Oracle® Communication and Mobility Server

Developer Guide

Release 10.1.3

B31511-01

March 2007

ORACLE®

Oracle® Communication and Mobility Server Developer Guide Release 10.1.3

B31511-01

Contents

B Programming Oracle Diameter Applications

Preface

This preface contains the following topics:

Audience

This guide is intended for developers and programmers who want to use Oracle Communication and Mobility Server to create custom applications.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Related Documents

For more information, see the following documents:

- *Oracle Communication and Mobility Server Installation Guide*
- *Oracle Communication and Mobility Server Administrator's Guide*
- *Oracle Communication and Mobility Server Developer's Cookbook* (available at the Oracle Technology Network at http://www.oracle.com/technology//index.html)
- A.Kristensen, "SIP Servlet API Version 1.0", February 4, 2003. Available in the OCMS SCE installation at C:\Program Files\Oracle\OCMS SCE\doc
- G. Murray, et al., "Java Servlet Specification, Version 2.5", May 11, 2006

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Overview

This chapter through the following sections, describes application development for the Oracle Communication and Mobility Server (OCMS):

- "Introduction to Oracle Communication and Mobility Server"
- "Application Development in OCMS"

Introduction to Oracle Communication and Mobility Server

Oracle Communication and Mobile Server (OCMS) is a carrier-grade SIP application environment for the development, deployment, and management of SIP applications. Built on a standard Java2 Enterprise Edition (J2EE) platform, OCMS is a flexible, scalable environment enabling easy integration of SIP applications and services.

Among the applications that may be developed and deployed on SIP platforms:

- Voice and video telephony, including call management services such as call forwarding and call barring
- Publication of and subscription to user presence information, such as online or offline status, notifications, permission to access a user's status
- Instant messaging
- Push-to-Talk applications, including Push-to-Talk over Cellular (PoC)

OCMS provides standard SIP applications out-of-the-box, including Presence, a combination Proxy and Registrar (Proxy/Registrar), and a SIP message routing application, the Application Router. An integral part of any SIP platform, these applications are automatically installed to the OCMS platform, reducing development resources and deployment time.

OCMS is distinguished by its standards-based, high performance Presence application which is built according to the OMA/Simple Presence Enabler 1.0. The OCMS Presence application is robust enough to support a significant amount of users, while still being a viable solution for ISVs, system integrators, and enterprises requiring an integration platform and an enterprise Presence Server.

Application Development in OCMS

The OCMS environment enables you to develop JSR-116-compliant applications through its support of the following:

- SIP Servlet API
- Parlay X Web Service Interface

SIP Servlet API

OCMS supports development of SIP servlets using the JSR-116 SIP Servlet API. Using these interfaces, you can build a SIP Servlet to create customized SIP-based components that terminate SIP messages, send SIP responses, and proxy SIP messages. For an overview of SIP servlets and the OCMS extensions to the SIP Servlet API, see Chapter 2, "SIP Servlets". For an overall view of developing applications in OCMS, see Chapter 5, "Building a SIP Servlet Application".

Parlay X Web Service Interface

OCMS supports application development using the Parlay X Presence Web Service interface as defined in *Open Service Access, Parlay X Web Services, Part 14, Presence ETSI ES 202 391-14*. Using Parlay X interface, you can build a Web Service that acts as a client for many users by providing them functions for publishing, subscribing and listening to notifies. For more information on OCMS support of the Parlay X Web Service interfaces and developing a Web Service using OCMS SCE as the IDE, see Chapter 6, "OCMS Parlay X Web Services". For examples of building a Parlay X Web Service for the Oracle WebCenter 10.1.3.2 platform, see the *Oracle Communication and Mobility Server Developer's Cookbook*.

Presence Web Services

You can use the Oracle JDeveloper 10*g* to test Presence Web Services. Oracle JDeveloper Release 10.1.3.2 or higher is the preferred development tool for customers creating applications for the WebCenter 10.1.3.2 platform. For more information on using Oracle JDeveloper 10g within OCMS, refer to *Oracle Communication and Mobility Server Developer's Cookbook*.

Oracle Communication and Mobility Server Development Tools

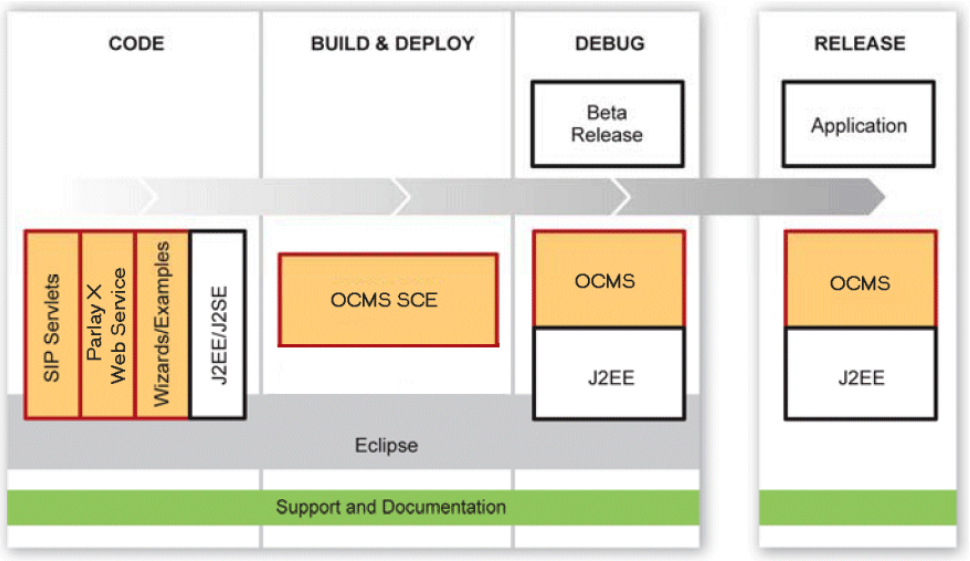
OCMS provides OCMS Service Creation Environment (OCMS SCE) as the primary development tool. Oracle JDeveloper is a secondary tool which is used to test Presence Web Services.

OCMS SCE

OCMS provides the OCMS Service Creation Environment (OCMS SCE), a development tool for SIP servlets and server-side applications. OCMS SCE (SCE), intended for developers on the Windows operating systems, provides tools and APIs for creating services using the OCMS platform. OCMS SCE, which runs on Eclipse, supports your development effort through the *Code*, *Build*, *Deploy* and *Debug* phases of the development cycle (Figure 1–1).

For information on building applications with OCMS SCE, see Chapter 5, "Building a SIP Servlet Application".

Figure 1–1 OCMS SCE Application Development Support



2

SIP Servlets

This chapter, though the following sections, provides an overview of SIP Servlets and the SIP Servlet API as described in JSR-116.

Introduction to SIP Servlets

This chapter provides an overview the SIP Servlet API, the SIP container, the SIP servlet environment, SIP classes and methods, and SIP application configuration.

The SIP Servlet API is defined in JSR116, [5] and describes a standard interface for programming SIP services and applications. A SIP servlet can be compared with a Java HTTP servlet [6] but with SIP protocol-related methods. In fact, the SIP servlet API builds on the general Servlet API [6] in the same way as the HTTP Servlet API

SIP Servlets are grouped into four categories:

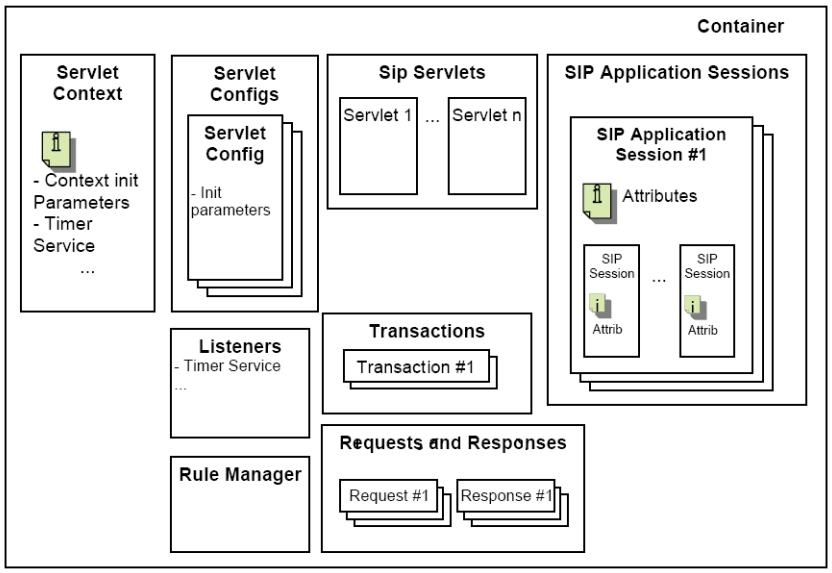
- User Agent Server: A servlet that acts as a User Agent Server (UAS) receives incoming requests and sends responses.
- User Agent Client: A servlet that acts as a User Agent Client (UAC) initiates outgoing requests and receives responses to these requests.
- Proxy: A servlet that acts as a proxy receives incoming requests and proxies them to other end-points (other proxies or a UAS). A proxy servlet will manipulate the request URI and proxy the request to that destination.
- Back to Back User Agent: A servlet that acts as a Back to Back User Agent (B2BUA) acts as both a UAS and UAC on a request. It receives and answers back to one request (first session) and initiates and receives responses on another request (second session).

The SIP Container

This section describes the SIP Container (Figure 2–1) by discussing the following components:

- Servlet Context
- SIP Application Sessions
- Protocol Sessions
- Transactions
- Servlets
- Listeners

Figure 2–1 The SIP Container Object Model



Servlet Context

The Servlet Context is a servlet view of the SIP Application to which it belongs. Using the Servlet Context, the servlet can access such global application services as listeners and context init parameters.

The application global init parameters are set in the deployment descriptor, the `sip.xml` file, and are marked with the `<context-param>` marker. Examples of global appliciation parameters are common resource addresses, like addresses to a database and backend applications.

The Servlet Context also holds all references to any implemented and configured listeners. Session and proxy timeouts are held by the Servlet Context. These are configured in the `sip.xml` file and the session timeout is found within the `<session-config>` and marked in the `<session-timeout>` section. The proxy timeout is found within the `<proxy-config>` and marked in the `<sequential-search-timeout>` section.

The Servlet Context is retrieved by calling the `getServletContext` method from the servlet. For example:

```
ServletContext servletContext = getServletContext();
```

SIP Application Sessions

While an application may involves several types of servlet sessions (for example, SIP and HTTP), these sessions must be related to the same application. This is done through a SIP Application Session. The SIP Application Session can hold multiple protocol sessions and then the protocol sessions can be of the same type or different types (SIP, HTTP).

A SIP Application Session can hold attributes available to all protocol sessions related to that application session. A SIP Application Session can hold multiple protocol sessions. For example, B2BUA creates two sessions that are located in the same SIP Application Session.

A SIP Application Session exists until it times out or when the application explicitly calls the invalidate method. A well-behaved application should always invalidate a session when it does not need it anymore. If the SIP Application Session is invalidated, all Protocol Session objects are invalidated together with it. The timeout value of a session is set with the `<session-timeout>` parameter in the `sip.xml` file. If no timeout value is defined within this element, then the default value for the container is used (the default is 15 minutes).

If a subsequent request arrives after a session has been invalidated, then the server will respond with a 481 "Call transaction does not exist" message. Requests and responses that correspond to a SIP application session can be retrieved with the `getApplicationSession()` method on the request (`SipServletRequest)` or the response (`SipServletResponse`) object.

Protocol Sessions

A Protocol Session is a session for a certain protocol. If an application involves several protocols, the SIP application session would then hold several protocol sessions.

A `SipSession` object can represent a point-to-point SIP relationship, an established SIP dialog, or an initial state SIP dialog. The initial state is described as a request that has been received or created with the `createRequest` method in the `sipFactory` class. The SIP session object is created by the container when an initial request has invoked one servlet.

A request or a response corresponding a SIP session can be retrieved with the `getSession()` method on the request (`SipServletRequest)` or the response (`SipServletResponse`) object.. All of the protocol sessions in an application session can be retrieved with the `getSessions()` method on the `SipApplicationSession` object.

Transactions

A SIP transaction consists of a single SIP request and all responses to that request, including the provisional responses and a final response.

A `SipServletRequest` and a `SipServletResponse` always belong to a SIP transaction. If a servlet tries to send a message that violates the SIP transaction state model, the container throws an `IllegalStateException`. The SIP Servlet API is designed in such a way that a developer does not need to think about transactions, only about requests and responses.

Servlets

Each servlet defined in the deployment descriptor has one instantiation. Because servlets are run in a multi-threaded environment, only data that is common to all requests and responses, such as init parameters, are stored as member variables.

Increased Servlet Modularity

Each servlet, which includes a `<servlet>` and a `<servlet-mapping>` section in the deployment descriptor file (`sip.xml`) also has a `<security-constraint>` section. OCMS makes these servlets reusable through the Servlet Creation wizard in OCMS SCE, because these sections are created together with the servlet.

Listeners

Listeners can be registered to listen on events and invoke a method. Each deployment descriptor can only have one defined listener of each type. Each of these listeners must be implemented by the developer. The different types of listeners are as follows:

- `SipApplicationSessionListener`: The listener for `SipApplicationSession` creation and invalidation. It can be used to ask for a session extension. The `SipApplicationSessionListener` interface must be implemented and the `sip.xml` updated.
- `SipSessionListener`: The listener for changes in active SipSessions in the SIP servlet application. The `SipSessionListener` interface must be implemented and configured in the `sip.xml` file.
- `SipSessionAttributeListener`: The listener for `SipSession` attribute changes. The `SipSessionAttributeListener` interface must be implemented.
- `TimerListener`: The timer service enables delayed actions for a servlet if the `TimerListener` interface has been implemented. Only one timer listener exists per application (`sip.xml`).
- `SipSessionActivationListener`: Objects that depend on a session can listen on their status about being active or passive if the `SipSessionActivationListener` interface is implemented.

SIP Servlets and SIP Applications

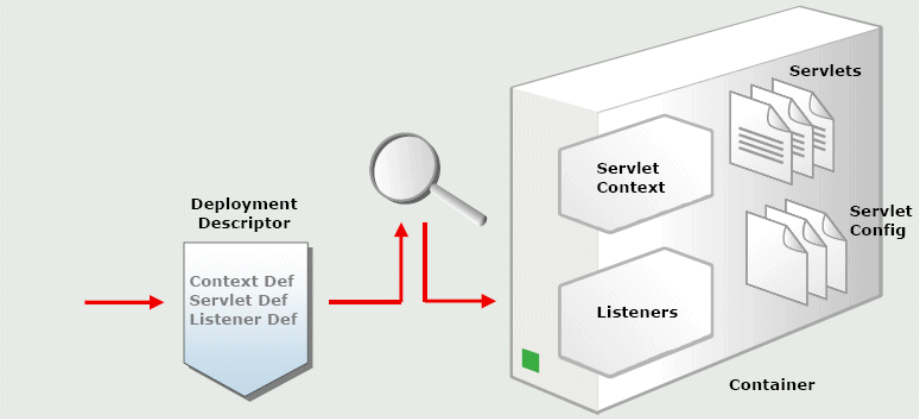
A SIP application is a J2EE-compliant application accessed over the SIP protocol. Applications are triggered by an inbound SIP protocol request, just as Web applications are triggered by an inbound HTTP protocol request.

An application has a protocol interface such as SIP or HTTP (presentation tier) that reaches servlets and other business objects (logic tier) which in turn are managed and have resource connections to the database (data tier). User and application data are accessed through the data tier which is represented by applicable parts from the J2EE specification. Applications are delimited by the scope of the deployment descriptor, the `sip.xml` file.

SIP Servlet Environment

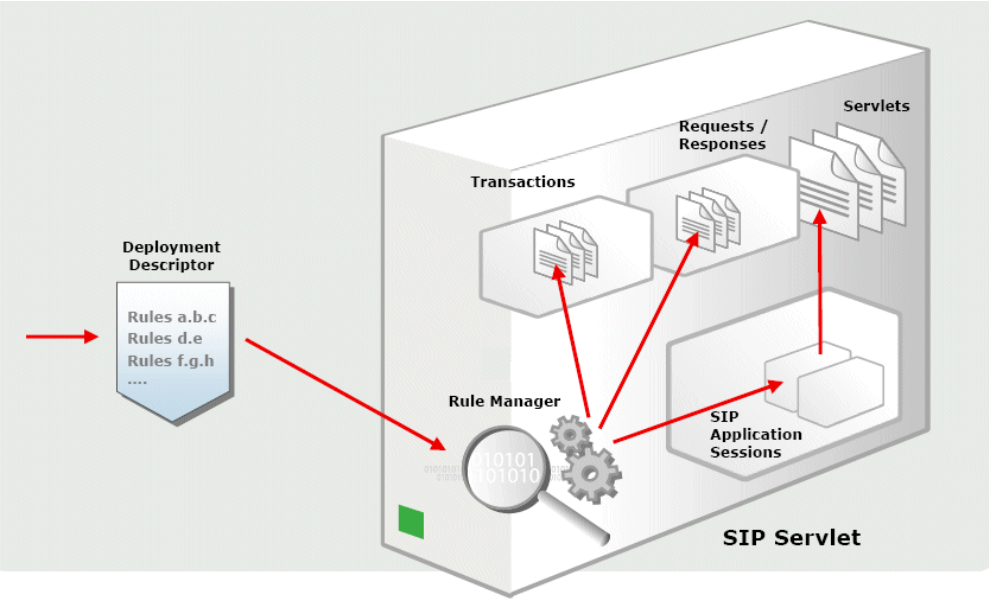
This section describes the environment of a SIP servlet and what occurs during both the startup of the application server (AS) and when a request enters the SIP Container.

Figure 2–2 The SIP Container at Startup



1. The following occurs at startup:
   a. The container reads the deployment descriptor (`sip.xml`).
   b. A Servlet Context is created.
      – The global init parameters are set. These are marked with `<context-param>` in the `sip.xml` file.
      – Session Configuration values are set. These are marked with `<session-config>` in the `sip.xml` file.
      – Proxy Configuration values are set. These are marked with `<proxy-config>` in the `sip.xml` file.
      – The container instantiates the `TimerListener`, `SipApplicationSessionListener`, `SipSessionActivationListener`, `SipSessionAttributeListener`, and the `SipSessionListener`. Each listener has one instantiation (if defined).
   c. A Servlet Config object is created. The Servlet Config holds all init parameters per servlet. These are marked with `<init-param>` and are set per servlet in the `sip.xml` file. It also includes the global init parameters (`context-param`).
   d. The servlets are created and initiated with the init parameters prepared by the Servlet Config. Each servlet is run as one instance. The init method in the servlets is executed.
   e. A SIP Application Manager, which contains a reference to `SipApplicationSessions` and the `SipFactory`, is created.

Figure 2–3 Initial Requests Processed by the SIP Container



2. The container distinguishes between initial requests and subsequent requests, where initial requests invoke a servlet depending on the invoke conditions (or rules) of the servlet and subsequent requests get forwarded directly to the same servlet as invoked for the initial request. At an Initial Request, the following steps occur:

   a. The container chooses a specific application based on the handling described in "Handling Initial Requests".

   b. What servlet to invoke is decided by evaluating the rules of each servlet in the servlet invocation order (Figure 2–3). The rule execution order is decided by the order of the `<servlet-mapping>` order for each servlet in the `sip.xml` file. The servlet with the top-most servlet mapping is the one that get the rules checked first.) For more information, see "Servlet Mapping".

   c. A SIP Application Session is created together with a SIP Session.

   d. A transaction and a request are created and associated with the SIP session object just created.

3. For a subsequent request, one for which the container already holds a dialog (a dialog is defined by the Local-tag + Remote-tag + CallId), the request is forwarded to the same servlet executed at the initial request. The request works with the same Message Context (SIP-Application Session + SIP-Session) as it had in the initial request.

