Oracle® Communications Converged Application Server

Diameter Application Development Guide

Release 7.1

F18461-01

May 2019

ORACLE®

Oracle Communications Converged Application Server Diameter Application Development Guide, Release 7.1

F18461-01

Contents

3 Using the Diameter Rf Interface Application for Offline Charging

4 Using the Diameter Ro Interface API for Online Charging

Preface

This document provides an overview of the Oracle Communications Converged Application Server base Diameter protocol packages, classes, and programming model used for developing client and server-side Diameter applications. It also provides an overview of the Converged Application Server Diameter Sh, Rf, and Ro Interface Applications that you can use when developing Diameter protocol applications in your SIP Servlets.

Audience

This document is intended for developers who build and implement Diameter applications in Converged Application Server.

Documentation Accessibility

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

Access to Oracle Support

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

1

Using the Diameter Base Protocol API

This chapter describes using the Diameter Base protocol implementation to create your own Diameter applications in Oracle Communications Converged Application Server.

Overview of Diameter Protocol Support

Diameter is a peer-to-peer protocol that involves delivering attribute-value pairs (AVPs). A Diameter message includes a header and one or more AVPs. The collection of AVPs in each message is determined by the type of Diameter application, and the Diameter protocol also allows for extension by adding new commands and AVPs. Diameter enables multiple peers to negotiate their capabilities with one another, and defines rules for session handling and accounting functions.

Converged Application Server includes an implementation of the base Diameter protocol that supports the core functionality and accounting features described in RFC 3588 (http://www.ietf.org/rfc/rfc3588.txt). Converged Application Server uses the base Diameter functionality to implement multiple Diameter applications, including the Sh, Rf, and Ro applications described later in this document.

You can also use the base Diameter protocol to implement additional client and server-side Diameter applications. The base Diameter API provides a simple, Servlet-like programming model that enables you to combine Diameter functionality with SIP or HTTP functionality in a converged application.

Note: The Diameter protocol offers limited support for clustering for both client and server applications.

The sections that follow provide an overview of the base Diameter protocol packages, classes, and programming model used for developing client and server-side Diameter applications. See also the following sections for information about using the provided Diameter protocol applications in your SIP Servlets:

- "Using the Diameter Sh Interface Application" describes how to access and manage subscriber profile data using the Diameter Sh application.
- "Using the Diameter Rf Interface Application for Offline Charging" describes how to issue offline charging requests using the Diameter Rf application.
- "Using the Diameter Ro Interface API for Online Charging" describes how to perform online charging using the Diameter Ro application.

Working with Diameter Applications Using CDI and POJOs

This section describes the creation and deployment of Diameter applications using Java EE Common Dependency Injections (CDIs) in conjunction with Plain Old Java Objects (POJOs).

Creating a Diameter Bean Using Annotations

Example 1–1 shows an implementation of a Diameter Bean, *ExamplePOJO*, that listens to RoApplication messages.

Example 1–1 Diameter Bean Listening to RoApplication Messages

```
@DiameterBean(applicationId = "4")
public class ExamplePOJO {

  @Inject DiameterSessionSource source;

  public void handleCCR(@Observes CCR ccr) {
    RoSession session = (RoSession) ccr.getSession();
    SipApplicationSession sas = source.getApplicationSession(session);
    //... Business logic...
  }

  public void handleSTR(@Observes @DiameterObserver(code=275) Message msg) {
    RoSession session = (RoSession) msg.getSession();
    SipApplicationSession sas = source.getApplicationSession(session);
    //... Business logic...
  }
}
```

The annotation, @DiameterBean, marks the Java class as a Diameter Bean. The Diameter stack looks for Common Dependency Injection (CDI) observer methods for feeding the requests or responses to the Diameter Bean. The method arguments and the command code, if any, act as a filter for feeding the messages.

Such an architecture avoids the need for registering a listener for subsequent messages in the Diameter session, since the Diameter session information can be obtained from the message.

The RoApplication can then be injected into a SIP servlet as shown in Example 1–2.

Example 1–2 SIP Servlet RoApplication Injection

```
public class FooServlet {

  @Inject @DiameterContext(applicationId = "4")
  Application roApp;

  @Inject
  DiameterSessionSource source;

  void doInvite(SipServletRequest request) throws IOException {
    RoSession session = (RoSession) source.createSession(roApp,
request.getApplicationSession());
    CCR ccr = session.createCCR(RequestType.INITIAL);
    ccr.send();
  }
}
```

Note: For information on the API used in this section, see the *Oracle Communications Converged Application Server Java API Reference*.

Built-in CDI Beans

The Diameter stack adds built-in CDI beans which can be injected into SIP Servlets and Diameter Beans.

SessionSource

This bean acts as a source for Diameter session objects, and helps SIP Servlet applications to associate a diameter Session with a SipApplicationSession and also obtain SipApplicationSession information from a Diameter session.

Application

A **com.bea.wcp.diameter.Application** can be injected with a CDI bean like a SIP servlet or a Diameter bean. The exact subtype of the Application is based on the annotation @DiameterContext.

Example 1–3 shows injecting an instance of RoApplication into a SIP Servlet.

Example 1–3 Injecting an RoApplication Instance into a SIP Servlet

```
public class ExampleServlet extends SipServlet {
  @Inject @DiameterContext(applicationId = 4)
  private Application application;
  ...
}
```

Application Subtypes

Application subtypes, such as RoApplication, RfApplication, ReApplication, RxApplication, GxApplication, GxxApplication, ShApplication, SyApplication can be injected into a CDI bean directly, and do not require the @DiameterContext annotation.

Example 1–4 shows injecting an instance of RoApplication, RfApplication, and ShApplication into a SIP servlet.

Example 1–4 Injecting Diameter Applications into a CDI Bean

```
public class ExampleServlet extends SipServlet {
  @Inject
  private RoApplication roApplication;
  @Inject
  private RfApplication rfApplication;
  @Inject
  private ShApplication shApplication;
  ...
}
```

Diameter Bean Selection

The Diameter stack selects the bean to handle incoming requests based on the following different criteria:

- Application Identifier
- Peer (host/realm)

- Origin (origin host/ origin realm)
- Destination (destination host / destination realm)

The stack finds the most specific Diameter bean to handle the message, irrespective of the application in which it is packaged. The following rules determine the most specific bean:

- If the bean contains and matches all the selection criteria annotations, then it is considered most specific.
- If the bean contains and matches a fewer number of selection critera annotations, then priority is given in the following order:

  Application then **Peer** then **Destination** then **Origin**.
- For matching a node (**Peer**, **Origin** or **Destination**), priority is given to host rather than realm.

Example 1–5 shows a bean with a more complicated set of selection criteria. When a Diameter message comes in, **ExampleBean1** is selected first, followed by the remaining beans in the following order:

ExampleBean2 then **ExampleBean3** then **ExampleBean4** then **ExampleBean5**

Example 1–5 Diameter Bean Selection Criteria

```
@DiameterBean(
applicationId = 4,
peers = { @DiameterNode(host = "ro.server", realm = "weblogic.com") },
origins = { @DiameterNode(host = "originserver", realm = "originserver.com"),
@DiameterNode(host = "ro.client", realm = "weblogic.com") },
destinations = { @DiameterNode(host = "ro.server", realm = "weblogic.com") })
public class ExampleBean1 {
}

//specify peers.
@DiameterBean(
applicationId = 4,
peers = { @DiameterNode(host = "ro.server", realm = "weblogic.com") },
public class ExampleBean2 {
}

//specify origins.
@DiameterBean(
applicationId = 4,
origins = { @DiameterNode(host = "originserver", realm = "originserver.com"),
@DiameterNode(host = "ro.client", realm = "weblogic.com") },
public class ExampleBean3 {
}

//destinations specify both host and realm.
@DiameterBean(
applicationId = 4,
destinations = { @DiameterNode(host = "ro.server", realm = "weblogic.com") })
public class ExampleBean3 {
}

//destinations only specify host.
@DiameterBean(
applicationId = 4,
destinations = { @DiameterNode(host = "ro.server") })
public class ExampleBean4 {
```

```
}

//destinations only specify realm.
@DiameterBean(
applicationId = 4,
destinations = { @DiameterNode(realm = "weblogic.com") })
public class ExampleBean5 {
}
}
```

Filtering Observer Methods Based on Command Codes and Parameters

A Diameter bean may need to filter the messages based on the command code of the message. For example, if the Diameter bean wants to receive Session Terminated Answer (STA) and Credit Control Answer (CCA) messages on two different methods, it can do so by using the @DiameterObserver annotation. If the @DiameterObserver annotation is specified in the method, then the code injected by @DiameterObserver will be used by the Diameter stack (CDI) for delivering the message to the correct method.

