

Oracle® Communications Services Gatekeeper

Platform Test Environment

Release 4.1

September 2009

Copyright © 2007, 2008, 2009, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this software or related documentation is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this software in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of this software. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software in dangerous applications.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

This software and documentation may provide access to or information on content, products and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

1. Installing and Running the Platform Test Environment

| | |
|--|-----|
| Overview | 2-1 |
| Installing and Running the Platform Test Environment | 2-3 |

2. Navigating the Platform Test Environment GUI

| | |
|----------------------------------|------|
| The Tools Panel. | 3-2 |
| The Tool Selector Panel. | 3-2 |
| The Tool Action Panel | 3-3 |
| The Server Tool | 3-4 |
| The Database Tool | 3-7 |
| The Clients Tool | 3-8 |
| The Tests Tool. | 3-13 |
| The Simulator Panel | 3-17 |
| The SLA Editor. | 3-20 |
| The Budget Monitor | 3-24 |

3. Extending the Platform Test Environment

| | |
|---|-----|
| Extending the Platform Test Environment | 4-1 |
| The Stateful SPI | 4-2 |
| The Stateless SPI | 4-4 |
| The Custom Base SPI | 4-5 |
| The Custom Results Provider SPI | 4-7 |
| The Custom Statistics Provider SPI. | 4-8 |

| | |
|--------------------------------------|------|
| The Context API | 4-9 |
| The Module.xml Descriptor File | 4-11 |

4. Using the Unit Test Framework with the Platform Test Environment

| | |
|--|-----|
| Creating and Running the Unit Test | 5-1 |
| The Example Unit Test | 5-3 |

Installing and Running the Platform Test Environment

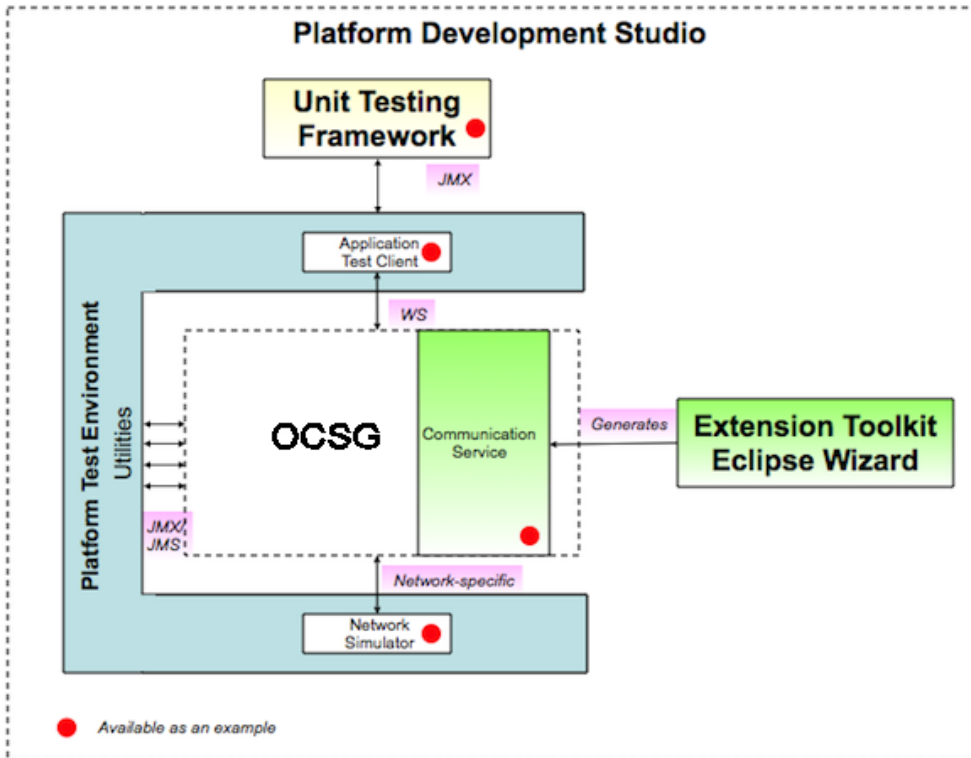
Oracle Communications Services Gatekeeper provides an entire suite of testing tools to help you develop your extensions quickly and efficiently. This chapter introduces the Platform Test Environment and describes installing and running it. It consists of:

- [Overview](#)
- [Installing and Running the Platform Test Environment](#)

Overview

The Platform Test Environment is a key part of the Platform Development Studio.