Servlet Mapping

A SIP Application can contain one or more SIP Servlets. When the SIP container receives SIP requests, the container selects a SIP application to serve the request. The SIP application then evaluates its rules (that is, the servlet-mapping defined in the `sip.xml` file) to find the appropriate SIP servlet that responds to the request. The SIP application then uses the first SIP servlet that matches the request.

> **Note:** SIP servlets can forward the request to another SIP servlet by using javax.servlet.ServletRequest.getRequestDispatcher()
>
> Additionally, you can chain applications using the Application Router. Refer to the *Oracle Communication and Mobility Server Administrator's Guide*.

For example, if the SIP container receives an INVITE request, the container selects a SIP Application containing the following SIP Servlets to match the request:

- Servlet 1 - Invoked for INVITE requests.
- Servlet 2 - Invoke for MESSAGE requests.
- Servlet 3 - Invoked for all requests.

The SIP application then attempts to find a match by evaluating its rules from the top to the bottom. In this case, Servlet 1 and Servlet 3 both match the INVITE request. Since only one of these servlets can serve the request and since the application reviews the rules sequentially, the application selects Servlet 1. Servlet 1 starves out Servlet 3. If the SIP container receives a MESSAGE request, then the application invokes Servlet 2. For REGISTER requests, the application invokes Servlet 3, the only servlet able to serve this type of request. If the application did not include Servlet 3, the application's `match` method would return null for the REGISTER request and the SIP container would issue a 403 response, *Application choose not to service the request*, which would be written to the log file.

> **Note:** While a SIP application can invoke only one SIP Servlet for a request, the SIP container can invoke more than one SIP application for an incoming request using the Application Router.

Classes and Methods

This section introduces the classes and methods in the SIP servlet API.

A SIP Servlet is a class that extends the `javax.servlet.sip.SipServlet` class and thereby interacts with a SIP application server to send and receive SIP messages.

Request and Response Handling Methods

The servlet overrides the methods needed for the particular service. The two main methods that the SIP Servlet uses to perform overrides are:

- `protected void doRequest(SipServletRequest req)`
- `protected void doResponse(SipServletResponse resp)`

The `doRequest()` method handles all of the requests and the `doResponse()` method handles all of the responses.

Outside of the `service` method, the `doRequest()` and `doResponse()` are the broadest methods provided by the `SipServlet` class. These methods enable tasks that are independent of the received method or response. You can extend methods to perform specific request and response handling tasks.

Extend the following methods for request handling:

- `doInvite` – for `SIP INVITE` requests
- `doAck` – for `SIP ACK` requests
- `doCancel` – for `SIP CANCEL` requests
- `doBye` – for `SIP BYE` requests
- `doOptions` – for `SIP OPTIONS` requests
- `doRegister` – for `SIP REGISTER` requests
- `doSubscribe` – for `SIP SUBSCRIBE` requests
- `doNotify` – for `SIP NOTIFY` requests
- `doMessage` – for `SIP MESSAGE` requests
- `doInfo` – for `SIP INFO` requests
- `doPrack` – for `SIP PRACK` requests

For response handling, extend the following:

- `doProvisionalResponse` – for SIP 1xx informational responses
- `doSuccessResponse` – for SIP 2xx responses
- `doRedirectResponses` – for SIP 3xx responses
- `doErrorResponse` – for SIP 4xx, 5xx, and 6xx responses

> **Note:** All of these response handling methods, as well as the `doAck` and `doCancel` request handling methods, are empty. The remaining request handling methods respond to a request with a 500 Response (*Internal Server Error*).

Messages

SIP Messages follow the RFC 822 standard. Headers, values and content can be accessed through the methods in the `javax.servlet.sip.SipServletMessage` interface.

```
public interface SipServletMessage
```

Both the `SipServletRequest` and `SipServletResponse` classes extend the `SipServletMessage` interfaces:

- `public interface SipServletRequest extends javax.servlet.ServletRequest,SipServletMessage`
- `public interface SipServletResponse extends javax.servlet.ServletResponse, SipServletMessage`

Requests

`SipServletRequest` and `SipServletResponse` objects in the SIP servlet methods `doRequest` and `doResponse` are implementations of the

`javax.servlet.sip.SipServletRequest` interface and the `javax.servlet.sip.SipServletResponse` interface which both extends the `javax.servlet.sip.SipServletMessage` interface.

A SIP request consists of the following:

- Request line
- Headers
- Empty line
- Message body

All parts of the SIP request for instance request URI, headers and parameters are accessible through these interfaces. The SIP container handles many of these system headers. Applications must not add, delete, or modify the following system headers:

- From
- To
- Call ID
- CSeq
- Via
- Route (except through pushRoute)
- Record Route
- Contact (Contact is a system header field in messages other than REGISTER requests and responses, as well as 3xx and 485 responses)

Responses

Responses to incoming requests are created using the `createResponse` method on the request object. It will automatically create a response with all SIP headers set according to the request. A SIP response has the same structure as a SIP request, except for a *Status* line instead of a request line. When receiving responses through, for instance the `doResponse` method, methods like `getStatus()` can be used on the response object to decide what action to take.

Content

The content in a request or a response can be set with the `setContent()` method as illustrated in Example 2–1.

Example 2–1 Setting the Content of a Request with the setConent Method

```
// Copy of the content from request A to request B
requestB.setContent(requestA.getContent(),
requestA.getContentType());
// Create new content
try
{
  req.setContent("Hello!", "text/plain");
}
catch (UnsupportedEncodingException e)
{
  // Handle the exception
}
```

Manipulating SIP headers

The `javax.servlet.sip.SipServletMessage` includes methods for manipulating SIP headers. The `getHeaderNames()` method is the most general method that returns an iterator of the SIP headers of the message. The setHeader method sets or adds an address for a header. Specifying the name of the header in `getHeaders(name)` method returns another iterator of `String` objects containing all of the values for that header.

For those headers that conform to AddressHeader, the `getAddressHeader(name)` method will return the header parsed as an Address object.

Addresses can be added or set for a header by using `setAddressHeader(name, address)`, `addAddressHeader(name, address, first)`, where name is the name of the header, the address to add, and first is a Boolean. If first is set to true, then the address will be added first, otherwise it will be added last.

Example 2–2 shows how to manipulate SIP headers:

Example 2–2 Manipulating SIP Headers

```
javax.servlet.sip.SipServletRequest req;

// Set a header with the given name and value.
// Overwrites any previous header values.
req.setHeader("Accept", "application/sdp");

// Add a header value to the end of a header
req.addHeader("Accept", "text/html");

// Clear all header values
req.removeHeader("Accept");

// Add a Route header value ahead of the existing Route header
// values.
req.pushRoute(sipURI);

// Get the first route as an Address object
Address first route = req.getAddressHeader("Route");

// Get all address header fields for a header
Iterator addrIter = req.getAddressHeaders("Route");
while (addrIter.hasNext())
{
  Address addr = (Address) addrIter.next();
  // ...
}
```

SipURI

A base class `javax.servlet.sip.URI` exists with two sub classes, `javax.servlet.sip.SipURI` and `javax.servlet.sip.TelURL` which represents SIP URIs and TEL URLs according to RFC 3261, and URLs defined by RFC 2806. The `Address` class stores a SipURI or a TelURL as the address of the user.

URIs can be described as `user@host`, where the user part is either a user name or a telephone number, and the host is a domain name or an IP address.

Parameters can be accessed with getters and setters. The `getParameterNames` method will return an iterator with the names of the parameters. A parameter can be accessed either with the general `getParameter` method or with specific parameter methods like `getUser`, `getHost` and `getLrParam`, which returns the user, the host, and true if the loose route, `lr` is set. Parameters can be set with its corresponding `setMethod()` as shown in Example 2–3.

Example 2–3 Accessing Parameters in SipURI

```
// Create a SipURI, set the port, the lr, and the transport
// protocol parameter
SipURI myURI;
myURI = sipFactory.createSipURI("bob", "10.0.0.10");
myURI.setPort(5072);
myURI.setLrParam(true);
myURI.setTransportParam("udp");
```

Address

SIP addresses found in headers, such as the *From*, *To*, and *Contact* headers, are stored as `javax.servlet.sip Address` objects. As shown in Example 2–4, the object holds, beside the URI, an optional display name and a set of name-value parameters.

Example 2–4 SIP Addresses Stored in java.servlet.sip Address Objects

```
Address contactAddress;
String displayName;
try
{
  // Get the contact address
  contactAdress = req.getAddressHeader("Contact");
  displayName = contactAdress.getDisplayName();
}
catch (ServletParseException spe)
{
  // Handle exception
}

// Create a new address
Address myNewAddress;
myNewAddress = sipFactory.createAddress(URI, "Bob's display
                                                    name");
myNewAddress.setParameter("myparameter","42");
```

The content of `myNewAddress` in the preceding example is as follows:

```
DisplayName            URI                          parameter myparameter
“Bob’s display name”   <sip:bob@example.com;lr>;    42
```

The `Address` class offers the following methods:

- `getDisplayName()`
- `setDisplayName()`
- `getURI()`
- `setURI()`
- `clone()`
- `toString()`

SIP Details

When working with the SIP Servlet API, many SIP details are hidden and performed automatically by the OCMS SIP Application Server.

- When the transport protocol is UDP, retransmissions are handled automatically.
- Dialog-related information such as tags, contacts and CSeq in the SIP messages are handled automatically.
- On outgoing requests, the OCMS server performs DNS lookups and supports NAPTR-, SRV- and A-records (according to RFC 3263).

Storing Data as Session Attributes

The SIP Servlet API enables data to be stored as session attributes. Session attributes can be used to store session specific data to be used in responses and subsequent requests or from a listener class. The attributes are stored and accessed through setters and getters directly on the session object. The session can be either a `SipApplicationSession` or a `SipSession`. An attribute can only be retrieved from the same session it was set. For example:

```
session.setAttribute("key", "value");
session.getAttribute("key"); // Will return "value"
session.removeAttribute("key");
```

Although the session attributes can store any object, the attributes and their keys must be serializable for applications that are distributable (for High Availability). Use the `getAttributeNames` method to retrieve all of the set attributes. For example:

```
Enumeration attributes = session.getAttributeNames();
String attributeName = null;
while (attributes.hasMoreElements())
{
  attributeName = (String) attributes.nextElement();
  Object attribute = session.getAttribute(attributeName);
  log (attribute.toString());
}
```

Adding Configuration Parameters

A servlet can be configured by adding parameters to the `<servlet>` section in the SIP Servlet application descriptor (`sip.xml`). The servlet can then access these parameters through the `getInitParameter` method. Example 2–5 illustrates how to configure the response code.

Example 2–5 Configuring Servlet Parameters

```
package com.mydomain.test;
import javax.servlet.sip.SipServlet;
import javax.servlet.sip.SipServletRequest;
import javax.servlet.sip.SipServletResponse;
import java.io.IOException;
public class MySipServlet extends SipServlet
{
  protected void doRequest(SipServletRequest req) throws IOException
  {
    int responseCode =
              Integer.parseInt(getInitParameter("responseCode"));
    SipServletResponse resp = req.createResponse(responseCode);
    resp.send();
  }
}
```

Configuring SIP Applications in sip.xml

SIP servlets must be declared in the application's deployment descriptor, the `sip.xml` file. The `sip.xml` file enables you to configure the init parameters and the rules that match initial requests with SIP servlets.

The deployment descriptor is divided into the following sections:

- Global init parameters (ServletContext init parameters)

  `<context-param>`
- Session configurations

  `<session-config>`
- Servlet definitions

  `<servlet>`
- Servlet mappings, contains the invocation rules

  `<servlet-mapping>`
- Listeners

  `<listener>`
  - Application life cycle listener classes
  - Error handler
- Security

Setting and Accessing Global Init Parameters

The `<context-param>` and the `<env-entry>` elements provide the means to set and access the application's global parameters, such as the database used with an

application or other resources that are common to the entire application. These elements differ in how the global parameters are accessed.

The <context-param> element declares the servlet's init parameters that are global to the entire Servlet Context. Example 2–6 illustrates setting the database for an application within the <context-param> element.

Example 2–6 Setting a Database Name within the <context-param>

```
<context-param>
  <param-name>Database</param-name>
  <param-value>10.0.0.100</param-value>
  <description>The database to be used with this application.</description>
</context-param>
```

Configuring Application Sessions

SIP Application Sessions are configured within the `<session-config>` clause. The session configuration can configure a session timeout value, which is done within the `<session-timeout>` clause as shown in the following example. The timeout is set in minutes; if it set to 0 or below, an application session will never timeout and must be invalidated explicitly. If no value is set within `<session-timeout>`, the default timeout session set for the sipservletcontainer is used instead (15 minutes).

```
<session-config>
   <session-timeout>10</session-timeout>
</session-config>
```

Defining a Servlet

The `<servlet>` element defines the SIP container's servlets. A servlet requires a servlet name `<servlet-name>`, and a servlet class, `<servlet-class>`. The name must be unique within the application and the class must be the fully qualified name of the servlet class, as illustrated in Example 2–7.

Example 2–7 Servlet Name and Servlet Class

```
<servlet>
   <servlet-name>MyServlet</servlet-name>
   <servlet-class>com.company.example.MyServlet</servlet-class>
</servlet>
```

A servlet can also have its own init parameters, `<init-param>` and a description `<description>`. When the container starts, it only instantiates and calls the `init()` method for the servlets which have set the `<load-on-startup>` element, as shown in Example 2–8.

Example 2–8 Defining a Servlet

```
<servlet>
   <servlet-name>MyServlet</servlet-name>
   <servlet-class>com.mydomain.test.MySipServlet</servlet-class>
   <init-param>
      <param-name>logging</param-name>
      <param-value>true</param-value>
      <description>
         Set if logging should be switched on (true) or not (false).
      </description>
   </init-param>
   <init-param>
      <param-name>infotainment</param-name>
      <param-value>http://www.infotainmentsite.com</param-value>
      <description>
         The site to use as the infotainment content provider.
      </description>
   </init-param>
   <load-on-startup/>
</servlet>
```

Defining the Servlet Mapping

The `<servletmapping>` element of the SIP servlet application deployment descriptor defines the conditions for invoking a servlet. Example 2–9 illustrates how a servlet named *MySipServlet* can only be invoked for incoming MESSAGE requests. For more information, see "Servlet Mapping".

Example 2–9 Servlet Mapping

```
<!-- Servlet Mappings for incoming requests-->
<servlet-mapping>
   <servlet-name>My Sip Servlet</servlet-name>
   <pattern>
      <equal>
         <var>request.method</var>
         <value>MESSAGE</value>
      </equal>
   </pattern>
</servlet-mapping>
```

Creating Rules Using the Request Object Structure

The request object model enables you to define rules for processing SIP requests. This section describes the object model and the predicates for building the rules.

The Request object contains many sub objects, for example SipURI which extends the URI object.

The request object contains the following elements:

- method: The method of the request is represented as String.
- uri: The URI object of the request object, such as SipURI or a TelURI.
- from: The From header address represented as Address.
- to: The To header address represented as Address.

The URI object consists of the URI scheme.

The SipURI object consists of the following elements:

- scheme: The string `sip` or `sips` (where `sips` indicates that TLS should be used).
- user: The user part of the SIP or SIPS URI. If the SIP address is `sip:alice@example.com`, then `request.uri.user` will return `alice`.
- host: The host part of the SIP or SIPS URI. If the SIP address is `sip:alice@example.com`, then `request.uri.host` will return `example.com`.
- port: The SIP URI port. If it is not present, then the default value for UDP and TCP is 5060 and TLS is 5061.
- tel: Returns the user part of the SIP or SIPS URI if the parameter user is set to phone for the URI. The initial visual separators as + and - are stripped away. For example, if the SIP URI is `sip:+12345@example.com;user=phone`, then the `request.uri.tel` would return `12345`.
- param.*name*: The value of the SipURI parameter with the name *name*. For example, given the following request `URI: INVITE sip:23479234@oracle.com;user=phone SIP/2.0`, the `request.uri.param.user` will return `phone`.

The TelURL consists of the following elements:

- scheme: The string, `tel`.
- tel: The tel URL subscriber name.
- param.*name*: the value of the SipURI parameter with the name *name*.

Address consists of the following elements:

- uri: The URI object
- display-name: The `To` or `From` display name of the header.

Conditions

Table 2–1 lists the operators.

Table 2–1 Operators

Condition Name	Description
Equal	Compares the value of a variable with a literal value and evaluates to true if the variable is defined and its value equals that of the literal. Otherwise, the result is false.
Exists	Takes a variable name and evaluates to true if the variable is defined, or to false if the variable has not been defined.
Contains	Returns true if the first argument, a variable, contains the literal string specified as the second argument.
subdomain of	If given a variable denoting a domain name (SIP/SIPS URI host) or telephone subscriber (tel property of SIP or Tel URLs), and a literal value, this operator returns `true` if the variable denotes a sub domain of the domain given by the literal value.

Table 2–2 lists the logical connectors.

Table 2–2 Logical Connectors

Logic	Description
and	Contains a number of conditions and evaluates to true if all of the contained conditions evaluate to true.
or	Contains a number of conditions and evaluates to true if and only if at least one contained condition evaluates to true.
not	Negates the value of the contained condition.

Examples

Example 2–10 describes the parts of a request which are referenced with the different types of variables.

Example 2–10 Referencing Request Elements

```
MESSAGE sip:bob@example.com SIP/2.0
Call-ID: a2412d22-161b-4cff-b4a5-a4c6c1f70f18
To: <sip:bob@example.com>
From: "Alice" <sip:alice@example.com>;tag=1234
Max-Forwards: 70
User-Agent: Oracle-CallTron/4.5.7.1445
Accept: text/plain
CSeq: 2 MESSAGE
Content-Type: text/plain; charset=UTF-8
Content-Length: 13
```

```
Via: SIP/2.0/TCP 10.0.0.20:3094;branch=z9hG4bK-477709c9-c064-4b95-
90bc-7391d0154368.1;rport
Route: <sip:messageapp@sipappserver.com;lr>
Proxy-Authorization: Digest username="alice", realm="example.com",
nonce="MTEzNDQwMzkwMDc3NDJiM2JkY2E1ZmRiY2M4YzBjYzQ2M2VhMTM2NWFlM2Fm",
uri="sip:bob@example.com", qop=auth, nc=00000001, cnonce="443857DE",
response="e5c48d5d2982cf3974260511218a246a",
opaque="4a94690f435b9049b896dce0b6ddb8fe"

The message!
```

In the variables have the values described in Table 2–3.

Table 2–3 Variable Values

Variable	Value
request.method	MESSAGE
request.uri	sip:bob@example.com
request.uri.scheme	sip
request.uri.user	bob
request.uri.host	example.com
request.uri.port	5060
request.uri.tel	
request.from	"Alice" <sip:alice@example.com>;tag=1234
request.from.uri	sip:alice@example.com
request.from.uri.scheme	sip
request.from.user	alice
request.from.port	5060
request.from.display-name	Alice
request.to	sip:bob@example.com
request.to.uri	sip:bob@example.com
request.to.uri.scheme	sip
request.to.uri.user	bob
request.to.uri.port	5060
request.to.display-name	

The following example evaluates to false.

```
<pattern>
   <and>
      <equal>
         <var>request.method</var>
         <value>MESSAGE</value>
      </equal>
      <contains>
         <var>request.from</var>
         <value>tag=1234</value>
      </contains>
      <equal>
         <var>request.uri.host</var>
         <value>otherexample.com</value>
      </equal>
   </and>
</pattern>
```

The first condition, `equal`, evaluates to `true`. The second condition, `contains`, also evaluates to `true`. The third condition ensures that it only serves requests for the `otherexample.com` but the request URI host is `example.com` and then it evaluates to `false` as does the whole rule while the three conditions are all inside the `and` condition.

SIP Servlets in OCMS

This section describes the OCMS extensions to the SIP Servlet API as well as its implementation of it. Sections include:

- "Handling Initial Requests"
- "Implementation Decisions"
- "Multi-Threading"
- "Sip Servlet API Javadoc"
- "External Access to SIP Servlets"
- "OCMS Authentication and Login Modules"

Handling Initial Requests

When the SIP container receives an initial request, it attempts to invoke a SIP application. The SIP container selects the application based on the `appId` parameter. See Chapter 3, "Advanced SIP Servlet Configuration" for information on `appId`. The workflow of the SIP container is as follows:

1. Check if the request contains an indication of which application to invoke.
2. If there is no name of an application to invoke, then the SIP container selects default applications.
3. If the application is named by the request, then the SIP container locates the application and then invokes it.