If two methods have the same @DiameterObserver annotation, the Diameter bean selects the method where the parameter is the same type as the message.

Example 1–6 illustrates this behavior.

Example 1–6 Filtering Observer Methods

```
public void handleCCA(@Observes @DiameterObserver(code=272) CCA cca) {
}

public void handleAnswer(@Observes @DiameterObserver(code=272) Answer answer) {
}

public void handleSTA(@Observes @DiameterObserver(code=275) Answer answer) {
}

public void handleCCR(@Observes @DiameterObserver(code=272) CCR ccr) {
}

public void handleRequest(@Observes @DiameterObserver(code=272) Request request) {
}

public void handleMessage(@Observes Message message) {
}
```

In Example 1–6 the following filtering occurs:

- When a CCA message comes in, it is delivered to **handleCCA**() not **handleAnswer**(), because the **handleCCA**() parameter *CCA* matches the CCA message type exactly.
- When an Answer message (not a CCA) with the command code 272 comes in, it it is delivered to the **handleAnswer**() method.
- When an STA message comes in, it is delivered to the **handleSTA**() method.
- When a CCR message comes in, it is delivered to the **handleCCR**() method rather than the **handleRequest**() method.
- When a request message (not a CCR) with command code 272 comes in, it is delivered to the **handleRequest**() method.

- When other Diameter messages come in, they are delivered to the **handleMessage**() method.

Dynamic Configuration

Since the *host* and *realm* values of a @DiameterNode may need to be updated frequently, Converged Application Server leverages Java Enterprise Edition Environmental Entries, and lets you define them as variables, placing their values in SIP application deployment descriptors, such as **sip.xml**, or **web.xml** rather than hard coding them in source code. The deployment descriptors can be then be modified without requiring an application recompile. Alternatively, you can use WebLogic deployment plans to modify environmental variables.

In a @DiameterNode if host or realm is defined using the pattern, ${*xxxx*}, the *xxxx* is treated as a variable name. That variable name is matched to **env-entry-name** elements in sip.xml or web.xml, and, when a match occurs, the associated value is returned, replacing ${xxxx}.

The host or realm variable can defined by using letters, digits, hyphen, dots and underscore character.

Example 1–7 is a typical **sip.xml** configuration file with two env-entry elements containing env-entry-name child elements.

Example 1–7 sip.xml Configuration File

```
<?xml version="1.0" encoding="UTF-8"?>
<sip-app xmlns="http://www.jcp.org/xml/ns/sipservlet"
xmlns:javaee="http://java.sun.com/xml/ns/javaee">
  <app-name>FooServlet</app-name>
  <javaee:display-name>diameter abstract API Test</javaee:display-name>
  <distributable />
  <javaee:env-entry>
    <javaee:description>peer server host name</javaee:description>
    <javaee:env-entry-name>peer-server</javaee:env-entry-name>
    <javaee:env-entry-type> java.lang.String </javaee:env-entry-type>
    <javaee:env-entry-value>ro.client</javaee:env-entry-value>
  </javaee:env-entry>
  <javaee:env-entry>
    <javaee:description>peer server realm</javaee:description>
    <javaee:env-entry-name>peer-realm</javaee:env-entry-name>
    <javaee:env-entry-type> java.lang.String </javaee:env-entry-type>
    <javaee:env-entry-value>example.com</javaee:env-entry-value>
  </javaee:env-entry>
</sip-app>
```

In Example 1–8, the host value for @DiameterNode is defined as ${*peer-server*}. To retrieve the value of *peer-server* from sip.xml in Example 1–7, the **env-entry** elements are searched for an **env-entry-name** element that matches *peer-server*. The match in this case is the **env-entry-value**, *ro.client*. Likewise, the realm value, *peer-realm* from ${*peer-realm*} matches the **env-entry-name**, *peer-realm*, which then returns the **env-entry-value**, *example.com*.

Example 1–8 Referencing sip.xml Configuration Parameters

```
@DiameterBean
(applicationId= 4,
  peers= {@DiameterNode (host = "${peer-server}", realm ="${peer-realm}")})
public class ExampleBean {
}
```

The realm and host values of @DiameterNode can also be changed using a WebLogic deployment plan.

Example 1–9 illustrates a WebLogic Deployment Plan file, **plan.xml**.

Example 1–9 WebLogic Deployment Plan

```
<?xml version='1.0' encoding='UTF-8'?>
<deployment-plan xmlns="http://xmlns.oracle.com/weblogic/deployment-plan"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.oracle.com/weblogic/deployment-plan
http://xmlns.oracle.com/weblogic/deployment-plan/1.0/deployment-plan.xsd">
  <application-name>thirdpartyprotocol</application-name>
  <variable-definition>
    <variable>
      <name>new_peer-server</name>
      <value>ro.client2</value>
    </variable>
    <variable>
      <name>new_peer-realm</name>
      <value>example.com2</value>
    </variable>
  </variable-definition>
  <module-override>
    <module-name>thirdpartyprotocol.war</module-name>
    <module-type>war</module-type>
    <module-descriptor external="false">
      <root-element>web-app</root-element>
      <uri>WEB-INF/web.xml</uri>
    </module-descriptor>
    <module-descriptor external="false">
      <root-element>sip-app</root-element>
      <uri>WEB-INF/sip.xml</uri>
      <variable-assignment>
        <name>new_peer-server</name>
<xpath>/sip-app/env-entry/[env-entry-name="peer-server"]/env-entry-value</xpath>
        <operation>replace</operation>
      </variable-assignment>
      <variable-assignment>
        <name>new_peer-realm</name>
<xpath>/sip-app/env-entry/[env-entry-name="peer-realm"]/env-entry-value</xpath>
        <operation>replace</operation>
      </variable-assignment>
    </module-descriptor>
  </module-override>
</deployment-plan>
```

In Example 1–9 two variables are defined, *new_peer-server* and *new_peer-realm*, which are then used to replace the env-entry-values in sip.xml after plan.xml is deployed. You can use the WebLogic Administration Console to specify a deployment plan for your application, or you can use the **weblogic.Deployer** to deploy an application with the **-plan** parameter:

```
java weblogic.Deployer -adminurl http://hostname:port -user username -password
password -deploy -name application-name -source application-name.war -targets
server-name -stage -plan plan.xml
```

Legacy Diameter Application Development

This section describes developing Diameter applications using version 1.x legacy Java techniques.

Diameter Application Development Environment Configuration

This section describes requirements for compiling legacy Diameter applications as well as configuring Diameter nodes.

File Required for Compiling Application Using the Diameter API

The wlssdiameter.jar file is part of the exposed Diameter API. To compile against this API, you must access the wlssdiameter.jar, which is located in the directory:

Middleware_Home/Oracle_Home/wlserver/sip/server/lib/

Where *MW_home* is the directory in which the Converged Application Server software is installed (the installation program used to install Converged Application Server refers to this as Middleware Home). For example:

```
/Oracle/Middleware/Oracle_Home/wlserver/sip/server/lib/
```

Configuring Diameter Nodes

A Diameter node is represented by the `com.bea.wcp.diameter.Node` class. A Diameter node may host one or more Diameter applications, as specified in the `diameter.xml` configuration file, located in the directory:
Middleware_Home/**Oracle_Home/user_projects/domains/***domain_name*/**config/custom/**

Where *Middleware_Home* is the directory in which the Converged Application Server software is installed, and *domain_name* is the name of the Diameter domain. For example:

```
/Oracle/Middleware/Oracle_Home/user_projects/domains/Diameter_domain/config/custom
```

Diameter nodes are generally configured and started as part of a Converged Application Server instance. However, for development and testing purposes, you can also run a Diameter node as a standalone process. To do so:

1. Set the environment for the Diameter domain using the `setDomainEnv.sh` (UNIX) or `setDomainEnv.cmd` (Windows) command located in the directory:
   Middleware_Home/**Oracle_Home/user_projects/domains/***domain_name*/**diameter/bin/**

   Where *Middleware_Home* is the directory where you installed the Converged Application Server software and *my_domain* is the name of the domain's directory. For example:

   ```
   cd
   /Oracle/Middleware/Oracle_Home/user_projects/domains/Diameter_domain/diameter/b
   in
   ./setDomainEnv.sh
   ```

2. Make the directory containing the `diameter.xml` configuration file for the Diameter Node you want to start your working directory. For example:

   ```
   cd
   /Oracle/Middleware/Oracle_Home/user_projects/domains/Diameter_domain/config/cus
   tom
   ```

3. Set the Java class path for the Diameter domain to the file: **com.bea.core.process_5.4.0.0.jar**

```
java -classpath
$CLASSPATH:/Oracle/Middleware/wlserver/server/lib/consoleapp/APP-INF/lib/com.bea.core.process_5.4.0.0.jar
```

4. Start the Diameter Node, specifying the `diameter.xml` configuration file to use with the domain:

```
java com.bea.wcp.diameter.Node ./diameter.xml
```

Overview of the Diameter API

All classes in the Diameter base protocol API reside in the `com.bea.wcp.diameter` package. Table 1–1 describes the key classes, interfaces, and exceptions in this package.