Figure 1-1 The Platform Test Environment in Context



The Platform Test Environment is a flexible, powerful tool, consisting of:

- Application service test clients for most out-of-the-box communication services, using both the Web Services facade and the RESTful facade
- PRM test clients for many operations covered by the Partner Relationship Management interfaces
- Network simulators for most node types supported by the out-of-the-box communication services
- A billing system simulator for use with the Diameter simulator
- A real-time graphing budget monitor

- Dual mode support:
 - Standalone with a Java Swing-based GUI
 - Console, in which the PTE's functionality can be accessed using JMX, as, for example, from a unit test
- MBean browser for performing Gatekeeper management tasks
- Log browser for checking server logs
- A JMS-based EDR/CDR/Alarm listener
- JNDI browser
- Database browser for interacting with the database
- Real-time duration test graphing
- SLA editor
- Embedded TCP Monitor
- Easily extendable architecture
 - An example application test client for use with the example communication service
 - An example network simulator for use with the example communication service
 - A set of SPIs that allows your modules to interact with the PTE
- A framework for building unit tests, including:
 - A base test class, derived from JUnit
 - Mechanisms that simplify connecting to the Platform Testing Environment
 - An example test case for use with the example communication service

Installing and Running the Platform Test Environment

The Platform Test Environment is automatically installed when you install Oracle Communications Services Gatekeeper. In standard installations, it is found in the `<bea_home>/wlng_pds400/pte` directory. Before you use the PTE, you must have:

- Installed Oracle Communications Services Gatekeeper

Installing and Running the Platform Test Environment

- Used the `setWLSEnv` script in `<bea_home>/wlserver_10.3/server/bin` or set the equivalent path so that you have access to the Ant 1.6.5 installation that comes with WLS.

To start the PTE in GUI mode, type `'ant run'` in a command window in the PTE directory.

To start the PTE in console mode, type `'ant console'` in a command window in the PTE directory.

WARNING: Compatibility between the settings of this version of the PTE and any future versions is *not* guaranteed.

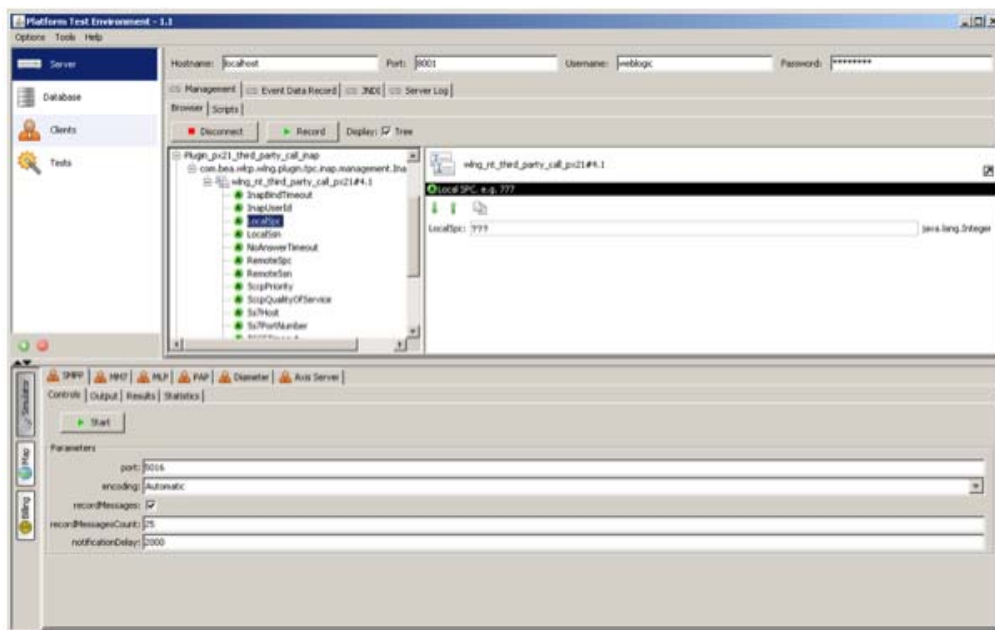
Navigating the Platform Test Environment GUI

This chapter describes the PTE's Java Swing-based GUI. It provides an easy to use access point to the many parts of the Platform Test Environment. Any changes made to the GUI are saved on exit. The GUI consists, broadly, of:

- [The Tools Panel](#), the upper panel
- [The Simulator Panel](#), the lower panel

The chapter also covers [The SLA Editor](#) and [The Budget Monitor](#), general purpose tools included in the PTE.

