsService.getCall().getMessageContext();

// Get the last response message.

org.apache.axis.Message reqMsg = context.getResponseMessage();

// Get the SOAP attachmnents
```

```
Attachments attachments = reqMsg.getAttachmentsImpl();

System.out.println("Number of attachments: " +
                     attachments.getAttachmentCount());
```

The different parts of the attachments are extracted as outlined in Listing 4-13, “Poll and receive new messages,” on page 4-8.

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 4-17, “Create an attachment,” on page 4-11 gives an example on how to create an attachment and to add it to the SOAP header. Two attachment parts are created.

Listing 4-17 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 Listing 4-18, "Define an attachment part," on page 4-12. 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 4-18 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);
```

```
    } catch (Exception ex) {
    ex.printStackTrace();
    }
    return apPart;
}
```

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 4-19, "Get a response message," on page 4-13, the response message is retrieved.

Listing 4-19 Get a response message

```
// 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 4-20 Get the SOAP attachments

```
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 4-21 Extract the attachments

```
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);
```

```
    fos.close();
}
```

Setting up a two-party call from an application

A two party call can be set up an 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.
- busy because of an already ongoing call.
- is unreachable because of a switched-off mobile terminal.

Using the method `cancelCallRequest`, the request to setup 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.

Listing 4-22 Get hold of the Third Party Call service

```
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 4-23 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 4-24 Setup the call

```
String callingParty = "+46111111";
String calledParty = "+462222222";
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 4-25 Retrieve the status of the call

```
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 4-26 Retrieve call information

```
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 the 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 the 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 4-4, 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 4-27, “Implementation of the

NotifyNwInitCallHandler Web Service," on page 4-19. The class is based on the auto-generated file `NetworkInitiatedCallBindingImpl.java`, examples of implementations of the different methods are given below.

Listing 4-27 Implementation of the NotifyNwInitCallHandler Web Service

```
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 4-28 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);
```

```
   }
   catch (Throwable e){
      return null;
   }
   return action;
}
```

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 4-29 Example on implementation of handleOffHook

```
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;
}
```

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 4-30 Example on implementation of handleBusy

```
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.

Listing 4-31 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 4-32 Add the security header

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

In Listing 4-33, “Define parameters and get the location,” on page 4-23, 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 4-33 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 4-34, "Retrieve the coordinates," on page 4-24.

Listing 4-34 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 4-35 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 4-36 Add the security header

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

In Listing 4-37, “Define parameters and get the status,” on page 4-25, the addresses of the requesting party and the requested party are defined, both in URI-format. An object of type UserStatusData is returned.

Listing 4-37 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 4-38, “Retrieve the status,” on page 4-26. If supported by the network, additional status data is provided

Listing 4-38 Retrieve the status

```
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 4-39 Get hold of the Reserve Amount Charging web service

```
ReserveAmountChargingService reserveAmountChargingService = new
ReserveAmountChargingServiceLocator();

java.net.URL endpoint = new java.net.URL(reserveAmountChargingWsdlUrl);

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 4-40 Add the security header

```
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.

Listing 4-41 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 4-42 Add the security header

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

In Listing 4-43, "Make reservations and charge the reservation," on page 4-27, 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 4-43 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 4-44, "Directly charge an account," on page 4-28.

Listing 4-44 Directly charge an account

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

APPENDIX A

References

API Description Parlay X for WebLogic Network Gatekeeper

Parlay X Specification, http://www.parlay.org

Apache Axis, http://ws.apache.org/axis

J2SE SDK, http://java.sun.com

JavaMail, http://java.sun.com