Table 1–1 Key Elements of the Diameter Base Protocol API

Category	Element	Description
Diameter Node	Node	A class that represents a Diameter node implementation. A diameter node can represent a client- or server-based Diameter application, as well as a Diameter relay agent.
Diameter Applications	Application, ClientApplication	A class that represents a basic Diameter application. ClientApplication extends Application for client-specific features such as specifying destination hosts and realms. All Diameter applications must extend one of these classes to return an application identifier. The classes can also be used directly to create new Diameter sessions.
Diameter Applications	ApplicationId	A class that represents the Diameter application ID. This ID is used by the Diameter protocol for routing messages to the appropriate application. The `ApplicationId` corresponds to one of the Auth-Application-Id, Acct-Application-Id, or Vendor-Specific-Application-Id AVPs contained in a Diameter message.
Diameter Applications	Session	A class that represents a Diameter session. Applications that perform session-based handling must extend this class to provide application-specific behavior for managing requests and answering messages.
Message Processing	Message, Request, Answer	The Message class is a base class used to represent request and answer message types. Request and Answer extend the base class.
Message Processing	Command	A class that represents a Diameter command code.
Message Processing	RAR, RAA	These classes extend the `Request` and `Answer` classes to represent re-authorization messages.
Message Processing	ResultCode	A class that represents a Diameter result code, and provides constant values for the base Diameter protocol result codes.
AVP Handling	Attribute	A class that provides Diameter attribute information.
AVP Handling	Avp, AvpList	Classes that represent one or more attribute-value pairs in a message. AvpList is also used to represent AVPs contained in a grouped AVP.
AVP Handling	Type	A class that defines the supported AVP datatypes.
Error Handling	DiameterException	The base exception class for Diameter exceptions.
Error Handling	MessageException	An exception that is raised when an invalid Diameter message is discovered.
Error Handling	AvpException	An exception that is raised when an invalid AVP is discovered.

Table 1–1 (Cont.) Key Elements of the Diameter Base Protocol API

Category	Element	Description
Supporting Interfaces	Enumerated	An enum value that implements this interface can be used as the value of an AVP of type INTEGER32, INTEGER64, or ENUMERATED.
Supporting Interfaces	SessionListener	An interface that applications can implement to subscribe to messages delivered to a Diameter session.
Supporting Interfaces	MessageFactory	An interface that allows applications to override the default message decoder for received messages, and create new types of `Request` and `Answer` objects. The default decoding process begins by decoding the message header from the message bytes using an instance of `MessageFactory`. This is done so that an early error message can be generated if the message header is invalid. The actual message AVPs are decoded in a separate step by calling `decodeAvps`. AVP values are fully decoded and validated by calling `validate`, which in turn calls `validateAvp` for each partially-decoded AVP in the message.

In addition to these base Diameter classes, accounting-related classes are stored in the `com.bea.wcp.diameter.accounting` package, and credit-control-related classes are stored in `com.bea.wcp.diameter.cc`. See Chapter 4, "Using the Diameter Ro Interface API for Online Charging" and Chapter 3, "Using the Diameter Rf Interface Application for Offline Charging" for more information about classes in these packages.

Working with Diameter Nodes

In order to access a Diameter application, a deployed application (such as a SIP Servlet) must obtain the Diameter Node instance and request the application. Example 1–10 shows the sample code used to access the Rf application.

Example 1–10 Accessing a Diameter Node and Application

```
ServletContext sc = getServletConfig().getServletContext();
Node node = sc.getAttribute("com.bea.wcp.diameter.Node");
RfApplication rfApp = (RfApplication)
node.getApplication(Charging.RF_APPLICATION_ID);
```

Implementing a Legacy Diameter Application

All Diameter applications must extend either the base `Application` class or, for client applications, the `ClientApplication` class. The model for creating a Diameter application is similar to that for implementing Servlets in the following ways:

- Diameter applications override the `init()` method for initialization tasks.
- Initialization parameters configured for the application in `diameter.xml` are made available to the application.
- A session factory is used to generate new application sessions.

Diameter applications must also implement the `getId()` method to return the proper application ID. This ID is used to deliver Diameter messages to the correct application.

Applications can optionally implement `rcvRequest()` or `rcvAnswer()` as needed. By default, `rcvRequest()` answers with UNABLE_TO_COMPLY, and `rcvRequest()` drops the Diameter message.

Example 1–11 shows a simple Diameter client application that does not use sessions.

Example 1–11 Simple Diameter Application

```
public class TestApplication extends ClientApplication {
  protected void init() {
    log("Test application initialized.");
  }
  public ApplicationId getId() {
    return ApplicationId.BASE_ACCOUNTING;
  }
  public void rcvRequest(Request req) throws IOException {
    log("Got request: " + req.getHopByHopId());
    req.createAnswer(ResultCode.SUCCESS).send();
  }
}
```

Working with Diameter Sessions

Applications that perform session-based handling must extend the base Session class to provide application-specific behavior for managing requests and answering messages. If you extend the base Session class, you must implement either `rcvRequest()` or `rcvAnswer()`, and may implement both methods.

The base Application class is used to generate new Session objects. After a session is created, all session-related messages are delivered directly to the session object. The Converged Application Server container automatically generates the session ID and encodes the ID in each message. Session attributes are supported much in the same fashion as attributes in `SipApplicationSession`.

Example 1–12 shows a simple Diameter session implementation.

Example 1–12 Simple Diameter Session

```
public class TestSession extends Session {
  public TestSession(TestApplication app) {
    super(app);
  }
  public void rcvRequest(Request req) throws IOException {
    getApplication().log("rcvReuest: " + req.getHopByHopId());
    req.createAnswer(ResultCode.SUCCESS).send();
  }
}
```

To use the sample session class, the `TestApplication` in Example 1–11 would need to add a factory method:

```
public class TestApplication extends Application {
  ...
  public TestSession createSession() {
    return new TestSession(this);
  }
}
```

`TestSession` could then be used to create new requests as follows:

```
TestSession session = testApp.createSession();
Request req = session.creatRequest();
req.sent();
```

The answer is delivered directly to the Session object.

Note: While a typical Converged Application Server Diameter client application does not need to concern itself with session management, a Diameter server applications must call `diameterSession.terminate()` to invalidate a session when it is no longer needed.

Working with Diameter Messages

The base `Message` class is used for both Request and Answer message types. A Message always includes an application ID, and optionally includes a session ID. By default, messages are handled in the following manner:

1. The message bytes are parsed.
2. The application and session ID values are determined.
3. The message is delivered to a matching session or application using the following rules:
   a. If the Session-Id AVP is present, the associated Session is located and the session's `rcvMessage()` method is called.
   b. If there is no Session-Id AVP present, or if the session cannot be located, the Diameter application's `rcvMessage()` method is called.
   c. If the application cannot be located, an UNABLE_TO_DELIVER response is generated.

The message type is determined from the Diameter command code. Certain special message types, such as RAR, RAA, ACR, ACA, CCR, and CCA, have getter and setter methods in the `Message` object for convenience.

Sending Request Messages

Either a `Session` or `Application` can originate and receive request messages. Requests are generated using the `createRequest()` method. You must supply a command code for the new request message. For routing purposes, the destination host or destination realm AVPs are also generally set by the originating session or application.

Received answers can be obtained using `Request.getAnswer()`. After receiving an answer, you can use `getSession()` to obtain the relevant session ID and `getResultCode()` to determine the result. You can also use `Answer.getRequest()` to obtain the original request message.