> **Note:** The container selects a default application when initial requests do not indicate which applications should be invoked.

For example, to respond to an incoming request, a SIP servlet container evaluates the following SIP applications:

- Application A has the following two SIP servlets defined in its `sip.xml`:
  - Servlet 1 matches INVITE requests.
  - Servlet 2 matches MESSAGE requests.
- Application B has the following SIP servlet defined in its `sip.xml`:
  - Servlet 1 matches INVITE requests.
- Application C has the following two SIP servlets defined in its `sip.xml`:
  - Servlet 1 matches REGISTER requests.
  - Servlet 2 matches all requests.
- Application D has the following two SIP servlets defined in its `sip.xml`:
  - Servlet 1 matches a SUBSCRIBE request.
  - Servlet 2 matches a PUBLISH request.
- Application E has the following SIP servlet defined in its `sip.xml`.
  - Servlet 1 matches all requests.

All of these applications have been deployed into the SIP container, but no default application has been specified. The SIP container then designates all of the SIP applications that it locates as the default application. For example, when the SIP container receives an INVITE request which does not designate which application to invoke, the SIP container checks for default applications. However, because none of the applications deployed to the container have been configured as default applications, the container retrieves all of the applications. These applications are in the following order:

- Application B
- Application D
- Application C
- Application E
- Application A

The SIP container reviews the list of applications sequentially, starting with Application B. It queries each application for a servlet that can process the incoming request. If the application contains the appropriate servlet, then the servlet is invoked and no other applications can be considered for this request. Example 2–11 illustrates the concept code for the SIP container's application evaluation process.

Example 2–11 Evaluating SIP Applications for Incoming Requests

```
List allDeployedApplications = retrieve all applications that have been deployed;
for each application in allDeployedApplications do:
   SipServlet servlet = application.match(incoming request);
   if servlet is not null then:
       servlet.processIncomingRequest(request);
       return;
   end if
end for
```

In this case, the SIP container selects Application B to serve the incoming INVITE request. If the SIP container received a PUBLISH request, it retrieves the list again and queries Application B (the first application). Since Application B does not match the request, the SIP Container moves to the next application, Application D. Because this application contains a servlet that processes this request, the SIP container selects Application D.

Application A can never be invoked, as it is the last on the list of deployed applications and will always be starved out by Application C, which accepts all incoming events. Application C starves out all of the subsequent applications as well, as its first servlet processes all REGISTER requests and its second servlet processes all other requests.

Implementation Decisions

The following sections describe implementation options in the SIP Servlet specification:

- "Protocol Sessions"
- "Extended doRequest Methods"
- "Asynchronous Send"

Protocol Sessions

The SIP Servlet specification infers that other protocol sessions, such as HTTP, can be placed in the same SIP Application Session. However, OCMS currently supports only the SIP protocol.

Note: This does not prevent you from creating converged SIP and HTTP applications. Please refer to the *Oracle Communication and Mobility Server Developer's Cookbook* for examples.

Extended doRequest Methods

The OCMS SCE wizard offers a `doPublish()` method. The wizard extends the `doRequest` methods by also dispatching the SIP PUBLISH request (RFC 3903).

For request handling, the `doPublish()` method is used for SIP PUBLISH requests.

See also "Request and Response Handling Methods".

Asynchronous Send

In OCMS, the `send()` method on a `SipServletRequest` never throws an `IOException`. The container posts the request on a send queue and returns it successfully. If the send operation later fails, a final response will be generated by the container. Failure in a DNS lookup returns 408 (*Request Timeout*) or for other types of failures, a 500 (*Server Internal Error*).

Multi-Threading

In OCMS, a servlet runs in a multi-threaded environment and multiple servlets may have access to the same session object at the same time. The request and responses use an available thread from the thread pool of the container. Therefore the developer may be responsible for synchronizing access to sessions and their resources if required by the application logic.

Sip Servlet API Javadoc

The Javadoc can be downloaded from the jcp.org Web page, http://jcp.org/aboutJava/communityprocess/final/jsr116/index.html.

When the download of the zip file is complete, you can update the Javadoc location in Eclipse to include the new location of the Javadoc.

External Access to SIP Servlets

To enable convergent applications between SIP and HTTP, the OCMS Container allows you to get access to the `javax.servlet.sip.SipFactory` by looking it up through JNDI. The SIP Factory will be registered under the same name as the display name of your SIP servlet as illustrated in Example 2–12. The <display-name> in the sip.xml in this case must be "My sip app".

Example 2–12 Accessing the Data for a SIP Session through JNDI

```
InitialContext ic = new InitialContext();
SipFactory sipFactory = (SipFactory)ic.lookup("sip/My sip app");
```

OCMS Authentication and Login Modules

OCMS provides digest authentication (HTTP) against the users, user roles, and user data stored in the TimesTen In-Memory Database or against an external RADIUS (Remote Authentication Dial-In User Service) server. OCMS also supports authentication and authorization services for users provisioned to OID (Oracle Internet Directory).

The login modules, `OCMSLoginModule` and the `RADIUSLoginModule` are custom login modules that act as JAAS security providers to execute digest authentication for SIP applications and basic authentication for J2EE applications. `OCMSLoginModule` authenticates against user data stored in the TimesTen database, while `RadiusLoginModule` authenticates against an external Radius server. Basic authentication is only supported when using the file-based login module, `FileLoginModule`, and is recommended only for test purposes.

For more information, see *Oracle Communication and Mobility Server Administrator's Guide*.

3

Advanced SIP Servlet Configuration

This chapter, describes advanced topics related to SIP Servlets. It contains the following sections:

- Addressing SIP Applications
- Configuring Application Security

Addressing SIP Applications

Because JSR 116 does not state how to address a SIP Application, OCMS includes a parameter called `appId`, which is located either in the URI of the top-most route header (Example 3–1), or in the REQUEST URI (Example 3–2). The SIP container checks both of these locations for this parameter.

If the SIP Container locates `appId`, it interprets the value defined for `appId` as the name of the application to invoke. The value can be defined as either the full application name, or the application alias that is set using the *ApplicationAliases* attribute of the *SipServletContainer* MBean. For example, if the SIP container receives a request that includes the `appId` parameter, the container searches for the value set for the parameter, such as *presence* in Example 3–1. If the container cannot find an application named *presence*, it returns a 403 response (*Forbidden*), because it cannot find an application with this name. However, if the container locates the application defined by the `appId` parameter, it requests this application to process the request using the `match` method described in Servlet Mapping in Chapter 2. If the application does not have a servlet with a matching rule, then the `match` method returns null and the container replies with a 403 response.

Note: `appId` is case-sensitive.

Identifying the appId

The container will only consider the appId if this parameter is included in the top-most-route and this route points to the container, or if there are no route headers present, the request-uri.

> **Note:** You can configure the route to point to the SIP Container through the SipServletContainer MBean.

In Example 3–1, there is a route header present in the request and if the container has been configured to acknowledge the ip-address 192.168.0.100 as an address pointing to itself, it will use the appId in the route-header and invoke the application "presence". However, if this IP address does not belong to the container, it will simply ignore the appId and invoke the default application as previously described.

Example 3–1 appId Defined in the Route Header of an Incoming Request

```
SUBSCRIBE sip:bob@example.com
To: .
From: .
Route: <sip:192.168.0.100:5060;appId=presence>
```

It is important to realize that if the appId would have been present in the request-uri instead, the container would not even had considered evaluating the request-uri for the presence of an appId parameter since the request did contain route-headers. Example 3–2 illustrates how the appId would have been ignored by the container even though the request-uri is pointing to the container itself.

Example 3–2 appId in the Request URI of an Incoming Request

```
SUBSCRIBE sip:bob@192.168.0.100;appId=presence
To: .
From: .
Route: <sip:192.168.0.200:5060>
```

The reason for this is that even though the final destination would be the container itself, this message must first be routed through the server at 192.168.0.200 before coming back to this container where the application presence would be invoked. Of course, depending on the application on the other server, this request may or may not be proxied back to us (that application might e.g. terminate the request by sending a final response to this message).

Configuring Application Security

The deployment descriptor file enables application security through its `<security-constraint>` element. Security is declared per servlet by adding a `<security-constraint>` element to servlets that require authentication and authorization.

The `<proxy-authentication/>` element defines servlet authentication. If a servlet requires authentication, then it can request either 401 (*Unauthorized*), which is the default, or a 407 (*Proxy Authentication Required*) .

A security constraint can hold one or more resource collections, `<resourcecollection>`, each indicating that the servlet requires authentication and the SIP methods that require authentication.

Users can have a single role, several roles, or no roles at all. Each security constraint can set zero or one authorization constraints, `<auth-constraint>`, containing zero or more role names, `<role-name>`, that the authenticated user is authorized against. Authorization can, beside from inside the deployment descriptor, also be checked programmatically from inside a servlet. For example, the `isUserInRole` method on the `SipServletRequest` or the `SipServletResponse` object.

Example 3–3 illustrates a security constraint that requires authentication for `MyServlet` when the request is either an `INVITE` or a `MESSAGE`. There are no authorization constraints to any roles. An unauthenticated user receives 407 (*Proxy Authentication Required*) on its request if `<proxy-authentication/>` is set.

Example 3–3 Configuring Application Security

```
<security-constraint>
   <display-name>MyServlet Security Constraint</display-name>
   <resource-collection>
      <resource-name>MyServletResource</resource-name>
      <description>Securing MyServlet</description>
      <servlet-name>MyServlet</servlet-name>
      <sip-method>MESSAGE</sip-method>
      <sip-method>INVITE</sip-method>
   </resource-collection>
   <proxy-authentication/>
   <auth-constraint>
      <role-name>*</role-name>
   </auth-constraint>
</security-constraint>
```

If a SIP client sends a `REGISTER` request to a server as illustrated in Example 3–4, then a 401 (*Unauthorized*) message is returned to the client. If the authentication succeeds, then the roles of the user are checked against the role names set in the `<auth-constraint>`.

Example 3–4 Configuring Security Constraints

```
<security-constraint>
   <display-name>MyServlet Security Constraint</display-name>
   <resource-collection>
      <resource-name>MyServletResource</resource-name>
      <description>Securing MyServlet</description>
      <servlet-name>MyServlet</servlet-name>
      <sip-method>REGISTER</sip-method>
   </resource-collection>
   <auth-constraint>
      <role-name>Location Service</role-name>
   </auth-constraint>
</security-constraint>
```

If the user re-sends the `REGISTER` request which is subsequently authenticated, then then the container checks the roles of the user against the required `Location Service` role. A 403 (*Forbidden*) message is sent as a response if the user does not have the appropriate role.

4

Programming Guidelines

This chapter, describes programming guidelines for SIP Servlet applications. It contains the following sections:

Introduction

For an application to take full advantage of OCMS as a JSR 116 SIP Servlet Application platform, observe the following recommendations.

Marking Applications as Distributable

For stateful applications, that is applications that save state in SipSessions or SipApplicationSessions, to be highly available in a SIP Application Server cluster, the sip.xml as well as the web.xml file must contain the <distributable> element. Marking applications as distributable will make sure that the application and session state is replicated to cluster nodes that will be able to resume execution of the session in the event of a failure of a cluster node.

There are performance implications related to how session state is replicated in a distributable environment. Replication is triggered each time there is a setAttribute() call on the session object, so large numbers of such calls in a servlet may impact performance.

For OC4J there are additional requirements if the application is packaged as an Enterprise Archive. See the *Oracle Communication and Mobility Server Administration Guide* for details.

Storing Data in Application Sessions

All data that must persist for a session must be stored in the SIP Application session explicitly and must be serializable.

Avoiding Static Data

As a corollary, to the previous recommendation, avoid using static variables in an application, instead use standard J2EE mechanisms such as EJB or database storage to achieve persistent storage of data that can be made available to another cluster node in the event of a failure.

Avoiding Blocking Calls

All blocking calls as part of the invocation of a SIP Servlet application should be avoided. Blocking calls include Synchronous Remote procedure calls and synchronous database calls.

Invalidating the SipApplicationSession and SIPSession

Remember to invalidate the SipApplicationSession in order to free up memory as soon as possible when the application has finished. For individual SIP Sessions, you should invalidate them as soon as they are finished as well. Make sure there are no active SIP Sessions when you invalidate a SIP Application Session as the owned SIP Sessions will be invalidated as well.

Monitoring the Memory Usage

Monitor the memory usage for the data you want to store in session objects. Make sure there is sufficient memory for the number of sessions created before the sessions time out.

Avoiding Storing Shared Resources in Sessions

Objects that are stored in the session objects will not be released until the session times out (or is invalidated). If you hold any shared resources that have to be explicitly released to the pool before they can be reused (such as a JDBC connection), then these resources may never be returned to the pool properly and can never be reused.

Avoiding Creating Threads

Use the available mechanisms to create an event driven model for your application instead of creating threads to perform work outside of the activation model for the containers.

Creating B2BUA Applications

For B2BUA Applications, use the createRequest that clones the relevant fields of the original request:

```
SipFactory.createRequest(SipServletRequest origRequest,
boolean sameCallId)
```

It will clone the request with the following modifications:

- The From header field of the new request has a new tag chosen by the container.
- The To header field of the new request has no tag.
- The Call-ID will be new (or duplicated in sameCallId is true)
- The Record-Route and Via header fields are not copied. The container will add its own Via header field to the request when it is actually sent outside the application server.
- For non-REGISTER requests, the Contact header field is not copied but is populated by the container

5

Building a SIP Servlet Application

This chapter illustrates how to build a SIP Servlet application through OCMS SCE by creating a new project, importing an existing project, or by importing one of the example project provided with OCMS SCE. For more information on the OCMS SCE example projects and other SIP servlets, including source code, refer to *Oracle Communication and Mobility Server Developer's Cookbook*.

This chapter consists of the following sections:

Prerequisites

Before building a SIP servlet application, ensure that you have completed the following prerequisite steps:

1. Installing Eclipse Web Tools Platform 1.5.

   Refer to *Oracle Communication and Mobility Server Installation Guide* for installation instructions. For information on Eclipse functionality, refer to the tutorial at http://www.eclipse.org/webtools/community/communityresources.html.

2. Installing OCMS with the OCMS Service Creation Environment component. Select from one of the following installation mode options: OC4J mode (Oracle Containers for J2EE must already be installed) or standalone mode (a standalone version of Oracle Containers for J2EE is installed with OCMS).

Refer to *Oracle Communication and Mobility Server Installation Guide* for installation instructions.

3. Initializing the OCMS Service Creation Environment, and adding OCMS as a Server in OCMS SCE. Refer to *Oracle Communication and Mobility Server Installation Guide*.

SIP Application Development Process

Applications developers can create a fully-converged SIP and HTTP application with OCMS by performing the following tasks.

For more information on performing these tasks, refer to Chapter 5, "Building a SIP Servlet Application".

1. Create a SIP servlet
   a. Create a SIP Servlet Project.
      – Create a SIP servlet.
      – Define the SIP servlet's initialization parameters (servlet definitions) and invocation rules (servlet mappings) in the application of the SIP Servlet in each of the deployment descriptors created.
2. Create a SIP application
   a. Create a SIP Application Project (ssr).
   b. Choose the servlets to include.
   c. Update the deployment descriptor with the following:
      – Servlet Context init parameters
      – Session configurations
      – Application lifestyle listener classes
      – Error handler
3. Create Utility Classes (business logic)
   a. Create an EJB Project.
   b. Create EJBs.
4. Create static resources and content such as Text and speech announcement.
5. Create Web functionality
   a. Create a Web Application Project.
   b. Create servlets and JSPs.
6. Create an Enterprise Application
   a. Create an Enterprise Application Project.
   b. Choose your applications that should form the Enterprise Application. (SIP Application, Business Logic, Web Application).
7. Test your application
   a. Deploy the application to the application server.
   b. Test and debug the application.

Creating a New Dynamic Web Project with SIP Support

Perform the following procedure to create a new project in Eclipse for deployment to OCMS:

1. Close the *Welcome* window of Eclipse if it is shown.
2. Select the Eclipse Java perspective by selecting **Window**, **Open Perspective**, and **Java**.
3. Create a new project by selecting **File, New, Project, OCMS SCE, Dynamic Web Project with SIP Support**.
4. Enter the project name.
5. Select the target runtime from the drop-down menu.

   This runtime will be preconfigured if the OCMS Server has been added as described in Prerequisites.
6. Click **Finish**.

   A new Java Project is created in the Package Explorer. Access the Package Explorer by selecting **Window**, **Show View**, **Package Explorer**. To deploy your project, perform the steps described in Deploying a SIP Application to OCMS.

Importing an Existing Project

Perform the following procedure to import an existing project into Eclipse for deployment to OCMS:

> **Note:** When importing a new project into Eclipse for use with the OCMS SCE, the project must be located in the workplace directory. Otherwise the deployment into OCMS will not work.

1. Close the “welcome” window of Eclipse if it is shown.
2. Select the Eclipse Java perspective by selecting **Window**, **Open Perspective**, and **Java**.
3. Create a new project by selecting **File, Import, Web**, WAR file.
4. Navigate to the Eclipse workspace directory and locate the WAR file to be imported.
5. Select the project.
6. Select "OCMS SipAS - OC4J 10.3.2" as the target runtime from the drop-down menu.

   This runtime will be preconfigured if the OCMS Server has been added as described in Prerequisites.
7. Click **Finish**.

   A new Java Project is created in the Package Explorer. Access the Package Explorer by selecting **Window**, **Show View**, **Package Explorer**. To deploy your project, perform the steps described in Deploying a SIP Application to OCMS.

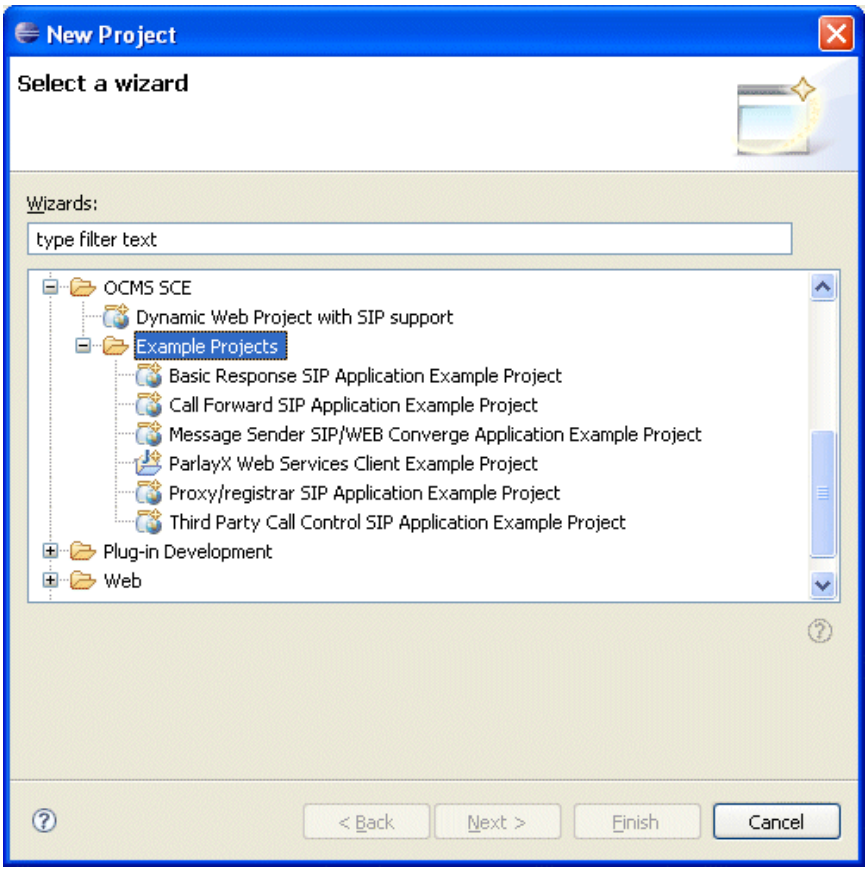
Importing Example Projects

The section describes how to import and configure the example projects provided by OCMS SCE. There is a Readme file provided with all the example projects (except the Basic Response project) that includes steps to deploy and use the project. Refer to the Readme file bundled with the project for additional information.