Figure 2-1 The Main PTE GUI



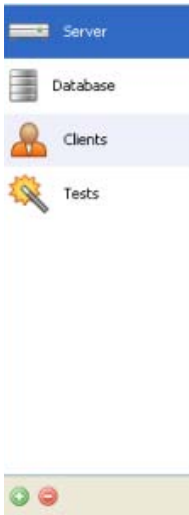
The Tools Panel

The Tools Panel is divided into two main sections:

- [The Tool Selector Panel](#)
- [The Tool Action Panel](#)

The Tool Selector Panel

Use this panel to select the tool you wish to use:

Figure 2-2 The Tool Selector

Notice the Plus and Minus icons in the lower left. You can use these to create multiple versions of the tools. For example, you could create a server tool to correspond to each server instance you are running.

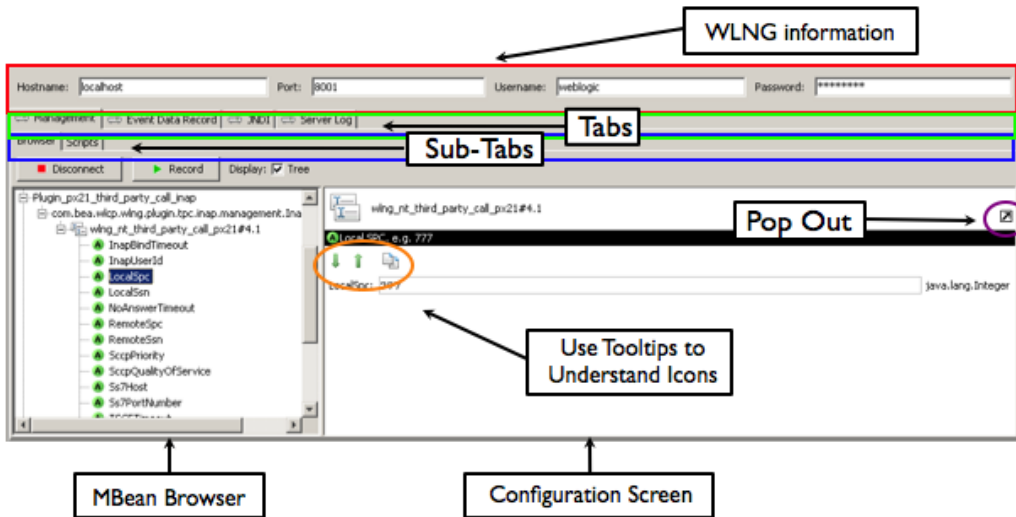
The Tool Action Panel

The Tool Action Panel displays the actions you can do in that particular tool.

- [The Server Tool](#)
- [The Database Tool](#)
- [The Clients Tool](#)
- [The Tests Tool](#)

The Server Tool

Figure 2-3 The Server Tool Action Panel with the Management Tab selected



At the top of the panel, you specify the server with which you wish to interact and your administrative username and password.

Below that are the three main tabs:

- **Management:** This tab lets you perform management tasks on Gatekeeper.
- **Browser:** The sub-tab lets you browse the MBeans on the server you have chosen. You can use this to make changes in the configuration of Gatekeeper instead of opening up the Management Console. Use the **Connect** button to connect to the server. Use your mouse to traverse the MBean list in the left column. Fill in your desired information in the right column. You can use the Pop Out icon on the right to have the same information displayed in a separate pop up window.

Note: In order to run the PTE, you must have set up application and service provider accounts in Oracle Communications Services Gatekeeper, associated them with appropriate groups, and attached SLAs. You must also have done any necessary configuration of the communications services that you will be using and created instances for those communication services that require it. For a detailed description of these and other Oracle Communications Services Gatekeeper

management tasks, see the [System Administration Guide](#) and [Managing Accounts and SLAs](#), separate documents in this set.