Requests can be sent asynchronously using the `send()` method, or synchronously using the blocking `sendAndWait()` method. Answers for requests that were sent asynchronously are delivered to the originating session or application. You can specify a request timeout value when sending the message, or can use the global `request-timeout` configuration element in `diameter.xml`. An UNABLE_TO_DELIVER result code is generated if the timeout value is reached before an answer is delivered. `getResultCode()` on the resulting Answer returns the result code.

Sending Answer Messages

New answer messages are generated from the `Request` object, using `createAnswer()`. All generated answers should specify a ResultCode and an optional Error-Message AVP value. The `ResultCode` class contains pre-defined result codes that can be used.

Answers are delivered using the `send()` method, which is always asynchronous (non-blocking).

Creating New Command Codes

A Diameter command code determines the message type. For instance, when sending a request message, you must supply a command code.

The `Command` class represents pre-defined commands codes for the Diameter base protocol, and can be used to create new command codes. Command codes share a common name space based on the code itself.

The `define()` method enables you to define codes, as in:

```
static final Command TCA = Command.define(1234, "Test-Request", true, true);
```

The `define()` method registers a new Command, or returns a previous command definition if one was already defined. Commands can be compared using the reference equality operator (==).

Working with AVPs

Attribute Value Pair (AVP) is a method of encapsulating information relevant to the Diameter message. AVPs are used by the Diameter base protocol, the Diameter application, or a higher-level application that employs Diameter.

The `Avp` class represents a Diameter attribute-value pair. You can create new AVPs with an attribute value in the following way:

```
Avp avp = new Avp(Attribute.ERROR_MESSAGE, "Bad request");
```

You can also specify the attribute name directly, as in:

```
Avp avp = new Avp("Error-Message", "Bad request");
```

The value that you specify must be valid for the specified attribute type.

To create a grouped AVP, use the `AvpList` class, as in:

```
AvpList avps = new AvpList();
avps.add(new Avp("Event-Timestamp", 1234));
avps.add(new Avp("Vendor-Id", 1111));
```

Creating New Attributes

You can create new attributes to extend your Diameter application. The `Attribute` class represents an AVP attribute, and includes the AVP code, name, flags, optional vendor ID, and type of attribute. The class also maintains a registry of defined attributes. All attributes share a common namespace based on the attribute code and vendor ID.

The `define()` method enables you to define new attributes, as in:

```
static final Attribute TEST = Attribute.define(1234, "Test-Attribute", 0,
Attribute.FLAG_MANDATORY, Type.INTEGER32);
```

Table 1–2 lists the available attribute types and describes how they are mapped to Java types.

The `define()` method registers a new attribute, or returns a previous definition if one was already defined. Attributes can be compared using the reference equality operator (==).

Table 1–2 Attribute Types

Diameter Type	Type Constant	Java Type
Integer32	Type.INTEGER32	Integer
Integer64	Type.INTEGER64	Long
Float32	Type.FLOAT32	Float
OctetString	Type.BYTES	ByteBuffer (read-only)
UTF8String	Type.STRING	String
Address	Type.ADDRESS	InetAddress
Grouped	Type.GROUPED	AvpList

Creating Converged Diameter and SIP Applications

The Diameter API enables you to create converged applications that utilize both SIP and Diameter functionality. A SIP Servlet can access an available Diameter application through the Diameter Node, as shown in Example 1–13.

Example 1–13 Accessing the Rf Application from a SIP Servlet

```
ServletContext sc = getServletConfig().getServletContext();
Node node = (Node) sc.getAttribute("com.bea.wcp.diameter.Node");
RfApplication rfApp = (RfApplication)
node.getApplication(Charging.RF_APPLICATION_ID);
```

SIP uses Call-id (the SIP-Call-ID header) to identify a particular call session between two users. Converged Application Server automatically links a Diameter session to the currently-active call state by encoding the SIP Call-id into the Diameter session ID. When a Diameter message is received, the container automatically retrieves the associated call state and locates the Diameter session. A Diameter session is serializable, so you can store the session as an attribute in a the `SipApplicationSession` object, or vice versa.

Converged applications can use the Diameter `SessionListener` interface to receive notification when a Diameter message is received by the session. The `SessionListener` interface defines a single method, `rcvMessage()`. Example 1–14 shows an example of how to implement the method.

Example 1–14 Implementing SessionListener

```
Session session = app.createSession();
session.setListener(new SessionListener() {
  public void rcvMessage(Message msg) {
    if (msg.isRequest()) System.out.println("Got request!");
  }
});
```

Note: `The SessionListener` implementation must be serializable for distributed applications.

2

Using the Diameter Sh Interface Application

This chapter describes how to use the Diameter Sh interface application, based on the Oracle Communications Converged Application Server Diameter protocol implementation, in your own applications.

Overview of Profile Service API and Sh Interface Support

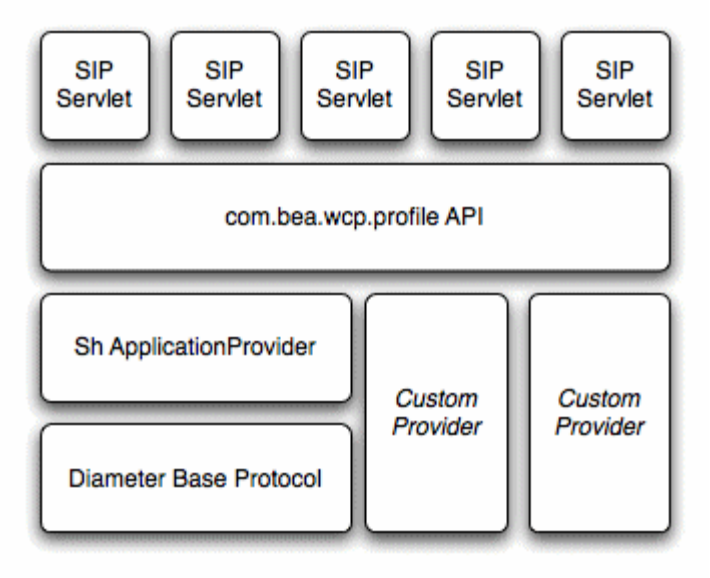
The IP Multimedia Subsystem (IMS) specification defines the Sh interface as the method of communication between the Application Server (AS) function and the Home Subscriber Server (HSS), or between multiple IMS Application Servers. The AS uses the Sh interface in two basic ways:

- To query or update a user's data stored on the HSS
- To subscribe to and receive notifications when a user's data changes on the HSS

The user data available to an AS may be defined by a service running on the AS (*repository data*), or it may be a subset of the user's IMS profile data hosted on the HSS. The Sh interface specification, 3GPP TS 29.328, defines the IMS profile data that can be queried and updated through Sh. All user data accessible through the Sh interface is presented as an XML document with the schema defined in 3GPP TS 29.328.

The IMS Sh interface is implemented as a provider to the base Diameter protocol support in ProductNameShort. The provider transparently generates and responds to the Diameter command codes defined in the Sh application specification. A higher-level Profile Service API enables SIP Servlets to manage user profile data as an XML document using XML Document Object Model (DOM). Subscriptions and notifications for changed profile data are managed by implementing a profile listener interface in a SIP Servlet.

Figure 2–1 Profile Service API and Sh Provider Implementation



Converged Application Server includes a provider for the Diameter Sh interface. Providers to support additional interfaces defined in the IMS specification may be provided in future releases. Applications using the profile service API are able to use additional providers as they are made available.

Enabling the Sh Interface Provider

See the chapter "Configuring Diameter Client Nodes and Relay Agents" in the *Converged Application Server Administrator's Guide* for information on enabling Diameter support.

Overview of the Profile Service API

Converged Application Server provides a simple profile service API that SIP Servlets can use to query or modify subscriber profile data, or to manage subscriptions for receiving notifications about changed profile data. Using the API, a SIP Servlet explicitly requests user profile documents through the Sh provider application. The provider returns an XML document, and the Servlet can then use standard DOM techniques to read or modify profile data in the local document. Updates to the local document are applied to the HSS after a "put" operation.

Creating a Document Selector Key for Application-Managed Profile Data

The document selector key identifies the XML document to be retrieved by a Diameter interface, and uses the format `protocol://uri/reference_type[/access_key]`. Servlets that manage profile data can explicitly obtain an Sh XML document from a factory using a key, and then work with the document using DOM.

The `protocol` portion of the selector identifies the Diameter interface provider to use for retrieving the document. Sh XML documents require the `sh://` protocol designation.

With Sh document selectors, the next element, `uri`, generally corresponds to the User-Identity or Public-Identity of the user whose profile data is being retrieved. If you

are requesting an Sh data reference of type LocationInformation or UserState, the URI value can be the User-Identity or MSISDN for the user.