OCMS SCE provides the following six example projects (Figure 5–1):

- **Basic Response SIP Application Example Project**: This SIP servlet receives an incoming MESSAGE and sends back a response to the sender.
- **Call Forward SIP Application Example Project**: This SIP servlet provides Call Forward on No Answer functionality.
- **Message Sender SIP/WEB Converged Application Example Project**: This application includes the Message Sender SIP Servlet and a web page from which a message can be sent to a SIP address.
- **Parlay X Web Services Client Example Project**: This is a Parlay X Web Services client example project that uses the Presence Supplier interface.
- **Proxy/registrar SIP Application Example Project**: This SIP application can register and authenticate users and can set up voice and video calls as well as allow instant messaging between registered users.
- **Third Party Call Control SIP Application Example Project**: This application is a Third Party Call Control SIP example application that performs basic Click-to-Call functionality, implements flow I in RFC 3725, and provides a Web interface.

Figure 5–1 OCMS SCE Example Projects



Importing the Basic Response SIP Application Example Project

Perform the following procedure to import the Third Party CAll Control example project into Eclipse SCE:

1. Close the "welcome" window of Eclipse if it is shown.
2. Select the Eclipse Java perspective by selecting **Window**, **Open Perspective**, and **Java**.
3. Import the example SIP servlet project by selecting **File, New, Project, OCMS SCE, Example Projects, Basic Response SIP Application Example Project**.
4. Enter basicresponseservlet as the project name and click **Finish**.

   A new Java Project is created in the Package Explorer. Access the Package Explorer by selecting **Window**, **Show View**, **Package Explorer**. To deploy your project, perform the steps described in Deploying a SIP Application to OCMS.

Importing the Call Forward SIP Application Example Project

Perform the following procedure to import the Third Party CAll Control example project into Eclipse SCE:

1. Close the "welcome" window of Eclipse if it is shown.
2. Select the Eclipse Java perspective by selecting **Window**, **Open Perspective**, and **Java**.
3. Import the example SIP servlet project by selecting **File, New, Project, OCMS SCE, Example Projects, Call Forward SIP Application Example Project**.
4. Enter *callforwardservlet* as the project name and click **Finish**.

   A new Java Project is created in the Package Explorer. Access the Package Explorer by selecting **Window**, **Show View**, **Package Explorer**. To deploy your project, perform the steps described in Deploying a SIP Application to OCMS.

> **Note:** When importing a new project into Eclipse for use with the OCMS SCE, the project must be located in the workplace directory. Otherwise the deployment into OCMS will not work.

Importing the Message Sender SIP/Web Converged Application Example Project

Perform the following procedure to import the Third Party CAll Control example project into Eclipse SCE:

1. Close the "welcome" window of Eclipse if it is shown.
2. Select the Eclipse Java perspective by selecting **Window**, **Open Perspective**, and **Java**.
3. Import the example SIP servlet project by selecting **File, New, Project, OCMS SCE, Example Projects, Message Sender SIP/Web Converged Application Example Project**.
4. Enter *messagesenderservlet* as the project name and click **Finish**.

   A new Java Project is created in the Package Explorer. Access the Package Explorer by selecting **Window**, **Show View**, **Package Explorer**. To deploy your project, perform the steps described in Deploying a SIP Application to OCMS.

Importing the Parlay X Web Services Client Example Project

Perform the following procedure to import the Third Party CAll Control example project into Eclipse SCE:

1. Close the "welcome" window of Eclipse if it is shown.
2. Select the Eclipse Java perspective by selecting **Window**, **Open Perspective**, and **Java**.
3. Import the example SIP servlet project by selecting **File, New, Project, OCMS SCE, Example Projects, Message Sender SIP/Web Converged Application Example Project**.
4. Enter *messagesenderservlet* as the project name and click **Finish**.

   A new Java Project is created in the Package Explorer. Access the Package Explorer by selecting **Window**, **Show View**, **Package Explorer**. To deploy your project, perform the steps described in Deploying a SIP Application to OCMS.

Importing the Proxy/Registrar Example Project

Perform the following procedure to import the Third Party CAll Control example project into Eclipse SCE:

1. Close the "welcome" window of Eclipse if it is shown.
2. Select the Eclipse Java perspective by selecting **Window**, **Open Perspective**, and **Java**.
3. Import the example SIP servlet project by selecting **File, New, Project, OCMS SCE, Example Projects, Proxy/registrar SIP Application Example Project**.
4. Enter *proxyregistrarservlet* as the project name and click **Finish**.

   A new Java Project is created in the Package Explorer. Access the Package Explorer by selecting **Window**, **Show View**, **Package Explorer**. To deploy your project, perform the steps described in Deploying a SIP Application to OCMS.

Importing the Third Party Call Control Example Project

Perform the following procedure to import the Third Party CAll Control example project into Eclipse SCE:

1. Close the "welcome" window of Eclipse if it is shown.
2. Select the Eclipse Java perspective by selecting **Window**, **Open Perspective**, and **Java**.
3. Import the example SIP servlet project by selecting **File, New, Project, OCMS SCE, Example Projects, Third Party Call Control**.
4. Enter *TpccServlet* as the project name and click **Finish**.

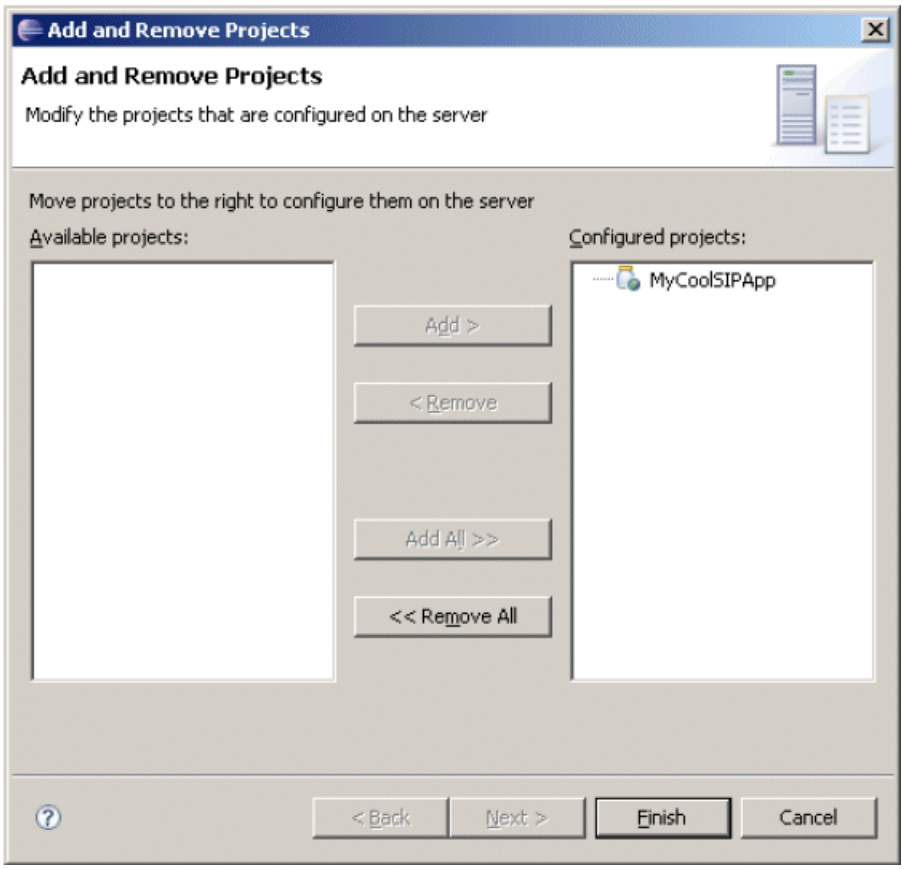
   A new Java Project is created in the Package Explorer. Access the Package Explorer by selecting **Window**, **Show View**, **Package Explorer**. To deploy your project, perform the steps described in Deploying a SIP Application to OCMS.

Deploying a SIP Application to OCMS

Perform the following procedure to deploy SIP applications to OCMS:

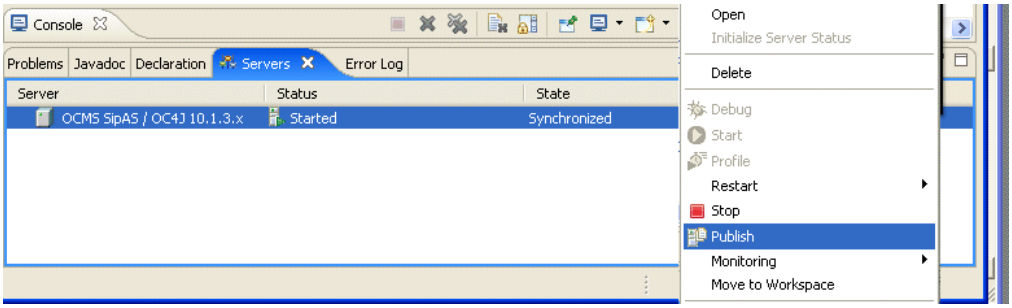
1. Associate the projects to be deployed to a server by performing the following steps:
   - In the Servers view right-click "OCMS SipAS - OC4J 10.1.3" and select **Add and Remove Projects** (Figure 5–2).

Figure 5–2 Adding Projects to a Server



   - In the Add or Remove Projects dialog box select the projects and use the Add button to add your projects to the list on the right called "Configured projects."
   - Select Finish to complete the association of the projects to the OCMS server.
2. Right-click the server in the Servers view and select **Publish** to start and deploy the SIP applications (Figure 5–3).

Figure 5–3 Selecting Publish in the Servers View to Deploy Applications



The application is ready to be tested.

Testing an Application

After you have built and deployed your application you must test its functionality. When starting OCMS the application will automatically be started. Verify that the application has started correctly by examining the system logging files.

> **Tip:** You can obtain a tool to view frequently updated log files. Install one of the following log viewers tools, if you do not already have one:
>
> - LogWatcher, downloadable from `http://graysky.sourceforge.net` or
> - Tail XP, downloadable from `http://infiero.com/tailxp`.

Changing the Logging Level

Setting the log level to DEBUG provides a more detailed level of information for the developer. Using the SIPServletContainerLogging MBean, set the system log level to DEBUG by entering DEBUG as the value of the System attribute. See *Oracle Communication and Mobility Server Administrator's Guide* for more information.

Viewing the System Log File

Start the log file viewer and view the system log file, `system.log`. Log files are located under <ocms_home>/j2ee/home/log/sdp.

Starting the OCMS Server in Eclipse

Start the server by opening the Server view. Select **Window**, **Show view**, **Other**… and select **Server**, **Servers** and click **OK**. Right click on the server and from the context menu select **Debug** to start the server in debug mode.

Eclipse changes the active view to the console view. This view is active until the server has changed the log output to the `system.log` file, as indicated by a message (Configuring from URL:...) in the preceding Console view.

Eclipse will start when the server has changed back to the server view. Look for the following message in the console: "<date> <time> Oracle Containers for J2EE 10g (10.1.3.2) initialized."

> **Note:** Even if a SIP servlet fails to startup the server will display that it is started. You must view the log file to verify whether the SIP servlet deployed successfully.

Testing a Third Party Call Control Servlet

To test an application, open a Web browser and navigate to the URL of the application on the OCMS server.

For example, to test the Third Party Call Control servlet perform the following steps:

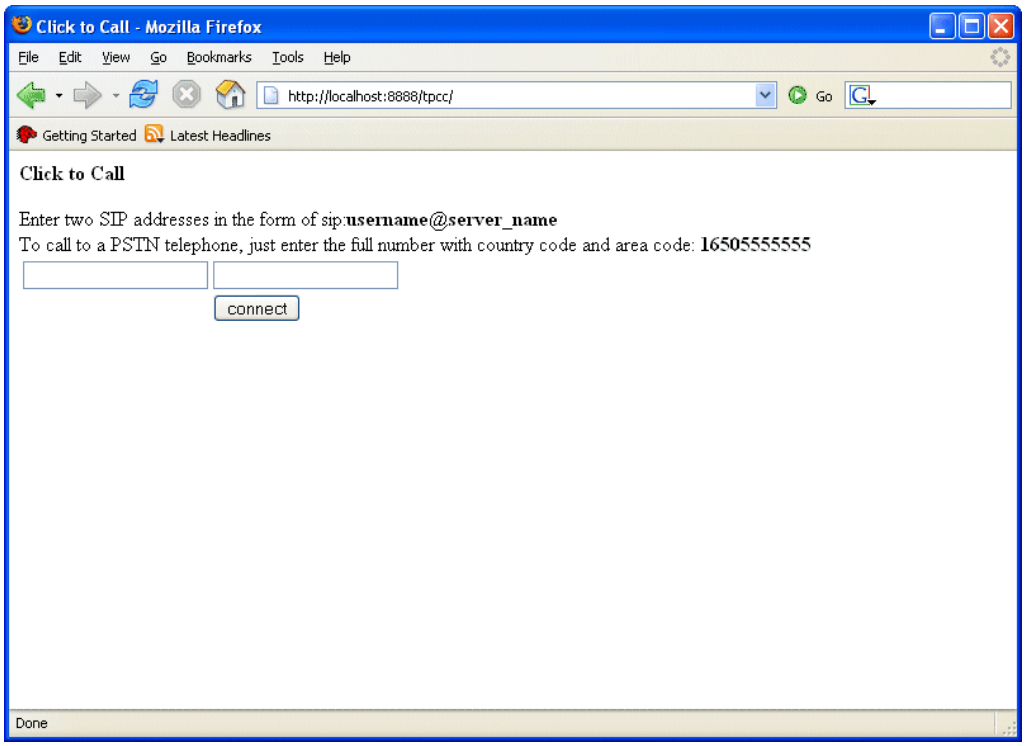
1. Navigate to the following URL `http://localhost:8888/tpcc` (Figure 5–4).
2. Enter two SIP addresses.

   If you have a PSTN Gateway, you can call a PSTN telephone, by entering the full number with the PSTN gateway address. For example, sip:16505555555@<ip-address-of-gateway>

3. Click **Connect**.

   The call is connected. Invitation 1 (the invitation to the first address/number) is sent first. Invitation 2 is sent after a short delay. Verify that the communication is successfully established.

Figure 5–4 Using Click-to-Call (Third-Party Call Control)



6

OCMS Parlay X Web Services

This chapter describes OCMS support for the Parlay X 2.0 Presence Web services interfaces for developing applications. The Web service functions as a Presence Network Agent which can publish, subscribe, and listen to notifies on behalf of the users of the Web service. This chapter consists of the following sections:

Introduction

OCMS provides support for Part 14 of the Parlay X Presence Web Service as defined in the *Open Service Access, Parlay X Web Services, Part 14, Presence ETSI ES 202 391-14* specification. The OCMS Parlay X Web service maps the Parlay X Web service to a SIP/IMS network according to the Open Service Access, Mapping of Parlay X Web Services to Parlay/OSA APIs, Part 14, Presence Mapping, Subpart 2, Mapping to SIP/IMS Networks, ETSI TR 102 397-14-2 specification.

> **Note:** Due to the synchronous nature of the Web service, to receive a callback from the Web service the client must implement the Web service callback interface. For presence, the required interface is the `PresenceNotification` interface described in *Open Service Access, Parlay X Web Services, Part 14, Presence ETSI ES 202 391-14*.
>
> Because this implementation is not common or practical for most Web service architectures, the PresenceNotification interface is not currently supported.

The presence Web service communicates directly with IMS presence network elements using the SIP/SIMPLE protocol interface, and uses the JSR-32 UAC framework to communicate with the SIP network.

The HTTP server that hosts the presence Web service is a Presence Network Agent or a Parlay X to SIP gateway.

Installing the Web Services

The Web services are packaged as a standard .ear file and can be deployed the same as any other Web services through Enterprise Manager. The .ear file contains two .war files that implement the two interfaces. If the Web services are deployed on the same server as the presence server, they must be a child application of the Presence server.

Installing the Aggregation Proxy

The aggregation proxy is packaged as a standard .ear file and can be deployed through Enterprise Manager. There are two requirements:

- The application name must be "aggregationproxyear".
- The parent application must be "subscriberdataservices". The aggregation proxy uses the authentication functions of subscriberdataservices to authenticate Web services.

Configuring Web Services with the Aggregation Proxy

The Web services operate within a trusted domain where another entity performs authentication. To authenticate Web services and identify the user of the services, OCMS uses the aggregation proxy to insert the `HTTP X-3GPP-ASSERTED-IDENTITY` header as defined in *3GPP TS 33.222 Generic Authentication Architecture (GAA); Access to network application functions using Hypertext Transfer Protocol over Transport Layer Security (HTTPS)*.

To define a Web services deployment server:

1. Open the Enterprise Manager page for the aggregation proxy.
2. Configure the *WebServiceHost* and *WebServicePort* to the host and port of the Web services deployment server.
3. Configure the *XCAPHost*, *XCAPPort*, and *XCAPRoot* to the location and parameters of the XDMS.

Presence Web Services Interface Descriptions

The presence Web services consist of three interfaces:

- PresenceConsumer: The watchers use these methods to obtain presence data (Table 6–1).
- PresenceNotification: The presence consumer interface uses the client callback defined in this interface to send notifications. OCMS does not currently support PresenceNotification (Table 6–2).
- PresenceSupplier: The presentity uses these methods to supply presence data and manage access to the data by its watchers (Table 6–3).

Table 6–1 PresenceConsumer Interface

Operation	Description
subscribePresence	The Web Services send a SUBSCRIBE to the presence server.
getUserPresence	Returns the cached presence status because the status changes of the presentity are asynchronously sent to the Web services through a SIP NOTIFY. The Web services actually have the subscription, not the Web services client.
startPresenceNotification	Not supported.
endPresenceNotification	Not supported.

Table 6–2 PresenceNotification Interface

Operation	Description
statusChanged	Not supported.
statusEnd	Not supported.
notifySubscription	Not supported.
subscriptionEnded	Not supported.

Table 6–3 PresenceSupplier Interface

Operation	Description
publish	Maps directly to a SIP PUBLISH.
getOpenSubscriptions	Called by the presentity (supplier) to check if any watcher wants to subscribe to its presence data. No SIP message maps to this method. Returns pending subscriptions currently in the Web services server.
updateSubscriptionAuthorization	The supplier uses this method to answer any open pending subscriptions. An XCAP PUT message is sent to the XDMS server to update the presence-rule document.
getMyWatchers	Retrieves the local list of watchers from the Web services server.
getSubscribedAttributes	Retrieves the local list of subscribed attributes from the Web services server. Currently, only returns *Activity*.
blockSubscription	Causes the Web services server to end a watcher subscription by modifying the XCAP document on the XDMS server (i.e., putting the watcher on the block list).

Using the Presence Web Services Interfaces

This section describes how to use each of the operations in the interfaces, and includes code examples.

Interface: PresenceConsumer, Operation: subscribePresence

This is the first operation the application must call before using another operation in this interface. It serves two purposes:

- It allows the Web services to associate the current HTTP session with a user.
- It provides a context for all the other operations in this interface by subscribing to at least one presentity (SUBSCRIBE presence event).

Code Example

```
// Setting the attribute to activity
PresenceAttributeType pa = PresenceAttributeType.Activity;
PresenceAttributeType[] pat = new PresenceAttributeType[]{pa};

// These inputs are required but not used.
SimpleReference sr = new SimpleReference();
sr.setCorrelator("unused_correlator");
sr.setInterfaceName("unused_interfacename");
sr.setEndpoint(new URI ("http://unused.com"));

// Calling the web service
consumer.subscribePresence (new URI
("sip.presentity@test.example.com") , pat, "unused", sr);
```

Interface: PresenceConsumer, Operation: getUserPresence

Call this operation to retrieve a subscribed presentity presence. If the person is offline, it returns `ActivityNone` and the hardstate note will be written to PresenceAttribute.note. If it returns `ActivityOther`, it returns the description in the form of `OtherValue`. It also returns the note on a device as an `OtherValue` with name = "DeviceNote".