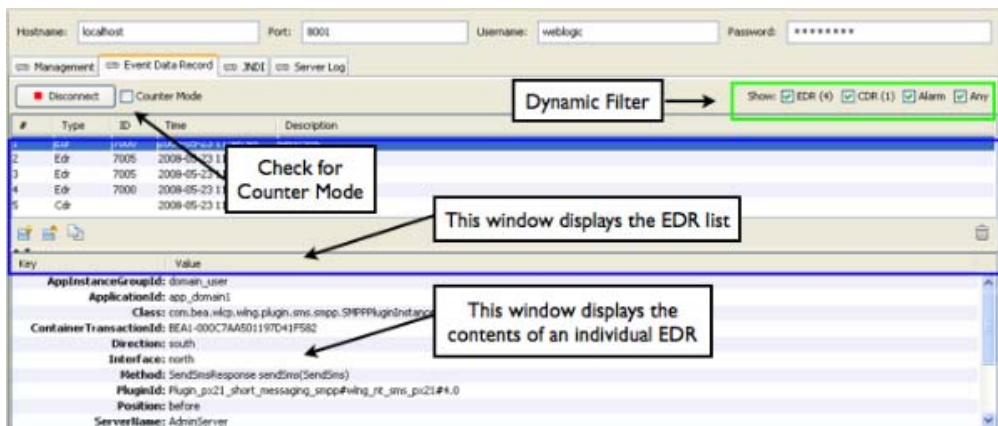
If you want to record a script so that you can automate the management tasks you need to do more than once, use the **Record** button. Once you have begun recording, simply do the tasks you wish to automate. When you are finished, click the **Recording** button to stop. You will get a prompt asking you to name the script. The script will automatically be saved when you close the PTE.

Note: Many windows display convenience functions in the form of icons, as is shown above in the orange circle. Hover your mouse over the icon, and a tooltip explaining its function will appear.

- **Scripts:** This sub-tab displays a list of all the scripts you have recorded in the past. To play a script back, select the name you gave it when you created it, and click the **Run** button.

- **Event Data Record:** Use this tab to monitor EDRs, CDRs, and Alarms

Figure 2-4 The Event Data Record Tab



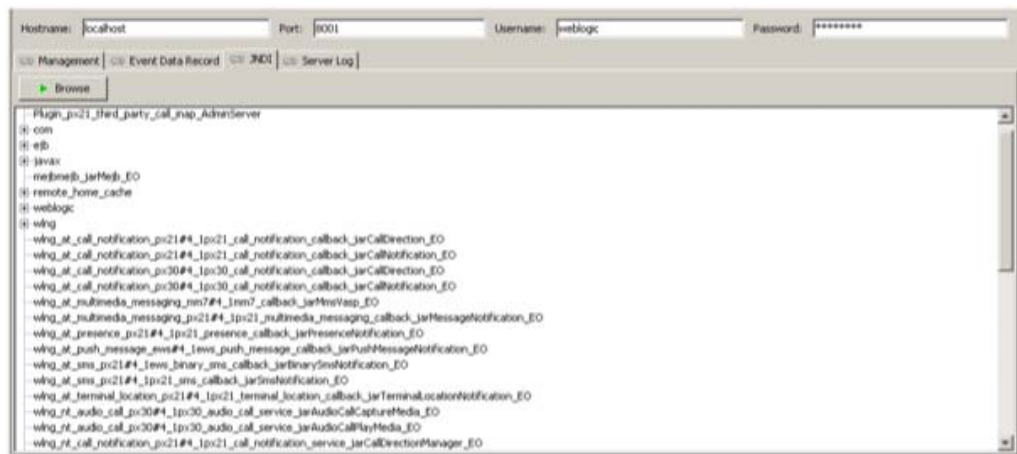
Click **Connect** to establish the JMS connection between the PTE EDR (and CDRs and Alarms) listener and the server instance. Select the sort of records you are interested in receiving using the Dynamic Filter, shown above in the green box. The list of EDRs displays in the window outlined in blue. Selecting a particular EDR in the list causes the contents of that record to be shown in the box outlined in red.

To export, import, or copy to the clipboard a list of EDRs, use the convenience icons in the bottom left of the EDR list window.