Table 2–1 summarizes the possible URI values that can be supplied depending on the Sh data reference you are requesting. 3GPP TS 29.328 describes the possible data references and associated reference types in more detail.

Table 2–1 Possible URI Values for Sh Data References

Sh Data Reference Number	Data Reference Type	Possible URI Value in Document Selector
0	RepositoryData	User-Identity or Public-Identity
10	IMSPublicIdentity	NA
11	IMSUserState	NA
12	S-CSCFName	NA
13	InitialFilterCriteria	NA
14	LocationInformation	User-Identity or MSISDN
15	UserState	NA
17	Charging information	User-Identity or Public-Identity
17	MSISDN	NA

The final element of the document selector key, `reference_type`, specifies the data reference type being requested. For some data reference requests, only the `uri` and `reference_type` are required. Other Sh requests use an access key, which requires a third element in the document selector key corresponding to the value of the Attribute-Value Pair (AVP) defined in the document selector key.

Table 2–2 summarizes the required document selector key elements for each type of Sh data reference request.

Table 2–2 Summary of Document Selector Elements for Sh Data Reference Requests

Data Reference Type	Required Document Selector Elements	Example Document Selector
RepositoryData	sh://uri/reference_type/Service-Indication	sh://sip:user@oracle.com/RepositoryData/Call Screening/
IMSPublicIdentity	sh://uri/reference_type/[*Identity-Set*] where *Identity-Set* is one of: ▪ All-Identities ▪ Registered-Identities ▪ Implicit-Identities	sh://sip:user@oracle.com/IMSPublicIdentity/Registered-Identities
IMSUserState	sh://uri/reference_type	sh://sip:user@oracle.com/IMSUserState/
S-CSCFName	sh://uri/reference_type	sh://sip:user@oracle.com/S-CSCFName/
InitialFilterCriteria	sh://uri/reference_type/Server-Name	sh://sip:user@oracle.com/InitialFilterCriteria/www.oracle.com/
LocationInformation	sh://uri/reference_type/(CS-Domain | PS-Domain)	sh://sip:user@oracle.com/LocationInformation/CS-Domain/

Table 2–2 (Cont.) Summary of Document Selector Elements for Sh Data Reference Requests

Data Reference Type	Required Document Selector Elements	Example Document Selector
UserState	sh://uri/reference_type/ (CS-Domain | PS-Domain)	sh://sip:user@oracle.com/UserState/PS-Domain/
Charging information	sh://uri/reference_type	sh://sip:user@oracle.com/Charging information/
MSISDN	sh://uri/reference_type	sh://sip:user@oracle.com/MSISDN/

Using a Constructed Document Key to Manage Profile Data

Converged Application Server provides a helper class, `com.bea.wcp.profile.ProfileService`, to help you easily retrieve a profile data document. The `getDocument()` method takes a constructed document key, and returns a read-only `org.w3c.dom.Document` object. To modify the document, you make and edit a copy, then send the modified document and key as arguments to the `putDocument()` method.

> **Note:** If Diameter Sh client node services are not available on the Converged Application Server instance when `getDocument()` method is invoked, the profile service throws a "No registered provider for protocol" exception.

Converged Application Server caches the documents returned from the profile service for the duration of the service method invocation (for example, when a `doRequest()` method is invoked). If the service method requests the same profile document multiple times, the subsequent requests are served from the cache rather than by re-querying the HSS.

Example 2–1 shows a sample SIP Servlet that obtains and modifies profile data.

Example 2–1 Sample Servlet Using ProfileService to Retrieve and Write User Profile Data

```
package demo;
import com.bea.wcp.profile.*;
import javax.servlet.sip.SipServletRequest;
import javax.servlet.sip.SipServlet;
import org.w3c.dom.Document;
import java.io.IOException;
public class MyServlet extends SipServlet {
      private ProfileService psvc;
      public void init() {
        psvc = (ProfileService)
getServletContext().getAttribute(ProfileService.PROFILE_SERVICE);
      }
      protected void doInvite(SipServletRequest req) throws IOException {
        String docSel = "sh://" + req.getTo() + "/IMSUserState/";
        // Obtain and change a profile document.
        Document doc = psvc.getDocument(docSel); // Document is read only.
        Document docCopy = (Document) doc.cloneNode(true);
        // Modify the copy using DOM.
        psvc.putDocument(docSel, docCopy); // Apply the changes.
      }
```

```
}
```

Monitoring Profile Data with ProfileListener

The IMS Sh interface enables applications to receive automatic notifications when a subscriber's profile data changes. Converged Application Server provides an easy-to-use API for managing profile data subscriptions. A SIP Servlet registers to receive notifications by implementing the `com.bea.wcp.profile.ProfileListener` interface, which consists of a single `update` method that is automatically invoked when a change occurs to the profile to which the Servlet is subscribed. Notifications are not sent if that same Servlet modifies the profile information (for example, if a user modifies their own profile data).

> **Note:** In a replicated environment, Diameter relay nodes always attempt to push notifications directly to the engine tier server that subscribed for profile updates. If that engine tier server is unavailable, another server in the engine tier cluster is chosen to receive the notification. This model succeeds because session information is stored in the SIP data tier, rather than the engine tier.

Prerequisites for Listener Implementations

In order to receive a call back for subscribed profile data, a SIP Servlet must do the following:

- Implement `com.bea.wcp.profile.ProfileListener`.
- Create one or more subscriptions using the `subscribe` method in the `com.bea.wcp.profile.ProfileService` helper class.
- Register itself as a listener using the `listener` element in `sip.xml`.

"Implementing ProfileListener" describes how to implement `ProfileListener` and use the `susbscribe` method. In addition to having a valid listener implementation, the Servlet must declare itself as a listener in the `sip.xml` deployment descriptor file. For example, it must add a `listener` element declaration similar to:

```
<listener>
   <lisener-class>com.mycompany.MyLisenerServlet</listener-class>
</listener>
```

Implementing ProfileListener

Actual subscriptions are managed using the `subscribe` method of the `com.bea.wcp.profile.ProfileService` helper class. The subscribe method requires that you supply the current `SipApplicationSession` and the key for the profile data document you want to monitor. See "Creating a Document Selector Key for Application-Managed Profile Data" for more information.

Applications can cancel subscriptions by calling `ProfileSubscription.cancel()`. Also, pending subscriptions for an application are automatically cancelled if the application session is terminated.

Example 2–2 shows sample code for a Servlet that implements the `ProfileListener` interface.

Example 2–2 Sample Servlet Implementing ProfileListener Interface

```
package demo;
    import com.bea.wcp.profile.*;
    import javax.servlet.sip.SipServletRequest;
    import javax.servlet.sip.SipServlet;
    import org.w3c.dom.Document;
    import java.io.IOException;
    public class MyServlet extends SipServlet implements ProfileListener {
      private ProfileService psvc;
      public void init() {
        psvc = (ProfileService)
        getServletContext().getAttribute(ProfileService.PROFILE_SERVICE);
      }
      protected void doInvite(SipServletRequest req) throws IOException {
        String docSel = "sh://" + req.getTo() + "/IMSUserState/";
        // Subscribe to profile data.
        psvc.subscribe(req.getApplicationSession(), docSel, null);
      }
      public void update(ProfileSubscription ps, Document document) {
        System.out.println("IMSUserState updated: " + ps.getDocumentSelector());
      }
    }
```

3

Using the Diameter Rf Interface Application for Offline Charging

This chapter describes how to use the Diameter Rf interface application, based on the Oracle Communications Converged Application Server Diameter protocol implementation, in your own applications.

Overview of Rf Interface Support

Offline charging is used for network services that are paid for periodically. For example, a user may have a subscription for voice calls that is paid monthly. The Rf protocol allows an IMS Charging Trigger Function (CTF) to issue offline charging events to a Charging Data Function (CDF). The charging events can either be one-time events or may be session-based.

Converged Application Server provides a Diameter Offline Charging Application that can be used by deployed applications to generate charging events based on the Rf protocol. The offline charging application uses the base Diameter protocol implementation, and allows any application deployed on Converged Application Server to act as CTF to a configured CDF.

For basic information about offline charging, see RFC 3588: Diameter Base Protocol (http://www.ietf.org/rfc/rfc3588.txt). For more information about the Rf protocol, see 3GPP TS 32.299 (http://www.3gpp.org/ftp/Specs/html-info/32299.htm).