Code Example

```
PresenceAttributeType pa = PresenceAttributeType.Activity;
PresenceAttributeType[] pat = new PresenceAttributeType[]{pa};

//Calling the Web service
URI Presentity = new URI("sip.presentity@test.example.com");
PresenceAttribute[] resultPA =
      consumer.getUserPresence(presentity,pat);

//Getting the result
for (int i = 0; i < resultPA.length; i++){
      PresenceAttribute attribute = resultPA[i];
      String note = attribute.getNote();
      if (attribute.getTypeAndValue().getUnionElement() ==
            PresenceAttributeType.Activity){
            String activity =
                attribute.getTypeAndValue().getActivity()
                .toString();
```

```
 }
    if (attribute.getTypeAndValue().getUnionElement() ==
          PresenceAttributeType.Other &&
          attribute.getTypeAndValue().getOther().getName() ==
         "DeviceNote"){
          String deviceNote =
            attribute.getTypeAndValue().getOther().getValue();
   }
}
```

Interface PresenceSupplier, Operation: publish and Oracle Specific "Unpublish"

This is the first operation the application must call before using another operation in this interface. It serves three purposes:

- It allows the Web services to associate the current HTTP session with a user.
- It publishes the user's presence status.
- It subscribes to watcher-info so that the Web services can keep track of any watcher requests.

Code Example

```
//publish
String note = "From Web Service Client";
PresenceAttributeType activity = ActivityValue.ActivityNone;
PresenceAttribute pa = new PresenceAttribute();
AttributeTypeAndValue typeValue = new AttributeTypeAndValue();
typeValue.setActivity(activity);
typeValue.setUnionElement(PresenceAttributeType.Activity);

//Fill up communication means to some default value
CommunicationMeans mean = new CommunicationMeans();
mean.setContact(new URI("sip:not.used@test.example.com"));
mean.setPriority(1);
mean.setType(CommunicationMeanType.Chat);
CommunicationValue commValue = new CommunicationValue();
commValue.setMeans(new CommunicationMeans[]{mean});
typeValue.setCommunication(commValue);

//Unpublish Functionality Implemented by OCMS
//To perform an "Unpublish", set OtherValue to (Expires, 0)
//OtherValue other = new OtherValue();
//other.setName("Expires");
//other.setValue(0);
//typeValue.setOther(other);
//typeValue.setUnionElement(PresenceAttributeType.Other);

//Set default values for other parameters
typeValue.setPrivacy(PrivacyValue.PrivacyNone);
typeValue.setPlace(PlaceValue.PlaceNone);
typeValue.setSphere(SphereValue.SphereNone);
pa.setTypeAndValue(typeValue);

//Set the note and time
pa.setNote(note);
```

```
Calendar dateTime = Calendar.getInstance();
pa.setLastChange(dateTime);

//Calling the Web service
publish(new PresenceAttribute[]{pa});
```

Interface: PresenceSupplier, Operation: getOpenSubscriptions

This operation retrieves a list of new requests to be on your watcher list.

Code Example

```
SubscriptionRequest[] srArray = getOpenSubscriptions();
for (SubscriptionRequest sr:srArray) {
    System.out.println(sr.getWatcher() .toString());
}
```

Interface: PresenceSupplier, Operation: updateSubscriptionAuthorization

This operation allows you to place a watcher on either the block or allow list.

Code Example

```
PresencePermission pp = new PresencePermission();
pp.setDecision(true);    //Put the user on the allow list

//You always pass in Activity
pp.set.PresenceAttribute(PresenceAttributeType.Activity);
updateSubscriptionAuthorization(new
URI("sip:allow@test.example.com"),
new PresencePermission[]{pp});
```

Interface: PresenceSupplier, Operation: getMyWatchers

This operation retrievesthe list of watchers in your allow list.

Code Example

```
URI[] uris;
uris = getMyWatchers();
for (URI uri:uris)
    System.out.println(uri.toString());
```

Interface: PresenceSupplier, Operation: getSubscribedAttributes

This operation returns only a single item of PresenceTypeAttribute.Activity. An exception will be thrown if there is no existing subscription.

Code Example

```
PresenceAttributeType[] pat =
getSubscriberdAttributes("sip:watcher@test.example.com");
```

Interface: PresenceSupplier, Operation: blockSubscription

This operation places a watcher into the block list.

Code Example

```
blockSubscription(new URI("sip:block.this.watcher@test.example.com"));
```

OCMS Parlay X Presence Custom Error Codes

OCMS introduces two extensions to the Parlay X standard exceptions:

- *PresencePolicyException* extends *PolicyException*, and
- *PresenceServiceException* extends *ServiceException*

Table 6–4 and Table 6–5 describe the error codes and their associated error message.

Table 6–4 OCMS Parlay X Presence Custom Error Codes: PresencePolicyException

Error Code	Error Message
OPOL0001	Watcher is on the block, polite-block or pending list.

Table 6–5 OCMS Parlay X Presence Custom Error Codes: PresenceServiceException

Error Code	Error Message
PRES0001	Invalid result from XDMS server.
PRES0002	Invalid HTTP session data.
PRES0003	Invalid URI.
PRES0004	Peer unavailable.
PRES0005	Unknown host.
PRES0006	Service unavailable.
PRES0007	Internal error.
PRES0008	User unauthenticated.

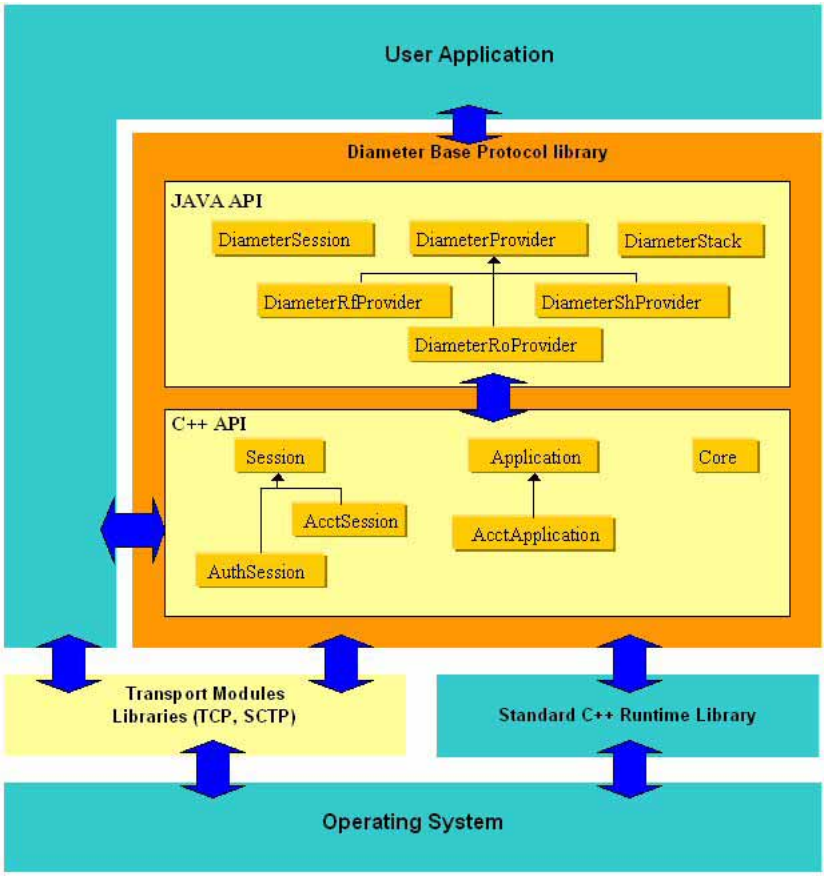
A

Oracle Diameter Java APIs

This appendix describes the Oracle Diameter Java APIs, in the following sections:

- "Diameter Java Base Protocol API"
- "3GPP/Rf Diameter Java API"
- "3GPP/Ro DIAMETER JAVA API"
- "3GPP/Sh Diameter Java API"
- "Diameter Application Example"

Figure A–1 Oracle Diameter Architecture



The Base Protocol library is the main element in the picture. It contains the base protocol implementation, and is available as a standalone C++ shared library.

On its upper side, the Diameter Base Protocol library, provides an application registration mechanism. The specific application subclass AcctApplication can be derived by users depending on the state machine required to be used by the application to be implemented. Each Application object can implement its own session subclass. Specific session subclass (AuthSession and AcctSession) can be derived by users depending on the state machine (Authentication, Authorization or Accounting) implemented by the application.

On its lower side, the Diameter Base Protocol library also provides an API to interact with its transport modules. This API enables the user application to instantiate any subclass of the Transport class. These subclasses can be system dependant, and can provide any kind of transport protocols for Diameter. Diameter comes with a socket based transport module for TCP and SCTP kernel implementations in Unix-like environments.

The Diameter Base Protocol library also provides a direct Core interface. This API supports all the configuration needs, including dynamic realm-based routing table configuration, generic tracing and logging callbacks, external memory manager support, and dynamic dictionary extensibility.

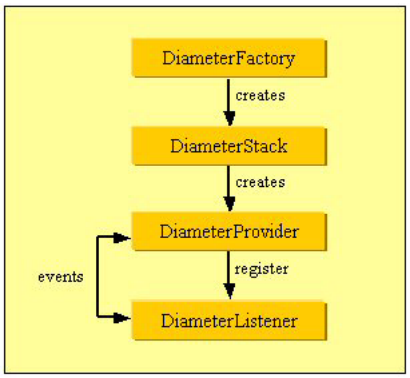
Diameter Java Base Protocol API

The Diameter Java API is an API compliant with JAIN. The JAIN initiative represents a community of communications experts defining the necessary Java interfaces as an extension of the core Java platform to migrate proprietary communications networks to open standardized based networks. The Diameter Java API wraps around the Diameter C++ API.

This section defines the Java API for the Diameter Base Protocol. The Java interfaces and classes described here enable Java developers to build application based on the Diameter base protocol.

This Java API was designed to provide several implementations. The next section will define and describe in detail one possible implementation: Rf Diameter Application.

Figure A–2 Oracle Diameter Java API Architecture



Through Diameter interfaces, a Diameter protocol stack provider is obtained from the factory, and Diameter listeners are then attached to the providers created by the Diameter stack.

Base Protocol Diameter Java Interface

The Diameter Java API is composed of several interfaces:

Diameter Factory

The DiameterFactory interface is used by the Diameter applications to obtain specific instances of different implementations of all interfaces defined in Diameter Java API. The only instance of the DiameterFactory can be obtained using the getInstance() method.

Diameter Stack

The DiameterStack interface represents a Diameter Protocol Stack as defined in [RFC 3588]. This interface is used also to create providers, routes and listening points. A Diameter stack must be initialized and configured before any messages are exchanged.

Diameter Application

The DiameterProvider interface represents a Diameter Application (with one Authorization/Authentication Application ID or one Accounting Application ID) running on top of the Diameter base protocol.

The DiameterListener interface processes events that are triggered by an object implementing the DiameterProvider interface. There can be only one listener per provider. This interface must be implemented by the application programmer.

Diameter Transport

The DiameterRoute interface represents a route entry in the realm-based routing table. A Diameter client application calls the createDiameterRoute method to declare remote peers.

The DiameterListeningPoint interface represents a local transport address for a Diameter stack and enables it to accept incoming connection from declared remote peers.

Diameter Attribute Value Pairs (AVPs)

The DiameterMessageFactory interface provides factory methods that enable a Diameter application to create AVPs and messages for a particular implementation of this specification.

AVPs are defined by the following class types:

- DiameterFloat32AVP: represents a 32-bit float AVP
- DiameterFloat64AVP: represents a 64-bit float AVP
- DiameterInteger32AVP: represents a 32-bit integer AVP
- DiameterInteger64AVP: represents a 64-bit integer AVP
- DiameterGroupedAVP: represents a grouped AVP
- DiameterOctetStringAVP: represents an AVP which value is a raw byte array
- DiameterGenericAVP: represents an AVP which has not been declared in the protocol dictionary

The DiameterMessage interface represents a Diameter Message as defined in [RFC 3588]. The Diameter messages are composed from AVPs added by calling add() method.

Diameter Session

The DiameterSession interface represents a Diameter session as per [RFC 3588]. A Diameter application calls the createClientDiameterSession method from the

DiameterProvider interface if it needs to send a request for a new Diameter session. Then it can send the request by calling sendMessage method. Also, an incoming request can be handled by calling createServerDiameterSession to create a new server session, then using sendMessage to send the response.

Diameter Event

The DiameterEvent interface is the base class for all Diameter events. These events specialize this interface:

- DiameterMessageEvent: This event is generated when an incoming message is available
- DiameterRealmStateChangeEvent: This event notifies the application of the reachability or unreachability of a remote realm, as a result of peers coming up or down
- DiameterSessionEvent: represents all the events which are related to a Diameter session such as received messages, session timeouts, and session errors.

The Diameter stack delivers information to the user application by creating instances of the Diameter Event class and passing these instances to the DiameterListener.processEvent method.

Diameter Exception

The DiameterException interface represents the base class for all the Diameter exceptions. These exceptions specialize this interface:

- DiameterInvalidArgumentException: This exception is thrown or when a method receives an invalid parameter value
- DiameterInvalidMessageException: This exception is thrown when a Diameter Message can't be encoded
- DiameterMessageException: This exception is thrown when a failure occurs during a sending message
- DiameterTransportFailureException: This exception is thrown when a synchronous transport failure occurs during an API action
- DiameterUnsupportedFeatureException: This exception is thrown by an implementation when an operation is invoked that is not supported by this implementation

3GPP/Rf Diameter Java API

This section defines a Java API for the 3GPP [TS 32.299] Rf Interface over the Diameter protocol ([TS 32.225], [TS 32.200] and [TS 32.240] in order to specify).

3GPP/Rf Diameter Java Interface

This section describes the Java interface.

Rf Provider

The DiameterRfProvider interface represents a Rf Diameter Application as defined in 3GPP Interfaces. An instance of a class implementing this interface can be obtained by calling the DiameterStack.createDiameterRfProvider(Properties, refFSM). The different values of refFSM are:

- RFC_SRV_LESS for Rfc 3588 stateless server state machine
- RFC_SRV_FULL for Rfc 3588 stateful server state machine
- RFC_CL for Rfc 3588 client state machine

The Properties parameter specifies the features that will be handled by this Rf provider. This is a list of properties which may contain a number of options.

Rf Listener

The DiameterRfListener interface processes accounting messages that are received by the DiameterRfProviderImpl class implementing the DiameterRfProvider interface. A DiameterRfListener instance is registered to a provider by the following method DiameterRfProvider. setDiameterRfListener ().

This interface provides two principles methods: accountingRequestReceived() and accountingAnswerReceived(). The first one must be implemented and called when the option AUTO_ANSWER is set to false. In this case, the server must be able to create an accounting answer corresponding to the received accounting request. The second one processes the received accounting answer.

Rf Message Factory

The DiameterRfMessageFactory interface provides factory methods that enable an application to create Diameter Rf messages as defined in 3GPP: createAccountingRequest() and createAccountingAnswer(). An instance of a class implementing this interface can be obtained by calling the DiameterStack.getRfDiameterMessageFactory() method.

Messages created by this factory are as following:

- Accounting-Request (ACR)--The first following method creates a basic ACR (with only mandatory AVPs) and the second one creates a more sophisticated ACR with the most used AVPs as defined in 3GPP:
  - DiameterMessage

```
createAccountingRequest (String Destination-Realm,
                         int Accounting-Record-Type,
                         int Accounting-Record-Number)
```

  - DiameterMessage

```
createAccountingRequest(String Destination-Realm,
                         int Accounting-Record-Type,
                         int Accounting-Record-Number,
```

```
                                String roleOfNode,
                                String userSessionId,
                                String callingPartyAddress,
                                String calledPartyAddress)
```

- Accounting-Answer (ACA)--The first following method creates a basic ACA (with only mandatory AVPs) and the second one creates a more sophisticated ACA with the most used AVPs as defined in 3GPP:
  - Diameter Message

```
createAccountingAnswer (String Result-Code,
                        int Accounting-Record-Type,
                        int Accounting-Record-Number)
```

  - Diameter Message

```
createAccountingAnswer (String Result-Code,
                        int Accounting-Record-Type,
                        int Accounting-Record-Number,
                        String roleOfNode,
                        String userSessionId,
                        String callingPartyAddress,
                        String calledPartyAddress)
```

3GPP/Rf Dictionary The 3GPP Rf Interface dictionary is returned by getRfDictionary() and must be extended by the Diameter stack.

Rf Events

Like any Diameter application, an Rf application can receive/send accounting events from/to the Diameter stack.

Process Event To receive an accounting event from the Diameter stack, user has to implement the processEvent(DiameterEvent) callback in the DiameterRfListener implementation.

The list of the possible received accounting events are as follows:

- DiameterTimeoutEvent(Object, DiameterSession, int): This event is generated by the Diameter implementation when one of the session timers elapses. Reasons to trigger an event for the Rf Interface are:
  - SESSION_TIMEOUT_EXPIRED: The event is triggered by the stack on session supervision timer expiration
  - TX_TIMEOUT_EXPIRED: The event is triggered by the stack on session supervision timer expiration.
  - INTERIM_TIMEOUT_EXPIRED: The event is triggered when the time specified by the interim interval duration has expired.
- DiameterServiceEvent(Object, DiameterSession, int): This event is generated by the Diameter implementation when one service is finished. Reasons to trigger an event for the Rf Interface are:
  - TERMINATE_SERVICE: The event is triggered by the stack when service is terminated by the stack or by server.
- DiameterInvalidMessageReceivedEvent(Object, DiameterSession, Message): This event is triggered by the stack when an invalid message has been received. This event object contains the invalid received message.

Raise Events To send an accounting event to the Diameter stack, the user must call the accounting application eventRaised(DiameterEvent) method.The only accounting event triggered by the Rf application is DiameterRfOutOfSpaceEvent(Object, int).

Events raised for an DiameterRfOutOfSpaceEvent are:

- EVT_OUT_OF_SPACE: The event can be triggered by the application to inform the stack that there is no more space to store the received/sent accounting message. This event is created by instantiating the DiameterRfOutOfSpaceEvent class with the type parameter set to EVT_OUT_OF_SPACE. When the Rf server AUTO_ANSWER mode is activated and once the EVT_OUT_OF_SPACE event has been triggered, the stack automatically responds with an answer message including the Result-Code AVP set to DIAMETER_OUT_OF_SPACE and this until the EVT_NOT_OF_OUT_SPACE event triggering.
- EVT_END_OUT_OF_SPACE: The event can be triggered by the application to inform the stack that there is space again to store the received/sent accounting request message. This event is created by instantiating DiameterRfOutOfSpaceEvent class with the type EVT_END_OUT_OF_SPACE. Triggering such an event is relevant only after having triggered an EVT_OUT_OF_SPACE event.

  The DiameterRfOutOfSpaceEvent is relevant only when AUTO_ANSWER mode is activated.

Rf Application Options

These options specify the features handled by the accounting Application. This is a list of Properties (see java.util.properties) which may contain a number of options. An option can be specific to an FSM. Here is the list of the Rf application configurable options which can be set depending on the implemented FSM:

Rf Server Stateless FSM admits the following option settings:

- AUTO_ANSWER
- AUTO_DELETION
- ENABLE_INVALID_COMMAND
- ENABLE_SND_MISSING_AVP

Rf Server StateFul FSM permits the following option settings:

- AUTO_ANSWER
- AUTO_DELETION
- DELETE_ON_TIMEOUT
- SUPERVISION_TIMER_DURATION
- ENABLE_INVALID_COMMAND
- ENABLE_SND_MISSING_AVP

Rf Client FSM permits the following option settings:

- AUTO_DELETION
- ENABLE_INVALID_COMMAND
- ENABLE_SND_MISSING_AVP
- DELETE_ON_TIMEOUT

- SUPERVISION_TIMER_DURATION
- INTERIM_INTERVAL
- REALTIME_REQUIRED

The default value of the INTERIM_INTERVAL and REALTIME_REQUIRED options can be overwritten dynamically by the application for a specific session by using the setSessionParameter() method.