Understanding Offline Charging Events

For both event and session based charging, the CTF implements the accounting state machine described in RFC 3588. The server (CDF) implements the accounting state machine "SERVER, STATELESS ACCOUNTING" as specified in RFC 3588.

The reporting of offline charging events to the CDF is managed through the Diameter Accounting Request (ACR) message. Rf supports the ACR event types described in Table 3–1.

Table 3–1 Rf ACR Event Types

Request	Description
START	Starts an accounting session.
INTERIM	Updates an accounting session.
STOP	Stops an accounting session.

Table 3–1 (Cont.) Rf ACR Event Types

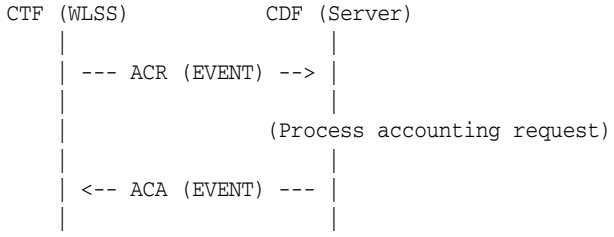
Request	Description
EVENT	Indicates a one-time accounting event.

The START, INTERIM, and STOP event types are used for session-based accounting. The EVENT type is used for event based accounting, or to indicate a failed attempt to establish a session.

Event-Based Charging

Event-based charging events are reported through the ACR EVENT message. Example 3–1 shows the basic message flow.

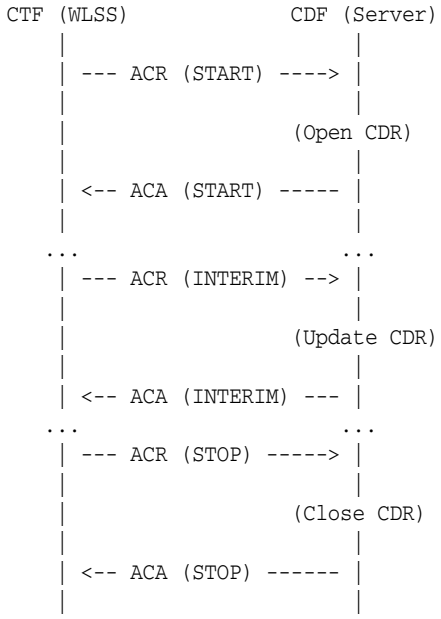
Example 3–1 Message Flow for Event-Based Charging



Session-Based Charging

Session-based charging uses the ACR START, INTERIM, and STOP requests to report usage to the CDF. During a session, the CTF may report multiple ACR INTERIM requests depending on the session lifecycle. Example 3–2 shows the basic message flow.

Example 3–2 Message Flow for Session-Based Charging



Here, ACA START is sent a receipt of a service request by Converged Application Server. ACA INTERIM is typically sent upon expiration of the AII timer. ACA STOP is issued upon request for service termination by Converged Application Server.

Configuring the Rf Application

The `Rf` API is packaged as a Diameter application similar to the Sh application used for managing profile data. The Rf Diameter API can be configured and enabled by editing the Diameter configuration file located in `Domain_Home/config/custom/diameter.xml`, or by using the Diameter console extension. Additionally, configuration of both the CDF realm and host can be specified using the cdf.realm and cdf.host initialization parameters to the Diameter Rf application.

Example 3–3 shows a sample excerpt from `diameter.xml` that enables Rf with a CDF realm of "example.com" and host "cdf.example.com:"

Example 3–3 Sample Rf Application Configuration (diameter.xml)

```
<application>
  <name>rfcharging</name>
  <class-name>com.bea.wcp.diameter.charging.RfApplication</class-name>
  <param>
    <name>cdf.realm</name>
    <value>example.com</value>
  </param>
  <param>
    <name>cdf.host</name>
    <value>cdf.example.com</value>
  </param>
</application>
```

Because the `RfApplication` uses the Diameter base accounting messages, its Diameter application id is 3 and there is no vendor ID.

Using the Offline Charging API

Converged Application Server provides an offline charging API to enable any deployed application to act as a CTF and issue offline charging events. This API supports both event-based and session-based charging events.

The classes in package `com.bea.wcp.diameter.accounting` provide general support for Diameter accounting messages and sessions. Table 3–2 summarizes the classes.

Table 3–2 Diameter Accounting Classes

Class	Description
ACR	An Accounting-Request message.
ACA	An Accounting-Answer message.
ClientSession	A Client-based accounting session.
RecordType	Accounting record type constants.

In addition, classes in package `com.bea.wcp.diameter.charging` support the Rf application specifically. Table 3–3 summarizes the classes.

Table 3–3 Diameter Rf Application Support Classes

Charging	Common definitions for 3GPP charging functions
RfApplication	Offline charging application
RfSession	Offline charging session

The `RfApplication` class can be used to directly send ACR requests for event-based charging. The application also has the option of directly modifying the ACR request before it is sent out. This is necessary in order for an application to add any custom AVPs to the request.

In particular, an application must set the Service-Information AVP it carries the service-specific parameters for the CDF. The Service-Information AVP of the ACR request is used to send the application-specific charging service information from the CTF (WLSS) to the CDF (Charging Server). This is a grouped AVP whose value depends on the application and its charging function. The Offline Charging API allows the application to set this information on the request before it is sent out.

For session-based accounting, the `RfApplication` class can also be used to create new accounting sessions for generating session-based charging events. Each accounting session is represented by an instance of `RfSession`, which encapsulates the accounting state machine for the session.

Accessing the Rf Application

If the Rf application is deployed, then applications deployed on Converged Application Server can obtain an instance of the application from the Diameter node (`com.bea.wcp.diameter.Node` class). Example 3–4 shows the sample Servlet code used to obtain the Diameter `Node` and access the Rf application.

Example 3–4 Accessing the Rf Application

```
ServletContext sc = getServletConfig().getServletContext();
Node node = sc.getAttribute("com.bea.wcp.diameter.Node");
RfApplication rfApp = (RfApplication)
node.getApplication(Charging.RF_APPLICATION_ID);
```

Applications can safely use a single instance of `RfApplication` to issue offline charging requests concurrently, in multiple threads. Each instance of `RfSession` actually holds the per-session state unique to each call.

Implementing Session-Based Charging

For session-based charging requests, an application first uses the `RfApplication` to create an instance of `RfSession`. The application can then use the session object to create one or more charging requests.

The first charging request must be an ACR START request, followed by zero or more ACR INTERIM requests. The session ends with an ACR STOP request. Upon receipt of the corresponding ACA STOP message, the `RfApplication` automatically terminates the `RfSession`.

Example 3–5 shows the sample code used to start a new session-based accounting session.

Example 3–5 Starting a Session-Based Account Session

```
RfSession session = rfApp.createSession();
```

```
ACR acr = session.createACR(RecordType.START);
acr.addAvp(Charging.SERVICE_INFORMATION, ...);
ACA aca = acr.sendAndWait(1000);
sipRequest.getApplicationSession().setAttribute("RfSession", session);
if (!aca.getResultCode().isSuccess()) {
  ... error ...
}
```

In Example 3–5, the `RfSession` is stored as a SIP application session attribute so that it can be used to send additional accounting requests as the call progresses. Example 3–6 shows how to send an INTERIM request.

Example 3–6 Sending an INTERIM request

```
RfSession session = (RfSession)
req.getApplicationSession().getAttribute("RfSession");
  ACR acr = session.createACR(RecordType.INTERIM);
  ACA aca = acr.sendAndWait(1000);
  if (!aca.getResultCode().isSuccess()) {
    // Handle the error...
  }
```

An application may want to send one or more ACR INTERIM requests while a call is in progress. The frequency of ACR INTERIM requests is usually based on the Acct-Interim-Interval AVP value in the ACA START message sent by the CDF. For this reason, an application timer must be used to send ACR INTERIM requests at the requested interval. See 3GPP TS 32.299 for more details about interim requests.

Specifying the Session Expiration

The Acct-Interim-Interval (AII) timer value is used to indicate the expiration time of an Rf accounting session. It is specified when ACR START is sent to the CDF to initiate the accounting session. The CDF responds with its own AII value, which must be used by the CTF to start a timer upon whose expiration an ACR INTERIM message must be sent. This INTERIM message informs the CDF that the session is still in use. Otherwise, the CDF terminates the session automatically.

It is the application's responsibility to send ACR INTERIM messages, because these are used to send updated Service-Information data to the CDF. Oracle recommends creating a ServletTimer that is set to expire according to the AII value. When the timer expires, the application must send an ACR INTERIM message with the updated service information data.

Sending Asynchronous Events

Applications generally use the synchronous `sendAndWait()` method. However, if latency is critical, an asynchronous API is provided wherein the application Servlet is asynchronously notified when an answer message is received from the CDF. To use the asynchronous API, an application first registers an instance of `SessionListener` in order to asynchronously receive messages delivered to the session, as shown in Example 3–7.

Example 3–7 Registering a SessionListener

```
RfSession session = rfApp.createSession();
session.setAttribute("SAS", sipReq.getApplicationSession());
session.setListener(this);
```

Attributes can be stored in an `RfSession` instance similar to the way SIP application session attributes are stored. In the above example, the associated SIP application was stored as an `RfSession` so that it is available to the listener callback.

When a Diameter request or answer message is received from the CDF, the application Servlet is notified by calling the `rcvMessage(Message msg)` method. The associated SIP application session can then be retrieved from the `RfSession` if it was stored as a session attribute, as shown in Example 3–8.

Example 3–8 Retrieving the RfSession after a Notification

```
public void rcvMessage(Message msg) {
  if (msg.getCommand() != Command.ACA) {
    if (msg.isRequest()) {
      ((Request) msg).createAnswer(ResultCode.UNABLE_TO_COMPLY, "Unexpected
                    request").send();
    }
    return;
  }
  ACA aca = (ACA) msg;
  RfSession session = (RfSession) aca.getSession();
  SipApplicationSession appSession = (SipApplicationSession)
              session.getAttribute("SAS");
    ...
}
```

Implementing Event-Based Charging

For an event-based charging request, the charging request is a one-time event and the session is automatically terminated upon receipt of the corresponding EVENT ACA message. The `sendAndWait(long timeout)` method can be used to synchronously send the EVENT request and block the thread until a response has been received from the CDF. Example 3–9 shows an example that uses an `RfSession` for sending an event-based charging request.

Example 3–9 Event-Based Charging Using RfSession

```
RfSession session = rfApp.createSession();
ACR acr = session.createACR(RecordType.EVENT);
acr.addAvp(Charging.SERVICE_INFORMATION, ...);
ACA aca = acr.sendAndWait(1000);
if (!aca.getResultCode().isSuccess()) {
  // Send error response ...
}
```

For convenience, it is also possible send event-based charging requests using the `RfApplication` directly, as shown in Example 3–10.

Example 3–10 Event-Based Charging Using RfApplication

```
ACR acr = rfApp.createEventACR();
acr.addAvp(Charging.SERVICE_INFORMATION, ...);
ACA aca = acr.sendAndWait(1000);
```

Internally, the `RfApplication` creates an instance of `RfSession` associated with the ACR request, so this method is equivalent to creating the session explicitly.

For both session and event based accounting, the `RfSession` class automatically handles creating session IDs, as well as updating the Accounting-Record-Number AVP used to sequence messages within the same accounting session.

In the above cases the applications waits for up to 1000 ms to receive an answer from the CDF. If no answer is received within that time, the Diameter core delivers an UNABLE_TO_COMPLY error response to the application, and cancels the request. If no timeout is specified with `sendAndWait()`, then the default request timeout of 30 seconds is used. This default value can be configured using the Diameter console extension.

Using the Accounting Session State

The accounting session state for offline charging is serializable, so it can be stored as a SIP application session attribute. Because the client APIs are synchronous, it is not necessary to maintain any state for the accounting session once the Servlet has finished handling the call.

For event-based charging events it is not necessary for the application to maintain any accounting session state because it is only used internally, and is disposed once the ACA response has been received.

4

Using the Diameter Ro Interface API for Online Charging

This chapter describes how to use the Diameter Ro interface API, based on Oracle Communications Converged Application Server's Diameter protocol implementation, in your own applications.

Overview of Ro Interface Support

Online charging, also known as credit-based charging, is used to charge prepaid services. A typical example of a prepaid service is a calling card purchased for voice or video. The Ro protocol allows a Charging Trigger Function (CTF) to issue charging events to an Online Charging Function (OCF). The charging events can be immediate, event-based, or session-based.

Converged Application Server provides a Diameter Online Charging Application that deployed applications can use to generate charging events based on the Ro protocol. This enables deployed applications to act as CTF to a configured OCF. The Diameter Online Charging Application uses the base Diameter protocol that underpins both the Rf and Sh applications.

The Diameter Online Charging Application is based on IETF RFC 4006: Diameter Credit Control Application (http://www.ietf.org/rfc/rfc4006.txt). However, the application supports only a subset of the RFC 4006 required for compliance with 3GPP TS 32.299: Telecommunication management; Charging management; Diameter charging applications (http://www.3gpp.org/ftp/Specs/html-info/32299.htm). Specifically, the Converged Application Server Diameter Online Charging Application provides no direct support for service-specific Attribute-Value Pairs (AVPs), but the API that is provided is flexible enough to allow applications to include custom service-specific AVPs in any credit control request.

Understanding Credit Authorization Models

RFC 4006 defines two basic types of credit authorization models:

- Credit authorization with unit reservation.
- Credit authorization with direct debiting.

Credit authorization with unit reservation can be performed with either event-based or session-based charging events. Credit authorization with direct debiting uses immediate charging events. In both models, the CTF requests credit authorization from the OCF prior to delivering services to the end user.

The sections that follow describe each model in more detail.

Credit Authorization with Unit Determination

RFC 4006 defines both Event Charging with Unit Reservation (ECUR) and Session Charging with Unit Reservation (SCUR). Both charging events are session-based, and require multiple transactions between the CTF and OCF. ECUR begins with an interrogation to reserve units before delivering services, followed by an additional interrogation to report the actual used units to the OCF upon service termination. With SCUR, it is also possible to include one or more intermediate interrogations for the CTF in order to report currently-used units, and to reserve additional units if required. In both cases, the session state is maintained in both the CTF and OCF.

For both ECUR and SCUR, the online charging client implements the "CLIENT, SESSION BASED" state machine described in RFC 4006.

Credit Authorization with Direct Debiting

For direct debiting, Immediate Event Charging (IEC) is used. With IEC, a single transaction is created where the OCF deducts a specific amount from the user's account immediately after completing the credit authorization. After receiving the authorization, the CTF delivers services. This form of credit authorization is a one-time event in which no session state is maintained.

With IEC, the online charging client implements the "CLIENT, EVENT BASED" state machine described in IETF RFC 4006.

Determining Units and Rating

Unit determination refers to calculating the number of non-monetary units (service units, time, events) that can be assigned prior to delivering services. Unit rating refers to determining a price based on the non-monetary units calculated by the unit determination function.

It is possible for either the OCF or the CTF to handle unit determination and unit rating. The decision lies with the client application, which controls the selection of AVPs in the credit control request sent to the OCF.

Configuring the Ro Application

The `RoApplication` is packaged as a Diameter application similar to the Sh application used for managing profile data. The Ro Diameter application can be configured and enabled by editing the Diameter configuration file located in `Domain_Home/config/custom/diameter.xml`, or by using the Diameter console extension.

The application init parameter `ocs.host` specifies the host identity of the OCF. The OCF host must also be configured in the peer table as part of the global Diameter configuration. Alternately, the init parameter `ocs.realm` can be used to specify more than one OCF host using realm-based routing. The corresponding realm definition must also exist in the global Diameter configuration.

Example 4–1 shows a sample excerpt from `diameter.xml` that enables Ro with an OCF host name of "myocs.example.com."

Example 4–1 Sample Ro Application Configuration (diameter.xml)

```
<application>
  <application-id>4</application-id>
  <class-name>com.bea.wcp.diameter.charging.RoApplication</class-name>
  <param>
```

```
      <name>ocs.host</name>
      <value>myocs.example.com</value>
    </param>
  </application>
```

Because the `RoApplication` is based on the Diameter Credit Control Application, its Diameter application id is 4.

Overview of the Online Charging API

Converged Application Server provides an online charging API to enable any deployed application to act as a CTF and issue online charging events to an OCF through the Ro protocol. All online charging requests use the Diameter Credit-Control-Request (CCR) message. The CC-Request-Type AVP is used to indicate the type of charging used. In the charging API, the CC-Request-Type is represented by the `RequestType` class in package `com.bea.wcp.diameter.cc`. Table 4–1 shows the request types associated with different credit authorization models.