Rf Application FSM

As presented in [RFC 3588], different types of accounting records are sent depending on the actual type of accounted service.

If the accounted service is a one-time event, meaning that the beginning and end of the event are simultaneous, then the Accounting-Record-Type AVP must be set to the value EVENT_RECORD.

If the accounted service is of a measurable length, then the AVP must use the values START_RECORD, STOP_RECORD, and possibly, INTERIM_RECORD.

Here are the two different message flows which can be handled by the Rf Diameter application.

One-time event accounting service:

- Client > ACR (EVENT_RECORD) > Server
- Client < ACA (EVENT_RECORD) < Server

Measurable length duration accounting service:

- Client > ACR (START_RECORD) > Server
- Client < ACA (START_RECORD) < Server
- Client > ACR (INTERIM_RECORD) > Server
- Client < ACA (INTERIM_RECORD) < Server
- Client > ACR (INTERIM_RECORD) > Server
- Client < ACA (INTERIM_RECORD) < Server
- Client > ACR (STOP_RECORD) > Server
- Client < ACA (STOP_RECORD) < Server

ACR(EVENT_RECORD) stands for Accounting-Request message with Accounting-Record-Type AVP set to EVENT_RECORD and request flag set to true.

ACA(EVENT_RECORD) stands for Accounting-Answer message with Accounting-Record-Type AVP set to EVENT_RECORD and request flag set to false.

Following are the types of FSM which can be implemented in an Rf application.

Rf Client FSM This FSM responds to [RFC 3588] requirements which define the client side state machine.

A session instantiated by an application implementing this FSM expects to receive an accounting request message including an Accounting-Record-Type AVP value set to EVENT_RECORD or START_RECORD as first received message.

Once an accounting request message with the Accounting-Record-Type AVP value set to START_RECORD is received, the FSM expects to receive other(s) accounting request message(s) with the Accounting-Record-Type AVP value set to INTERIM_RECORD or

STOP_RECORD. The reception of an accounting request message including an Accounting-Record-Type AVP value set to STOP_RECORD corresponds to the end of the session.

RF Stateful Server FSM This FSM responds to the [RFC 3588] requirements which define the server side state machine that may be followed by applications that require keeping track of the session state at the accounting server.

An application implementing this FSM waits for a request accounting message and sends a response with the same Accounting-Record-Type AVP value as received in the request message.

A session instantiated by an application implementing this FSM expects to receive an accounting request message including an Accounting-Record-Type AVP value set to EVENT_RECORD or START_RECORD as first received message. Once an accounting request message with the Accounting-Record-Type AVP value set to START_RECORD is received, the FSM expects to receive other(s) accounting request message(s) with the Accounting-Record-Type AVP value set to INTERIM_RECORD or STOP_RECORD. The reception of an accounting request message including an Accounting-Record-Type AVP value set to STOP_RECORD represents the end of the session.

Rf Stateless Server FSM This FSM responds to the [RFC 3588] requirements which define the default server side state machine requiring the reception of the accounting records in any order and at any time, and doesn't place any standard requirement on the processing of the records.

An application implementing this FSM waits for a request accounting message and sends a response with the same Accounting-Record-Type AVP value as received in the request message.

3GPP/Ro DIAMETER JAVA API

This section defines a Java API for the 3GPP [TS 32.299] Ro Interface over the Diameter protocol ([TS 32.225], [TS 32.200] and [TS 32.240] in order to specify).

3GPP/Ro DIAMETER JAVA INTERFACE

This section details the 3GPP/Ro Diameter Java Interface

Ro Provider

The DiameterRoProvider interface represents a Ro Diameter Application as defined in 3GPP Interfaces. An instance of a class implementing this interface can be obtained by calling the DiameterStack.createDiameterRoProvider(Properties, refFSM). The different value of refFSM are:

- RFC_CL_LESS for Rfc 4006 event base client state machine
- RFC_CL_FULL for Rfc 4006 session based client state machine
- RFC_SRV for Rfc 4006 server state machine

The Properties parameter specifies the features that will be handled by this Ro provider. This is a list of properties which may contain a number of options.

Ro Listener

The DiameterRoListener interface processes accounting messages that are received by the DiameterRoProviderImpl class implementing the DiameterRoProvider interface. A

DiameterRoListener instance is registered to a provider by the following method: DiameterRoProvider. setDiameterRoListener ().

This interface provides four principle methods: creditControlRequestReceived(), creditControlAnswerReceived(), reAuthRequestReceived() and reAuthAnswerReceived().The first and third one must be implemented and called when the option AUTO_ANSWER is set to false. In this case, the server must be able to create an accounting answer corresponding to the received accounting request. The second and last methods process the received accounting answers.

Ro Message Factory

The DiameterRoMessageFactory interface provides factory methods that enable an application to create Diameter Ro messages as defined in 3GPP: createCreditControlRequest(), createCreditConrolAnswer(), createReAuthRequest() and createReAuthAnswer(). An instance of a class implementing this interface can be obtained by calling the DiameterStack.getRoDiameterMessageFactory() method.

Messages created by this factory are:

- Credit-Control-Request (CCR)

  These methods create a basic CCR (with only mandatory AVPs). The first one creates the CCR without the Requested-Action AVP (mandatory only in event sessions). In the others, the method's name specifies the Requested-Action value:

  - DiameterMessage

    ```
    createCreditControlRequest (String destinationRealm,
        int authApplicationId,
        StringserviceContextId,
        String CCRequestType
        int CCRequestNumber)
    ```

  - DiameterMessage

    ```
    createCreditControlRequestDirectDebiting(String destinationRealm,
        int authApplicationId,
        String serviceContextId,
        int CCRequestNumber)
    ```

  - DiameterMessage

    ```
    createCreditControlRequestRefundAccount(String destinationRealm,
        int authApplicationId,
        String serviceContextId,
        int CCRequestNumber)
    ```

  - DiameterMessage

    ```
    createCreditControlRequestCheckBalance(String destinationRealm,
         int authApplicationId,
         String serviceContextId,
         int CCRequestNumber)
    ```

  - DiameterMessage

    ```
    createCreditControlRequestPriceEnquiry(String destinationRealm,
        int authApplicationId,
        String serviceContextId,
        int CCRequestNumber)
    ```

- Credit-Control-Answer (CCA)

  The first following method creates a basic CCA (with only mandatory AVPs) and the second one creates an CCA from an CCR:

- DiameterMessage

```
createCreditControlAnswer(String resultCode,
    int authApplicationId,
    String CCRequestType,
    int CCRequestNumber)
```

- Diameter Message

```
createCreditControlAnswer(String resultCode,
    DiameterMessage messageCCR)
```

- Re-Auth-Request (RAR)

  This method creates a basic RAR (with only mandatory AVPs)

  - Diameter Message

```
createReAuthRequest(String destinationRealm,
    String destinationHost,
    int authApplicationId,
    String reAuthRequestType)
```

- Re-Auth-Answer (RAA)

  This method creates an RAA, specifying only the Result-Code AVP.

  - Diameter Message

```
createReAuthAnswer(String resultCode)
```

3GPP/Ro Dictionary

The 3GPP Ro Interface dictionary is returned by getRoDictionary() and must be extended by the Diameter stack.

Ro Events

Like any Diameter application, an Ro application can receive/send events from/to the Diameter stack.

Process Events To receive an accounting event from the Diameter stack, user must implement the processEvent(DiameterEvent) callback in the DiameterRoListener implementation.

The list of the possible received accounting events are as follows:

- DiameterTimeoutEvent(Object, DiameterSession, int): This event is generated by the Diameter implementation when one of the session timers elapses. Reasons to trigger an event for the Ro Interface are:
  - SESSION_TIMEOUT_EXPIRED: The event is triggered by the stack on session supervision timer expiration
  - TX_TIMEOUT_EXPIRED: The event is triggered by the stack on session supervision timer expiration.
  - TRA_TIMEOUT_EXPIRED: The event is triggered by the stack when the RAA is not sent on time by the application.
- DiameterServiceEvent(Object, DiameterSession, int): This event is generated by the Diameter implementation when one service is finished. Reasons to trigger an event for the Rf Interface are:

  - TERMINATE_SERVICE: The event is triggered by the stack when the service is terminated by the stack or by the server.
  - GRANT_SERVICE: The event is triggered by the stack when the service is granted by server.
  - BACKUP_SERVICE: The event is triggered by the stack when the Tx2 timer expires and the backup server needs to be contacted by the application.
  - ERROR_SERVICE: The event is triggered by the stack when a service error is detected by the stack.
- DiameterInvalidMessageReceivedEvent(Object, DiameterSession, Message): This event is triggered by the stack when an invalid message has been received. This event object contains the invalid received message.

Raise Events To send an accounting event to the Diameter stack, user has to call the accounting application eventRaised(DiameterEvent) method. The only accounting event triggered by the Rf application is DiameterBackupEvent(Object, int).

The different types of event for an DiameterBackupEvent are:

- EVT_NOBACKUP_ACCEPTED: The event can be triggered by the application to inform the stack that it doesn't have any backup server address.
- EVT_BACKUP_SUCCESS: The event can be triggered by the application to inform the stack that it received a successful credit control answer from the backup server.
- EVT_BACKUP_FAILED: The event can be triggered by the application to inform the stack that it received a failed credit control answer from the backup server.

These events are relevant only when the stack has sent a BackupService event to the application.

Ro Application Options

These options specify the features handled by the accounting Application. This is a list of properties (see java.util.properties) which may contain a number of options. An option can be specific to an FSM. Here is the list of the Ro application configurable options which can be set depending on the implemented FSM:

Ro Server FSM permits the following option settings:

- AUTO_ANSWER
- AUTO_DELETION
- SUPERVISION_TIMER_DURATION
- DELETE_ON_TIMEOUT
- ENABLE_INVALID_COMMAND
- ENABLE_SND_MISSING_AVP

Ro StateFul Client FSM permits the following option settings:

- AUTO_DELETION
- DELETE_ON_TIMEOUT
- TX_TIMER_DURATION
- TRA_TIMER_DURATION
- CCFH

- ENABLE_INVALID_COMMAND
- ENABLE_SND_MISSING_AVP

Ro StateLess Client FSM admits following options setting

- AUTO_DELETION
- DDFH
- TX1_TIMER_DURATION
- TX2_TIMER_DURATION
- BACKUP_TIMER_DURATION
- RETRANSMISSION_INTERVAL
- RETRANSMISSION_ATTEMPS
- ENABLE_INVALID_COMMAND
- ENABLE_SND_MISSING_AVP

Ro Application FSM

As presented in [RFC 4006], different types of credit control requests are sent depending on the actual type of credit control service. If the credit control service is a one-time event, meaning that the start and stop of the event are simultaneous, then the CC-Request-Type AVP must be set to the value EVENT_REQUEST.

If the credit control service is of a measurable length, then the AVP must use the values INITIAL_REQUEST, TERMINATION_REQUEST, and possibly, UPDATE_REQUEST.

Here are the two different message flows which can be handled by the Ro Diameter application.

One-time event credit control service:

- Client > CCR (EVENT_REQUEST) > Server
- Client < CCA (EVENT_REQUEST) < Server

Measurable length duration accounting service:

- Client > CCR (INITIAL_REQUEST) > Server
- Client < CCA (INITIAL_REQUEST) < Server
- Client > CCR (UPDATE_REQUEST) > Server
- Client < CCA (UPDATE_REQUEST) < Server
- Client > CCR (UPDATE_REQUEST) > Server
- Client < CCA (UPDATE_REQUEST) < Server
- Client > CCR (TERMINATION_REQUEST) > Server
- Client < CCA (TERMINATION_REQUEST) < Server

CCR(EVENT_REQUEST) stands for Credit-Control-Request message with CC-Request-Type AVP set to EVENT_REQUEST.

CCA(EVENT_REQUEST) stands for Credit-Control-Answer message with CC-Request-Type AVP set to EVENT_REQUEST.

Following are the types of FSM which can be implemented in a Ro application.

Ro Stateful Client FSM This FSM responds to the [RFC 4006] requirements which define the session-based credit control client state machine when the first interrogation is executed after the authorization and authentication process.

A session instantiated by an application implementing this FSM expects to send a credit control request message including a CC-Request-Type AVP value set to INITIAL_REQUEST as its first sent message. Once a credit control answer message with the CC-Request-Type AVP value set to INITIAL_REQUEST is received, the FSM expects to send either other(s)) credit control request message(s) with the CC-Request-Type AVP value set to UPDATE_REQUEST or UPDATE_REQUEST or a Re-Authentication request message. The reception of a credit control answer message including a CC-Request-Type AVP value set to TERMINATION_REQUEST represents the end of the session.

Ro StateLess Client FSM This FSM responds to the [RFC 4006] requirements which define the event-based credit control client state machine.

An application implementing this FSM sends a credit control request message with the CC-Request-Type AVP set to EVENT_REQUEST and waits for the credit control answer with the CC-Request-Type AVP set to EVENT_REQUEST. The reception of a credit control answer message including a CC-Request-Type AVP value set to EVENT_REQUEST represents the end of the session.

Ro Server FSM This FSM responds to the [RFC 4006] requirements which define the credit control server state machine. An application implementing this FSM waits for a credit control request message and expects to send a response with the same CC-Request-Type AVP value as received in the request message.

A session instantiated by an application implementing this FSM expects to receive an accounting request message including an CC-Request-Type AVP value set to EVENT_REQUEST or INITIAL_REQUEST as its first received message. Once a credit control request message with the CC-Request-Type AVP value set to INITIAL_REQUEST is received, the FSM expects to receive other credit control request messages with the CC-Request-Type AVP value set to UPDATE_REQUESTor TERMINATION_REQUEST. The emission of a credit control answer message including a CC-Request-Type AVP value set to TERMINATION_REQUEST represents the end of the session.

3GPP/Sh Diameter Java API

This section defines a Java API for the 3GPP [TS 29.328] and [TS 29.329] Sh Interface over the Diameter protocol.

3GPP/Sh Diameter Java Interface

This section provides information about the 3GPP/Sh Diameter Java Interface.

Sh Provider

The DiameterShProvider interface represents a Sh Diameter Application as defined in 3GPP Interfaces. An instance of a class implementing this interface can be obtained by calling the DiameterStack.createDiameterShProvider(Properties)

The Properties parameter specifies the features that will be handled by this Sh provider. This is a list of properties which may contain a number of options.

Sh Listener

The DiameterShListener interface processes Diameter messages that are received by the DiameterShProviderImpl class implementing the DiameterShProvider interface. A DiameterShListener instance is registered to a provider by the following method DiameterShProvider. setDiameterShListener ().

This interface provides the following methods:

- profileUpdateRequestReceived()
- profileUpdateAnswerReceived()
- pushNotificationRequestReceived()
- pushNotificationAnswerReceived()
- subscribeNotificationRequestReceived()
- subscribeNotificationAnswerReceived()
- userDataRequestReceived()
- userDataAnswerReceived()

These methods process the received Diameter messages.

Sh Message Factory

The DiameterShMessageFactory interface provides factory methods which enable an application to create Diameter messages as defined by the 3GPP [TS 29.328] specification (Sh interface): createProfileUpdateRequest(), createProfileUpdateAnswer(), createPushNotificationRequest(), createPushNotificationAnswer(), createSubscribeNotificationRequest(), createSubscribeNotification Answer(), createUserDataRequest() and createUserDataAnswer(). An instance of a class implementing this interface can be obtained by calling the DiameterStack.getShDiameterMessageFactory() method.

Messages created by this factory are:

- Profile-Update-Request (PUR)

```
DiameterMessage
createProfileUpdateRequest(String destinationRealm,
  String userIdentity,
  String userData)
```

- Profile-Update-Answer (PUA)

```
DiameterMessage
createProfileUpdateAnswer(String resultCode)
```

- Push-Notification-Request (PNR)

```
DiameterMessage
createPushNotificationRequest(String destinationRealm,
  String destinationHost,
  String userIdentity,
  String userData)
```

- Push-Notification-Answer (PNA)

```
DiameterMessage
  createPushNotificationAnswer(String resultCode)
```

- Subscribe-Notification-Request (SNR)

```
DiameterMessage
  createSubscribeNotificationRequest(String destinationRealm,
  String userIdentity,
  String dataReference,
  String subsReqType)


DiameterMessage
  createSubscribeNotificationRequest(String destinationRealm,
    String userIdentity,
    String dataReference,
    String subsReqType,
    String serviceIndication,
    String serverName)
```

- Subscribe-Notification-Answer (SNA)

```
DiameterMessage
  createSubscribeNotificationAnswer(String resultCode)
```

- User-Data-Request (UDR)

```
DiameterMessage
  createUserDataRequest(String destinationRealm,
    String userIdentity,
    String userIdentityType,
    String dataReference)


DiameterMessage
  createUserDataRequest(String destinationRealm,
    String userIdentity,
    String userIdentityType,
    String dataReference,
    String requestedDomain,
    String currentLocation,
    String serviceIndication,
    String serverName,
    String destinationHost)
```

- User-Data-Answer (UDA)

```
DiameterMessage
  createUserDataAnswer(String resultCode, String userData)
```

3GPP/Sh Dictionary

The 3GPP Sh Interface dictionary is returned by getShDictionary() and must be extended by the Diameter stack.

Sh Events

Like any Diameter application, an Sh application can receive and send events to and from the Diameter stack.

To receive an event from the Diameter application, users must implement the processEvent(DiameterEvent) callback in the DiameterShListener implementation.

Sh Application Options

These options specify the features handled by the accounting Application. This is a list of Properties (see java.util.properties) which may contain a number of options. Following is the list of the Sh application configurable options which can be set depending on the implemented FSM:

Values for the options are:

- AUTO_SESSION_DELETION
- SESSION_DELETED_EVENT
- AUTO_SESSION_TERMINATION_REQUEST
- AUTO_SESSION_ABORT_ANSWERAUTO_SESSION_TERMINATION_ANSWER
- SESSION_TIMEOUT_EVENT
- AUTH_AUTHORIZATION_LIFETIME_EXPIRED_EVENT
- AUTH_GRACE_PERIOD_EXPIRED_EVENT
- LISTENER_SEND_HOOK

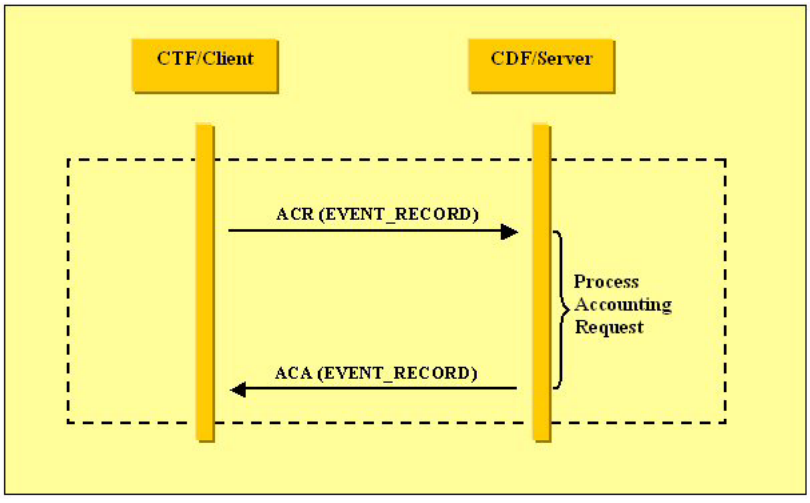
Diameter Application Example

This section presents an example of an accounting call flow and describes in detail the steps that are performed by a typical Accounting Diameter application to represent it. The aim is to show how to use the Rf Java API to exchange accounting information between the Accounting Client and the Accounting Server.

Accounting Call Flow

The following figure shows the transactions that are required on the Diameter offline interface in order to perform event based charging. Following is an example of a call flow between an Accounting Client and an Accounting Server.

Figure A–3 Accounting Call Flow Example



The network element (acting as client) sends an Accounting-Request (ACR) with Accounting-Record-Type AVP set to EVENT_RECORD to indicate service specific information to the CDF (acting as server).

The server processes the received Accounting-Request and returns an Accounting-Answer (ACA) message with Accounting-Record-Type AVP set to EVENT_RECORD to the client.

The following section describes all the steps that must be performed to create an Rf client/server Diameter Application. The steps where the side (client or server) is not specified are common to both sides.

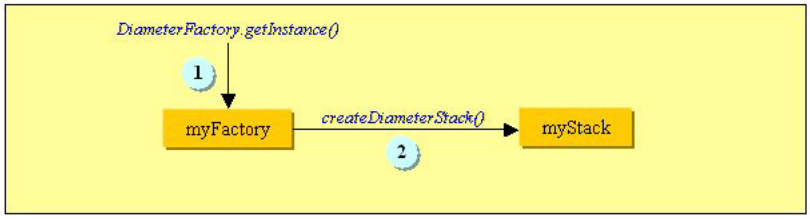
Several steps are required to initialize the Diameter stack before messages can be exchanged.

- Create a DiameterStack instance
- Register the Rf Diameter application to the created Diameter stack
- Create listening points to bind to local transport addresses (optional / server side)
- Configure routes and connect to Diameter peers (optional / client side)

Application initialization

An instance of the Diameter stack can be created as follows:

Figure A–4 Oracle Diameter Stack Creation



1. Get an instance of the DiameterFactory class myFactory using the following method: DiameterFactory.getInstance().
2. Create myStack, an instance of the DiameterStack class, using the following method: myFactory.createDiameterStack().

Rf Diameter Application Once the Diameter stack is defined, the current accounting application objects may be instantiated. An Accounting Diameter application is represented by instances of the following interfaces: DiameterRfProvider and DiameterRfListener. The former represents the Accounting Diameter application for use by the higher-level user code; the latter represents the Accounting Diameter application for use by the Diameter stack. The DiameterRfProvider interface is used by the higher-level user code to send accounting messages (Accounting Request (ACR) and accounting Answer (ACA)) and control application behavior. All the call back methods defined in the DiameterRfListener interface are used by the Diameter stack to deliver these accounting messages and some events to the user code.

The Rf provider is created as follows:

Figure A–5 Rf Provider Creation

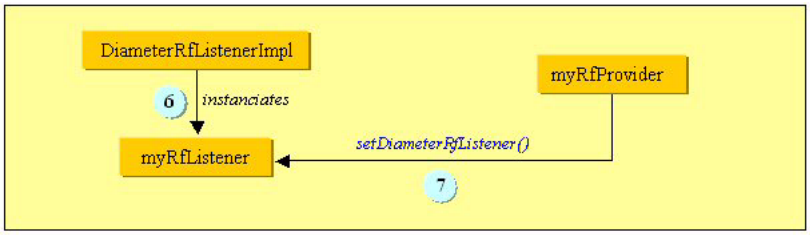
3. Create my Rf Provider, an instance of DiameterRfProvider, using the following method: myStack.createDiameterRfProvider(). The instantiation is done with the application name matching the one configured in the dictionary, and attaching it to a listener so that it receives all incoming messages. Among the parameters passed during the provider creation, properties is a set of accounting options previously defined.

   The 3GPP Rf Interface dictionary is returned by getRfDictionary() and must be extended by the Diameter stack.

   For increased flexibility in a real accounting application, extend the dictionary with application specific AVP and command codes. Use the myStack.extendGrammar(myDictionnary) dictionary.

   The Rf Listener is created and registered as follows:

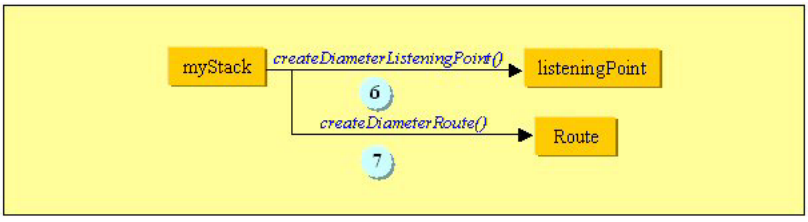
Figure A–6 Rf Listener Creation and Registration



4. DiameterRfListenerImpl is an implementation of DiameterRfListener interface and may be implemented by the application programmer. myRfListener is an instance of DiameterRfListenerImpl.
5. Attach myRfProvider to the created listener myRfListener using the following method: setDiameterRfListener().

Transport configuration The Accounting Application must create one or several instances of the DiameterListeningPoint interface to listen for incoming connections on one or several transport addresses. As soon as the listening point has been created, the Diameter stack is ready to accept incoming connection from remote peers which can be declared by using the createDiameterRoute() method.

Figure A–7 Routes and Listening Points Creation



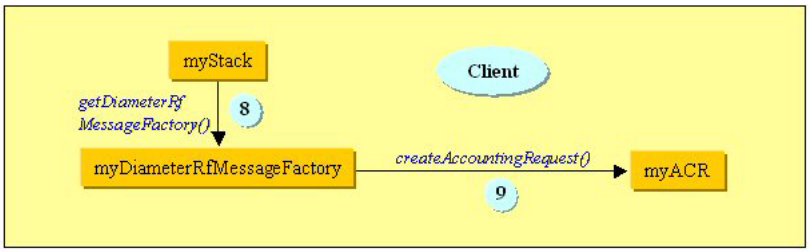
6. Create a listening Point from a LocalURI, an instance of DiameterListeningPoint, using the following method: myStack.createDiameterListeningPoint()
7. Create a route configuration using the following method: myStack.createDiameterRoute(). This is not mandatory for server processes, if the isUnknownPeerAuthorized() callback is implemented in the listener. This callback is called if the Diameter stack receives a connection request from a peer that has not been declared in the routing table. The connection is accepted only if this method returns true.

Accounting Diameter message exchange

Once the initialization and configuration phase is over, the Accounting Diameter application is able to send and receive Diameter accounting messages (ACR/ACA).

Accounting-Request (ACR) The Accounting Request is created as follows:

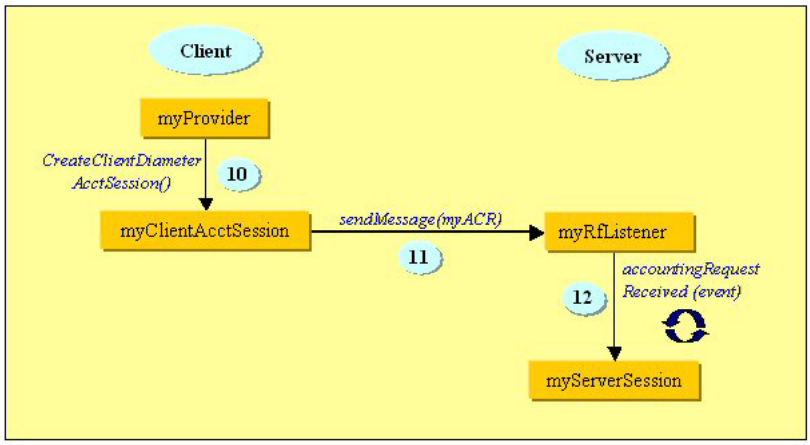
Figure A–8 ACR Creation



8. Get an instance of the DiameterRfMessageFactory class myDiameterRfMessageFactory using the following method: getDiameterRfMessageFactory() for the client side.
9. Create an Accounting Request for the client side, an instance of the DiameterMessage class, using the following method: createAccountingRequest().

   All incoming messages are delivered to the user accounting application through the DiameterRfListener interface. When accounting messages (ACR/ACA) are received by the client or server side, the corresponding callback implemented by the programmer is called. For example, if an accounting Request (ACR) is received, the accountingRequestReceived() callback is called. In all the other cases, the received messages are delivered through the myRfListener.processEvent() implemented by the accounting application programmer.

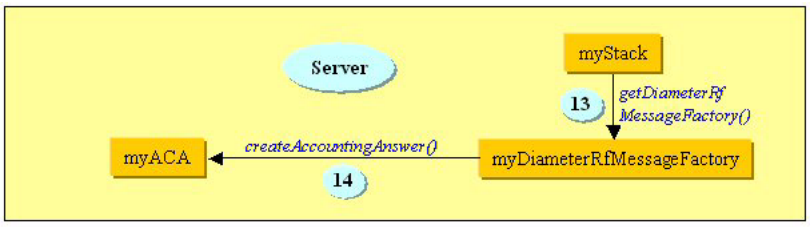
Figure A–9 ACR Reception



10. Create a client diameter session, an instance of DiameterSession, by using the method: createClientDiameterAcctSession().
11. Send the created Accounting Request (ACR) myACR to the server side by using the method: sendMessage().
12. Process an incoming Accounting Request (ACR) message contained in the received event by using the following method: accountingRequestReceived(event). Create a server session from the received request. This session, on server side, can be deleted just after sending the answer.

Accounting-Answer (ACA) The Accounting Answer is created as follows:

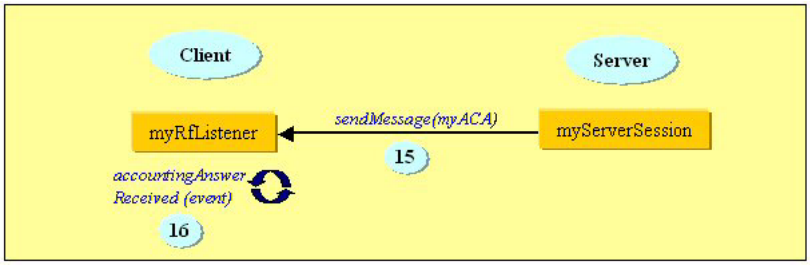
Figure A–10 Creating the Corresponding ACA



13. Get an instance of the DiameterRfMessageFactory class myDiameterRfMessageFactory by using the method: getDiameterRfMessageFactory() on the server side.
14. Create an Accounting Answer for the server side, an instance of the DiameterMessage class, by using the method: createAccountingAnswer().

    All incoming messages are delivered to the user accounting application through the DiameterRfListener interface.

Figure A–11 ACA Reception

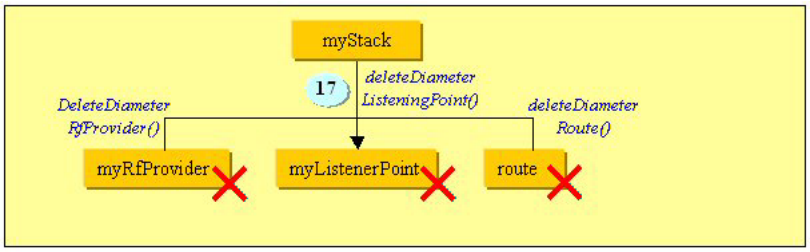


15. Send the created Accounting Answer myACA to the client side.
16. Process the incoming Accounting Answer (ACA) message contained in the received event by using the method: accountingAnswerReceived(event).

Cleanup

In the last step, a global cleanup of all created routes, providers and listening points is done.

Figure A–12 Cleaning Up of Routes, Providers, and Listening Points



17. Cleaning:
    - Delete all the created listening points by using the method: myStack.deleteDiameterListeningPoint(listeningPoint)
    - Delete the created provider by using myStack.deleteDiameterRfProvider(myRfProvider).
    - Delete all the created routes by using myStack.deleteDiameterRoute(route).

B

Programming Oracle Diameter Applications

This appendix describes programming Oracle Diameter applications in the following sections:

- "IP and Routes Configuration"
- "Counters Management"
- "Dictionary"
- "Tracing and Logging Mechanism"

IP and Routes Configuration

Before a Java Diameter application is able to process messages exchanged with a distant peer, the IP configuration and Diameter protocol-specific configuration have to be done by the application as follows:

1. Create a `DiameterStack` instance.
2. Register the Diameter application to the Diameter stack.
3. Create listening points to bind to local transport addresses.
4. Configure routes and connect to Diameter peers.

Creating a Diameter Stack

An instance of the Diameter stack can be created as follows:

```
import oracle.sdp.diameter.*;
DiameterFactory myFactory;
DiameterStack myStack;
myFactory = DiameterFactory.getInstance();
myStack =  myFactory.createDiameterStack("realm.domain.com",
        "server.realm.domain.com",
        null);
```

This code creates a Diameter stack for use by the local Diameter node in which Fully Qualified Domain Name (FQDN) is `server.realm.domain.com` and Origin Realm is `realm.domain.net`.

Binding to Local Transport Addresses

When a Diameter application needs to listen for incoming connections on one or several transport addresses, it has to create one or several instances of the `DiameterListeningPoint` interface:

```
String localURI = "aaa://server.realm.domain.com:41001";
myStack.createDiameterListeningPoint(localURI);
```

As soon as the listening point has been created, the Diameter stack is ready to accept incoming connection from remote peers. If the Diameter stack receives a connection request from a peer that as not been declared in the routing table, then the `isUnknownPeerAuthorized()` of the `DiameterListener` interface is called. The connection is accepted only if this method returns *true*.

Note: There is no need for the user application to keep the references on the listening points since they can be retrieved later by calling `DiameterStack.getDiameterListeningPoints()`.

Configuring Routes and Binding to Diameter Peers

A Diameter client application can declare remote peers by using the `createDiameterRoute()` method.

The code fragment illustrated in Example 6–1 configures two Diameter realms, `realm1.domain.com` and `realm2.domain.com`. The first realm is served by two peers: `peer1.realm1.domain.com` and `peer2.realm1.domain.com`, whereas the second realm is served by only one peer, `peer.realm2.domain.com`. The metric values (1 and 2) are such that `peer1` and `peer2` are set up in master/backup mode. Example 6–1 illustrates this source code for setting up this peer configuration.

Example 6–1 Configuring Peers

```
myStack.createDiameterRoute("ExampleApp", "realm1.domain.com",
                            "aaa://peer1.realm1.domain.com", 1);
myStack.createDiameterRoute("ExampleApp", "realm1.domain.com",
                            "aaa://peer2.realm1.domain.com", 2);
myStack.createDiameterRoute("ExampleApp", "realm2.domain.com",
                            "aaa://peer.realm2.domain.com:41002", 1);
```

Note: If a peer name (FQDN) is used in `createDiameterRoute()` and if that peer is not yet known to the local stack, a transport connection is initiated with the peer using the specified peer URI (taking into account any URI optional information such as port number and transport protocol). On the contrary, if the peer specified by the FQDN part of the URI is already known, the URI is ignored, and the existing peer entry is added to the routing table for the specified realm and application ID.

Realm State Availability

The `DiameterRealmStateChangeEvent` class is used to notify the application of the reachability or unreachability of a remote realm as a result of peers coming up or down. This is important because the Diameter stack will not accept an outgoing message for which the remote realm is not available. Therefore, the application should wait until the realm is available before sending requests.

A `RealmStateChange` event is passed to `DiameterListener.processEvent()` whenever the availability of a pair (Remote-Realm, Application-ID) changes. The availability of a remote realm for a given application ID depends on the availability of active connections to at least one remote peer that is able to serve the specific realm

and application ID. Since the route is not available, the application is not able to exchange messages with the remote realm peers.

Example 6–2 illustrates a typical implementation of the `DiameterListener.processEvent()` method.

Example 6–2 Implementing the DiameterListener.processEvent() Method

```
public void processEvent(DiameterEvent event)
{
  if (event instanceof DiameterRealmStateChangeEvent) {
      // A remote realm has become available or unavailable
      DiameterRealmStateChangeEvent event =
 (DiameterRealmStateChangeEvent)event;
      if (event.isRealmAvailable()) {
        System.out.println("Realm " + event.getRealm() + " is available");
      } else {
        System.out.println("Realm " + event.getRealm() + " is unavailable");
      }
      // ...
  }
```

Counters Management

Upon Diameter stack initialization, a set of defined counters is initialized and associated to each `DiameterStack` and `DiameterProvider` instance created by the application. These counters are defined in the `DiameterStackImplMBean` and `DiameterProviderImplMBean` management interfaces.

There are two ways to access to these counters:

1. Directly, by calling one of the different methods defined in both management interface.
2. Remotely, by registering the Diameter MBeans to a JMX agent using javax.management package. Only the JDK 1.5 provides this package.

MBeans Management Interface

There are two management interfaces defined in the `oracle.sdp.diameterimpl` package:

1. `DiameterStackImplMBean`: This interface represents the management API for an instance of the `DiameterStack` interface.
2. `DiameterProviderImplMBean`: This interface represents the management API for an instance of the `DiameterProvider` interface

Example 6–3 illustrates how to directly get the value of one of these defined counters:

Example 6–3 Getting the Value of a Counter

```
//--> Example Stack: getting the NbTransactions counter
MyStack->getNbTransactions();
//--> Example Provider: getting the NbSessionsCreated counter
MyProvider->getNbSessionsCreated();
```

Managing a Diameter Application with MBeans

A Diameter Application can be managed remotely by registering the Diameter MBeans to a JMX agent and can be monitored by using the Java JConsole program.

Registering the Diameter MBeans

A Java application using the Diameter API can publish management information by registering instances of the `DiameterStackImplMBean` and `DiameterProviderImplMBean` interfaced to a JMX agent. This can be done as follows:

```
import javax.management.MBeanServerFactory;
import javax.management.MBeanServer;
import javax.management.ObjectName;

import oracle.sdp.diameter.*;

// Create DiameterStack and DiameterProvider instances.
// DiameterStack myStack = ...
// DiameterProvider myProvider = ...
List srvList = MBeanServerFactory.findMBeanServer(null);

if (srvList.isEmpty() == false) {
 MBeanServer server = (MBeanServer)srvList.iterator().next();
 try {
   ObjectName name;

   name = new ObjectName( "oracle.sdp.diameterimpl:name=DiameterProvider");
   server.registerMBean(myProvider, name);

   name = new ObjectName("oracle.sdp.diameterimpl:name=DiameterStack");
   server.registerMBean(myStack, name);
 }
 catch (Exception e) {
   // Handle register exception
   // ...
 }
}
```

Note: This code requires the JDK 1.5 or later. Previous versions do not have the required `javax.management` package.

If you intend to allow remote access to the MBeans, you must define the Java property `com.sun.management.jmxremote` when running your application, as follows:

```
java -Dcom.sun.management.jmxremote -classpath mdiameter.jar
MyApplication
```

Using jconsole to Monitor Diameter Applications

When you have a Diameter application running -- and provided you have registered the MBeans as described above -- you can use the JDK's jconsole application to browse the management characteristics of the stack and application.

Start jconsole as follows:

```
jconsole
```

And then select the application's JVM from the local list. The Diameter MBeans should be visible under the *MBeans* tab.

Dictionary

This section includes the following topics:

Dictionary Composition

When a user's application requires using commands or AVPs that are not defined in the default loaded application dictionary, the user can extend the dictionary to define new commands and/or AVPs syntaxes to be used by the Diameter stack.

dictionary Element

The root or top-level element of a Diameter dictionary extension is the `<dictionary>` element:

```
<dictionary>
.... (other elements)
</dictionary>
```

The `<dictionary>` element contains zero or more `<vendor>` elements and zero or more `<application>` elements.

vendor Element

The `<vendor>` element defines a vendor by a name and associated IANA.

The `<vendor>` attributes are:

1. The vendor `id` attribute must be unique across all `<vendor>` element definitions of the dictionary. The value 0 is dedicated to the base protocol which corresponds to the syntaxes defined in [RFC-3588] and [RFC-4006].
2. The vendor `name` attribute is some text describing the vendor.

In Example 6–4, the `<vendor>` element defines the vendor named "3GPP" whose enterprise code is 10415:

Example 6–4 Defining a Vendor

```
<dictionary>
      <vendor id="10415" name="3GPP">
      ....(other elements)
      </vendor>
</dictionary>
```

The `<vendor>` element contains zero or more `<returnCode>` elements and zero or more `<avp>` elements.

application Element

One of the ways in which the Diameter protocol can be extended is through the addition of new applications.