Table 4–1 Credit Control Request Types

Type	Description	RequestType Field in com.bea.wcp.diameter.cc.RequestType
IEC	Immediate Event Charging	`EVENT_REQUEST`
ECUR	Event Charging with Unit Reservation	`INITIAL` or `TERMINATION_REQUEST`
SCUR	Session Charging with Unit Reservation	`INITIAL`, `UPDATE`, or `TERMINATION_REQUEST`

For ECUR and SCUR, units are reserved prior to service delivery and committed upon service completion. Units are reserved with `INITIAL_REQUEST` and committed with a `TERMINATION_REQUEST`. For SCUR, units can also be updated with `UPDATE_REQUEST`.

The base diameter package, `com.bea.wcp.diameter`, contains classes to support the re-authorization requests used in Ro. The `com.bea.wcp.diameter.cc` package contains classes to support credit-control applications, including Ro applications. `com.bea.wcp.diameter.charging` directly supports the Ro credit-control application. Table 4–2 summarizes the classes of interest to Ro credit-control.

Table 4–2 Summary of Ro Classes

Class	Description	Package
`Charging`	Constant definitions	`com.bea.wcp.diameter.charging`
`RoApplication`	Online charging application	`com.bea.wcp.diameter.charging`
`RoSession`	Online charging session	`com.bea.wcp.diameter.charging`
`CCR`	Credit Control Request	`com.bea.wcp.diameter.cc`
`CCA`	Credit Control Answer	`com.bea.wcp.diameter.cc`
`ClientSession`	Credit control client session	`com.bea.wcp.diameter.cc`
`RequestType`	Credit-control request type	`com.bea.wcp.diameter.cc`
`RAR`	Re-Auth-Request message	`com.bea.wcp.diameter`

Table 4–2 (Cont.) Summary of Ro Classes

Class	Description	Package
RAA	Re-Auth-Answer message	com.bea.wcp.diameter

Accessing the Ro Application

If the Ro application is deployed, then applications deployed on Converged Application Server can obtain an instance of the application from the Diameter node (com.bea.wcp.diameter.Node class). Example 4–2 shows the sample Servlet code used to obtain the Diameter Node and access the Ro application.

Example 4–2 Accessing the Ro Application

```
private RoApplication roApp;
void init(ServletConfig conf) {
  ServletContext ctx = conf.getServletContext();
  Node node = (Node) ctx.getParameter("com.bea.wcp.diameter.Node");
  roApp = node.getApplication(Charging.RO_APPLICATION_ID);
}
```

This code example would make RoApplication available to the Servlet as an instance variable. The instance of RoApplication is safe for use by multiple concurrent threads.

Implementing Session-Based Charging

The RoApplication can be used to create new sessions for session-based credit authorization. The RoSession class implements the appropriate state machine depending on the credit control type, either ECUR (Event-Based Charging with Unit Reservation) or SCUR (Session-based Charging with Unit Reservation). The RoSession class is also serializable, so it can be stored as a SIP session attribute. This allows the session to be restored when necessary to terminate the session or update credit authorization.

The example in Example 4–3 creates a new RoSession for event-based charging, and sends a CCR request to start the first interrogation. The RoSession instance is saved so that it can be terminated later, after the service has finished.

Note that the RoSession class automatically handles creating session IDs; the application is not required to set the session ID.

Example 4–3 Creating and Using a RoSession

```
RoSession session = roApp.createSession();
CCR ccr = session.createCCR(RequestType.INITIAL);
CCA cca = ccr.sendAndWait();
sipAppSession.setAttribute("RoSession", session);
...
```

Handling Re-Auth-Request Messages

The OCS may initiate credit re-authorization by issuing a Re-Auth-Request (RAR) to the CTF. The application can register a session listener for handling this type of request. Upon receiving a RAR, the Diameter subsystem invoke the session listener on the applications corresponding RoSession object. The application must then respond to the OCS with an appropriate RAA message and initiate credit re-authorization to the CTF by sending a CCR with the CC-Request-Type AVP set to the value

UPDATE_REQUEST, as described in section 5.5 of RFC 4006 (http://www.ietf.org/rfc/rfc4006.txt).

A session listener must implement the SessionListener interface and be serializable, or it must be an instance of SipServlet. A Servlet can register a listener as follows:

```
RoSession session = roApp.createSession();
session.addListener(new SessionListener() {
  public void rcvMessage(Message msg) {
    System.out.println("Got message: id = " msg.getSession().getId());
  }
}
```

Example 4–4 shows sample rcvMessage() code for processing a Re-Auth-Request.

Example 4–4 Managing a Re-Auth-Request

```
RoSession session = roApp.createSession();
session.addListener(new SessionListener() {
public void rcvMessage(Message msg) {
  Request req = (Request)msg;
  if (req.getCommand() != Command.RE_AUTH_REQUEST) return;
  RoSession session = (RoSession) req.getSession();
  Answer ans = req.createAnswer();
  ans.setResultCode(ResultCode.LIMITED_SUCCESS); // Per RFC 4006 5.5
  ans.send();
  CCR ccr = session.createCCR(Ro.UPDATE_REQUEST);
    // Set CCR AVPs according to requested credit re-authorization...
  ccr.send();
  CCA cca = (CCA) ccr.waitForAnswer();
}
```

In Example 4–4, upon receiving the Re-Auth-Request the application sends an RAA with the result code DIAMETER_LIMITED_SUCCESS to indicate to the OCS that an additional CCR request is required in order to complete the procedure. The CCR is then sent to initiate credit re-authorization.

Note: Because the Diameter subsystem locks the call state before delivering the request to the corresponding RoSession, the call state remains locked while the handler processes the request.

Sending Credit-Control-Request Messages

The CCR class represents a Diameter Credit-Control-Request message, and can be used to send credit control requests to the OCF. For both ECUR (Event-Based Charging with Unit Reservation) and SCUR (Session-Based Charging with Unit Reservation), an instance of RoSession is used to create new CCR requests. You can also use RoApplication directly to create CCR messages for IEC (Immediate Event Charging). Example 4–5 shows an example of how to create and send a CCR.

Example 4–5 Creating and Sending a CCR

```
CCR ccr = session.createCCR(RequestType.INITIAL);
ccr.setServiceContextId("sample_id");
CCA cca = ccr.sendAndWait();
```

Once a CCR request is created, you can set whatever application- or service-specific AVPs that are required before sending the request using the addAvp() method. Because

some of the same AVPs need to be included in each new request for the session, it is also possible to set these AVPs on the session itself. Example 4–6 shows a sample that sets:

- Subscription-Id to identify the user for the session
- Service-Identifier to indicate the service requested
- Requested-Service-Unit to specify the units requested.

A custom AVP is also added directly to the CCR request.

Example 4–6 Setting AVPs in the CCR

```
session.setSubscriptionId(...);
session.setServiceIdentifier(...);
CCR ccr = session.createCCR(RequestType.INITIAL);
ccr.setRequestedServiceUnit(...);
ccr.addAvp(CUSTOM_MESSAGE, "This is a test");
ccr.send();
```

In this case, the same Subscription-Id and Service-Identifier are added to every new request for the session. The custom AVP "Custom-Message" is added to the message before it is sent out.

Handling Failures

Applications can examine the Result-Code AVP in CCA error responses from the OCF to detect the cause of a failure and take an appropriate action. Locally-generated errors, such as an unavailable peer or invalid route specification, cause the request send method to throw an `IOException` to with a detailed message indicating the nature of the failure.

Applications can also use the Diameter Timer Tx value for determining when the OCF fails to respond to a credit authorization request. Timer Tx has a default value of 10 seconds, but can be overridden using the `tx.timer` init parameter in the `RoApplication` configuration. Timer Tx starts when a CCR is sent to the OCF. The timer resets after the corresponding CCA is received.

If Tx expires before a corresponding CCA arrives, any call to `waitForAnswer` immediately returns null to indicate that the request has timed out. An application can then take action according to the value of the Credit-Control-Failure-Handling (CCFH) AVP in the request. See section 5.7, "Failure Procedures" in RFC 4006 (`http://www.ietf.org/rfc/rfc4006.txt`) for more details.

Example 4–7 terminates the credit control session if timer Tx expires before receiving the CCA. If the CCA is received later by the Diameter subsystem, the message is ignored because the session no longer exists.

Example 4–7 Checking for Timer Tx Expiry

```
CCR ccr = session.createCCR(RequestType.INITIAL);
ccr.setCreditControlFailureHandling(RequestType.TERMINATION);
ccr.send();
CCA cca = ccr.waitForAnswer();
if (cca == null) {
  session.terminate();
}
```