The `<application>` element defines the new commands needed to support a new vendor Diameter application.

The <application> attributes are:

1. The application id attribute is the IANA-assigned Application Identifier for this application. The value 0 is dedicated to the base protocol which corresponds to the commands defined in RFC-3588 and RFC-4006.
2. The application name attribute is the human-readable name of this application.
3. The application vendor attribute is the name of the application vendor as previously defined in the <vendor> element.
4. The application service-type attribute defined the type of service delivered by the application. Possible values are "Acct" for accounting and "Auth" for authorization.

In Example 6–5, the <application> element contains information for the 3GPP accounting "Rf" application identified by the value "3":

Example 6–5 Defining an <application> Element

```
<dictionary>
      <application id="3" name="Rf" vendor="3GPP" service-type="Acct">
      ....(other elements)
      </application>
</dictionary>
```

The <application> element contains zero or more <command> elements.

command Element

A <command> element defines the attributes for a command.

The <command> attributes are:

1. The command name attribute defines the name of the command. Because only one command is defined for both "Request" and "Answer" portions, the "Accounting" command defines both "Accounting-Request" and "Accounting-Answer" messages.
2. The command code attribute defines the command code used to transmit this command.

In Example 6–6, the Rf application contains the command "Accounting" whose code is 271:

Example 6–6 Defining the <command> Element

```
<dictionary>
      <application id="3" name="Rf" vendor="3GPP" service-type="Acct">
         <command name="Accounting" code="271" />
         ....
      </application>
</dictionary>
```

returnCode Element

The <returnCode> element defines a possible value of the Result-Code AVP. In Example 6–7, the 3GPP vendor defines the returnCode 5030 named DIAMETER_USER_UNKNOWN.

Example 6–7 Defining the <returnCode> Element

```
<dictionary>
      <vendor id="10415" name="3GPP">
         <returnCode name="DIAMETER_USER_UNKNOWN" code="5030" />
         ....
      </vendor>
</dictionary>
```

avp Element

The <avp> element defines an AVP as described in RFC-3588.

The <avp> attributes are:

1. The avp name attribute is the human-readable name of this AVP.
2. The avp code attribute defines the integer value used to encode the AVP for transmission on the network.
3. The avp mandatory attribute defines whether the mandatory bit of this AVP should or should not be set. Possible values are "must" or "mustnot".
4. The avp protected attribute defines whether the protected bit of this AVP should or should not be set. Possible values are "may" or "maynot".
5. The avp may-encrypt attribute defines whether the AVP has to be encrypted in case of CMS security usage. Possible values are "yes" or "no".
6. The avp vendor-specific attribute specifies if this is a vendor specific AVP or not. Possible values are "yes" or "no".

In Example 6–8, the 3GPP vendor extends the dictionary with the AVP "Application-provided-called-party-address".

Example 6–8 Defining the <avp> Element

```
<dictionary>
      <vendor id="10415" name="3GPP">
         <avp name="Application-provided-called-party-address"
            code="837"
            mandatory="mustnot"
            protected="may"
            may-encrypt="no"
            vendor-specific="yes">
            ....
         </avp>
      </vendor>
</dictionary>
```

The <avp> element regroups either a <type> element or a <grouped> element.

type Element

The <type> element defines the data type of the AVP in which it appears. This element must appear in all non-grouped AVP definitions.

The type-name attribute of the <type> element contains the data type name as defined in RFC-3588: Possible values are:

- "OCTETSTRING"
- "INTEGER32"
- "INTEGER64"

- "UNSIGNED32"
- "UNSIGNED64"
- "FLOAT32"
- "FLOAT64"
- "ADDRESS"
- "IPADDRESS"
- "TIME"
- "UTF8STRING"
- "DIAMETERIDENTITY"
- "DIAMETERURI"
- "IPFILTERRULE"
- "QOSFILTERRULE"
- "ENUMERATED"
- "GROUPED"

Note: These values are case-sensitive.

In Example 6–9, the AVP "Application-provided-called-party-address" is an UTF8String.

Example 6–9 Defining the <type> Element

```
<dictionary>
   <vendor id="10415" name="3GPP">
      <avp name="Application-provided-called-party-address"
         code="837"
         mandatory="mustnot"
         protected="may"
         may-encrypt="no"
         vendor-specific="yes">
            <type type-name="UTF8String"/>
      </avp>
   </vendor>
</dictionary>
```

enum Element

The `<enum>` element defines a name which is mapped to an Unsigned32 value used in encoding and decoding AVPs of type Unsigned32. Enumerated elements should only be used with Unsigned32 typed AVPs.

The `<enum>` element's attributes are:

1. The enum `name` attribute is the text corresponding to a particular value for the attribute.
2. The enum `code` attribute is the Unsigned32 value corresponding to this enumerated value

In Example 6–10, the Accounting-Record-Type AVP has four values: EVENT_RECORD, START_RECORD, INTERIM_RECORD and STOP_RECORD.

Example 6–10 Defining the <enum> Element

```
<dictionary>
      <vendor id="10415" name="3GPP">
         <avp name="Accounting-Record-Type"
            code="480"
            mandatory="must"
            protected="may"
            may-encrypt="yes">
               <type type-name="Unsigned32"/>
               <enum name="EVENT_RECORD" code="1"/>
               <enum name="START_RECORD" code="2"/>
               <enum name="INTERIM_RECORD" code="3"/>
               <enum name="STOP_RECORD" code="4"/>
         </avp>
      </vendor>
</dictionary>
```

grouped Element

The `<grouped>` element defines an AVP which encapsulates a sequence of AVPs together as a single payload. It consists in grouping one or more `<gavp>` elements. This way, a single "grouped" element can contain references to multiple AVPs. Each `<gavp>` element holds an AVP `name` and a `vendor-id` attribute.

The <gavp> attributes are:

1. The gavp `name` attribute must correspond to some existing AVP's `name` attribute.
2. The gavp `vendor-id` attribute refers to an existing vendor's `id` attribute.

   In Example 6–11, the 3GPP vendor defines an AVP named "CC-Money" which is a set of previously defined AVPs named "Unit-Value" and "Currency-Code".

Example 6–11 Defining the <grouped> and <gavp> Elements

```
<dictionary>
      <vendor id="10415" name="3GPP">
         <avp name="CC-Money"
            code="413"
            mandatory="must"
            protected="may"
            may-encrypt="yes">
               <grouped>
                  <gavp name="Unit-Value" />
                  <gavp name="Currency-Code" />
                  ....
               </grouped>
         </avp>
      </vendor>
</dictionary
```

Dictionary Extension

Once the Diameter dictionary extension has been defined, use the `extendGrammar()` method to apply the extension to the default dictionary as follows:.

```
//--> Define dictionary extension string
String myDictionary =
   "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n"             +
   "<!DOCTYPE dictionary SYSTEM \"dictionary.dtd\">\n"        +
   "<dictionary>\n"                                           +
```

```
   " <vendor id=\"10415\" name=\"3GPP\">\n"                    +
   " </vendor>\n"                                              +
   " <application id=\"3\" name=\"Rf\" vendor=\"3GPP\" \n"     +
   " service-type=\"Acct\">\n"                                 +
   " <command name=\"Accounting\" code=\"271\" />\n"           +
   " </application>\n"                                         +
   "</dictionary>\n";

//--> Apply new extension to current dictionary
try {
  myStack . extendGrammar(myDictionary);
} catch (DiameterException e) {
  // Handle dictionary syntax errors ...
}
```

For increased flexibility in a real application, you may want to read the XML syntax description from a file rather than having it embedded in the Java source code. This way, it becomes possible to change the mapping between names and codes without recompiling the application.

The 3GPP `Rf` Interface dictionary is returned by `getRfDictionary()` and the 3GPP `Ro` Interface dictionary is returned by `getRoDictionary()` and can be extended by the Diameter stack.

Tracing and Logging Mechanism

The `DiameterTraceLoggerListener` class is an interface to the Diameter tracing and logging mechanism. This interface represents the communication channel implemented by an application to receive debug traces and logs from the Diameter stack implementation. Logs are messages targeted to the user of `DiameterStack`. Traces are for internal use and are meaningful only to people with a good knowledge of the `DiameterStack` implementation.

All the messages that may be sent through the `DiameterTraceLoggerListener.log()` logging interface are defined in `LogMessages.def`. There is no definition file for trace messages.

By default, logs are sent to stdout and traces are not sent. This behavior may be modified by users either by registering a user-defined subclass of `DiameterTraceLoggerListener` or by defining specific environment variables. An example of a `DiameterTraceLoggerListener` implementation is as follows:

```
Class MyTraceLoggerListener implements DiameterTraceLoggerListener
{
      // true or false.
      public boolean isTracingEnabled()
      {
         return true;
}

public void log (String file, int line, int severity, String message)
{
         String severity;
         switch (severity) {
            case LOG_INFO_SEVERITY: severity="INFO"; break;
            case LOG_WARNING_SEVERITY: severity="WARNING"; break;
            case LOG_ERROR_SEVERITY: severity="ERROR"; break;
            case LOG_DISASTER_SEVERITY: severity="DISASTER"; break;
            default: severity="?"; break;
         }
```

```
            // ...
        }

        public void trace (String file, int line, int mask, String message)
        {
            System.err.println(...);
    }
    }
```

C

Accounting Event API

This appendix describes the OCMS Accounting Event API, in the following sections:

Introduction

OCMS provides the Accounting Event API, an interface to create SIP event data that can be used for accounting and charging. The event output data can be post-processed to create CDRs.

The Accounting Event API consists of the following two interfaces:

- `oracle.sdp.sipservletframework.eventlogger.EventLoggerFactory`
- `oracle.sdp.sipservletframework.eventlogger.EventLogger`

The container creates an `EventLoggerFactory` instance available through a `ServletContext` attribute with the name `oracle.sdp.sipservlet.EventLoggerFactory.EVENT_LOGGER_FACTORY`. The string cannot be used by itself, but must be specified as follows:

```
EventLoggerFactory elf =
getServletContext().getAttribute(oracle.sdp.sipservletframework.eventlogger.
EventLoggerFactory.EVENT_LOGGER_FACTORY
```

The `EventLogger` interface is implemented by the loggers created by the `EventLoggerFactory.getLogger(String loggerName)` method. The `EventLogger` instances use Apache Log4j internally for the actual logging. Log4j allows flexible logging for instance to a file or to a centralized log server using syslog.

The `EventLogger` interface has three methods for logging events (illustrated in Example C–1):

- `logEvent(SipServletRequest req, Map<Object, Object> additional)`
- `logEvent (SipServletResponse resp, Map<Object, Object> additional`
- `logEvent(Map<Object, Object> event, String category)`

Example C–1 Logging Events in the EventLogger Interface

```
public interface EventLogger
{
  public void logEvent(SipServletRequest req, Map<Object, Object> additional);

  public void logEvent(SipServletResponse resp, Map<Object, Object> additional);

  public void logEvent(Map<Object, Object> event, String category);

  public boolean isEnabled(SipServletRequest req);
  public boolean isEnabled(SipServletResponse resp);
  public boolean isEnabled(String category);
}
```

logEvent(SipServletRequest req, Map<Object, Object> additional) Method

The `logEvent(SipServletResponse resp, Map<Object, Object> additional)` method logs an event with the characteristic request attributes listed together with any additional event attributes provided. Table C–1 describes the request attributes of the logEvent Method. The event is logged to the following Log4j category name:

```
"eventlogger." + loggerName + ".REQUEST." + method.
```

Table C–1 Request Attributes of the logEvent(SipServletRequest req, Map<Object, Object> additional) Method

Attribute	Description
RequestUri	The *Request-URI* value.
Method	The *method* value.
To	The *To* header value.
From	The *From* header value.
Call-ID	The *Call-ID* header value.
CSeq	The *CSeq* header value.
Via	A string of all *Via* header values, separated by semicolons (;).
Content-Type	The *Content-Type* header value, or "".
Content-Length	The *Content-Length* header value, or "".
MEDIA	A string of all SDP media line values (that is, the portion following *m*=), which are separated by semicolons(;). Only include this attribute for INVITE requests when SDP content is not present.
Service	The name of the service generating the event (that is, the full Log4j category name).

logEvent(SipServletResponse resp, Map<Object, Object> additional) Method

The `public void logEvent(SipServletResponse resp, Map<Object, Object> additional)` method logs an event with the characteristic response attributes listed in Table C–2 with any additional event attributes provided. The event will be logged to the following full Log4j category name:

```
"eventlogger." + loggerName + ".RESPONSE." + method + "." + statuscode.
```

Table C–2 describes the attributes for `public void logEvent(SipServletResponse resp, Map<Object, Object> additional)`.

Table C–2 Response Attributes for the logEvent(SipServletResponse resp, Map<Object, Object> additional) Method

Attribute	Description
StatusCode	The response status code.
Method	The method value.
To	The *To* header value.
From	The *From* header value.
Call-ID	The *Call-ID* header value.
CSeq	The *CSeq* header value.
Via	A string of all *Via* header values, separated by semicolons (;).
Content-Type	The *Content-Type* header value, or "".
Content-Length	The *Content-Length* header value, or "".
MEDIA	A string of all SDP media line values (that is, the portion following *m*=), which are separated by semicolons(;). Only include this attribute for INVITE requests when SDP content is not present.
Service	The name of the service generating the event (that is, the full Log4j category name).

logEvent(Map <Object, Object> event, String category) Method

The `public void logEvent(Map <Object, Object> event, String category)` method is a generic method that can be used to log any kind of event in the form of an attribute map. The `Service` attribute listed in Table C–3 is also added to the attribute map. The event will be logged to the full Log4j category name, which can take either of the following two forms:

- "eventlogger." + loggerName
- "eventlogger." + loggerName + "." + category if a non-null category is provided.

Table C–3 lists the attributes for this method.

Table C–3 Attributes of the logEvent(Map <Object, Object> event, String category) Method

Attribute Key	Description of attribute value
Service	The name of the service generating the event (that is, the full Log4j category name).

The `isEnabled` method returns `true` only if the Log4j filter is configured so that a call to `logEvent` method with the same argument is logged. Example C–2 illustrates how a servlet uses the Accounting Event API to log events for all of its requests and responses.

Example C–2 Logging Events with the Accounting API

```
public class EventTestServlet extends SipServlet
{

  private EventLogger eventLogger;

  public void init() {

     ServletContext sc = getServletContext();
     EventLoggerFactory factory = (EventLoggerFactory) sc
        .getAttribute("oracle.sdp.sipservlet.EventLoggerFactory");
     eventLogger = factory.getLogger(EventTestServlet.class);
   }
   public void doRequest(SipServletRequest req) {
      if (eventLogger.isEnabled(req)) {
      eventLogger.logEvent(req, null);
   }
      URI requestURI = req.getRequestURI();
      req.getProxy().proxyTo(requestURI);
   }
   public void doResponse(SipServletResponse resp) {
      if (eventLogger.isEnabled(resp)) {
         Map additional = new HashMap();
         additional.put("foo", "bar")
         eventLogger.logEvent(resp, additional);
      }
   }
}
```

Event Processing in Log4j

Log4j is executes the following tasks:

- Event filtering
- Event formatting
- Deciding the destination for events

These configurations can be changed in runtime by editing the Log4j configuration file (`log4j.xml`).

For more information about Log4j, see `http://logging.apache.org/log4j`.

Event Filtering

By uncommenting or commenting categories in the Log4j configuration file (`log4j.xml`), events can be enabled or disabled as illustrated in Example C–3. The Log4j configuration determines which events generated from the servlets are logged to the `event.log` file.

Example C–3 illustrates a Log4j configuration for displaying 200 (*OK*) responses to INVITE and BYE requests from the `EventTestServlet`. These responses are logged to the `event.log` file.

Example C–3 Log4j Configuration for Displaying BYE Requests and 200 (OK) Responses to INVITE Requests

```
<category
  name="eventlogger.com.example.EventTestServlet.RESPONSE.INVITE.200"
  additivity="false">
  <priority value="ALL"/>
  <appender-ref ref="EVENTLOGGER"/>
  <appender-ref ref="EVENTLOGGER_LOCAL" />
</category>
<category name="eventlogger.com.example.EventTestServlet.REQUEST.BYE"
  additivity="false">
  <priority value="ALL"/>
  <appender-ref ref="EVENTLOGGER"/>
  <appender-ref ref="EVENTLOGGER_LOCAL" />
</category>
```

Note the notation with class name and type of RESPONSE/REQUEST.

Event Formatting and Destination

The format of the events in the `event.log` is determined by the layout class in `Log4j`. The default layout class used for SipServlet event logging is `oracle.sdp.eventlogger.BasicLayout`, and it logs each event as a single row with comma-separated attributes. The method `toString()` is then called on the attribute keys and values, which are concatenated with an equals sign (=). The `BasicLayout` class also adds the attributes described in Table C–4. If needed, you can create a new layout class.

Table C–4 Event Logging Attributes

Attribute Key	Description of Attribute Value
Creation Time	The event creation time in format: yyyy-MM-dd'T'HH:mm:ss.SSS'Z'
TimeZoneOffset	The offset, in minutes from GMT

The configuration also determines where events will be sent, for example to console, file, syslog, remote socket, or JMS. In Example C–4 the events are logged to a file, by using syslog to the localhost (127.0.0.1).

For more information about Log4j, see `http://logging.apache.org/log4j`.

Example C–4 Logging Events to a File

```
<appender name="EVENTLOGGER_LOCAL"
  class="org.oracle.logging.appender.RollingFileAppender">
  <param name="File" value="/var/log/sdp/events.log"/>
  <param name="Append" value="true"/>
  <param name="MaxFileSize" value="100000KB"/>
```

```
  <param name="MaxBackupIndex" value="10"/>
  <layout class="oracle.sdp.eventlogger.BasicLayout">
  </layout>
</appender>
```

Example of BasicLayout Output

Each attribute is displayed in a separate row with the key in bold text.

REQUEST.INVITE:

CreationTime=2005-04-15T07:50:51.767Z,
To=<sip:08505@example.com>;tag=0f54d930-f916-4702-8293-d8b014810a29,
TimeZoneOffset=120,
Method=INVITE,
Content-Type=application/sdp,
Service=eventlogger.com.example.EventTestServlet.REQUEST.INVITE,
Call-ID=8e02578d-2021-4d06-b98d-bf9827c93003@192.0.2.0,
RequestUri=sip:08505@example.com;transport=TCP,
MEDIA=audio 8502 RTP/AVP 97 103 100 101 0 8 102 18 107,
CSeq=2 INVITE,
Content-Length=418,
Via=SIP/2.0/TCP 192.0.2.0:5060;branch=z9hG4bKc549d2be44c901ac050be55ff622e1c8;
rport; SIP/2.0/TCP
192.0.2.0:2051;received=192.0.2.0;branch=z9hG4bK-b20b96e2-
1fac-4500-8030-580da1d81051.1;rport=2051,
From=Alice <sip:alice@example.com>;tag=35ec862a-1d03-4f78-904c-
1accf44d15fe

RESPONSE.INVITE.200

CreationTime=2005-04-15T07:50:51.830Z,To=<sip:08505@example.com>;tag=0f54d930-f916
-4702-8293-d8b014810a29,
TimeZoneOffset=120,
Method=INVITE,
Content-Type=application/sdp,
Service=eventlogger.com.example.EventTestServlet.RESPONSE.INVITE.200,
Call-ID=8e02578d-2021-4d06-b98d-bf9827c93003@192.0.2.0,
StatusCode=200,
MEDIA=audio 8502 RTP/AVP 0 8,
CSeq=2 INVITE,
Content-Length=392,
Via=SIP/2.0/TCP 192.0.2.0:5060;branch=z9hG4bKc549d2be44c901ac050be55ff622e1c8;
rport; SIP/2.0/TCP
192.0.2.0:2051;received=192.0.2.0;branch=z9hG4bK-b20b96e2-
1fac-4500-8030-580da1d81051.1;rport=2051,
From=Alice <sip:alice@example.com>;tag=35ec862a-1d03-4f78-904c-
1accf44d15fe

Index

M

O

P

R

S

T

U