To save memory in situations where you are expecting a large number of EDRs, check Counter Mode. This will count the number of records, but will not display the contents.

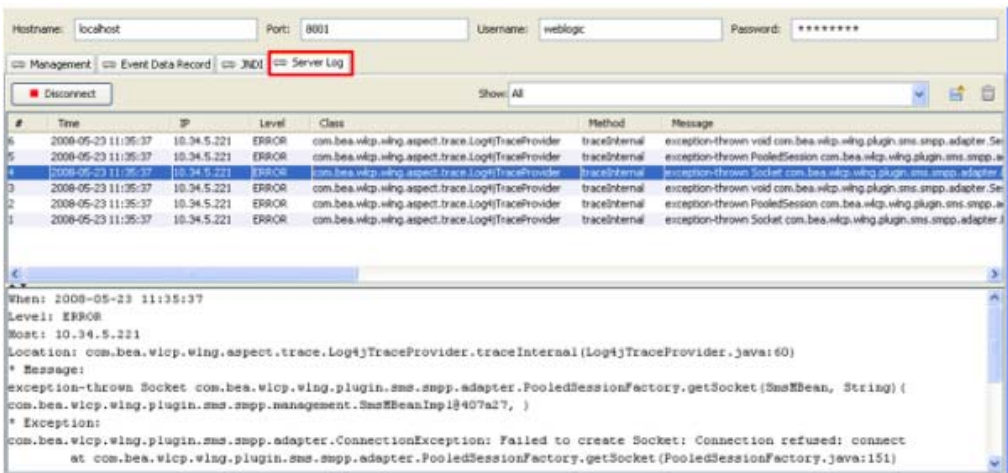
- **JNDI:** Use this tab to browse the JNDI tree.

Figure 2-5 The JNDI Browser Tab



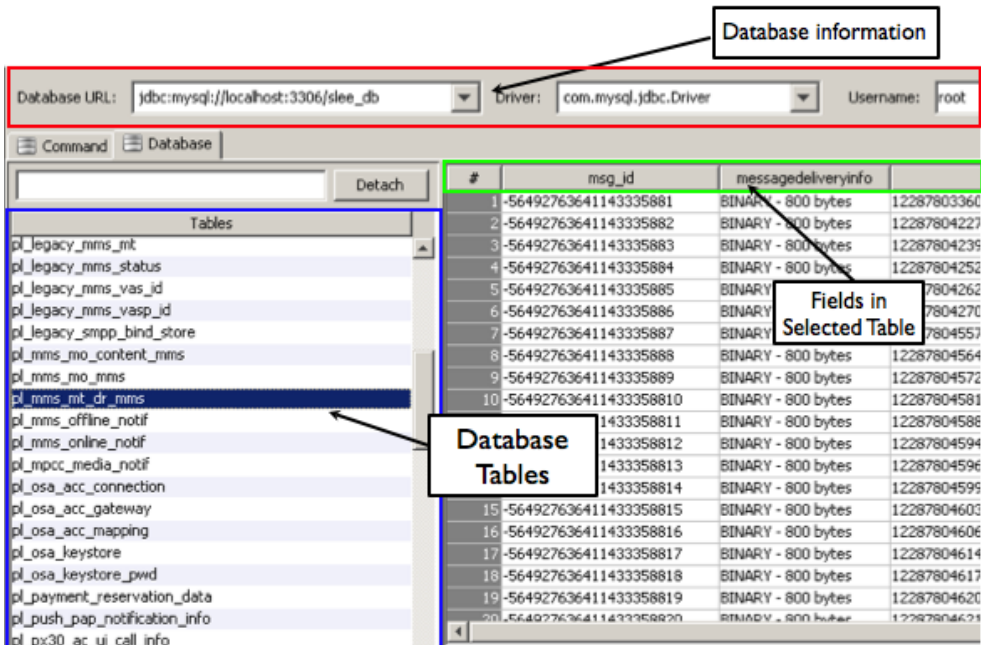
- **Server Log:** Use this tab to browse the logs.

Figure 2-6 Server Log Tab



The Database Tool

Figure 2-7 The Database Tool Action Panel with the Database Tab selected



The Database tool lets you scan your database tables and manipulate them directly.

At the top of the panel you enter your database information, including your database username and password. Click the **Connect** button to connect to the database.

Below that are the two main tabs:

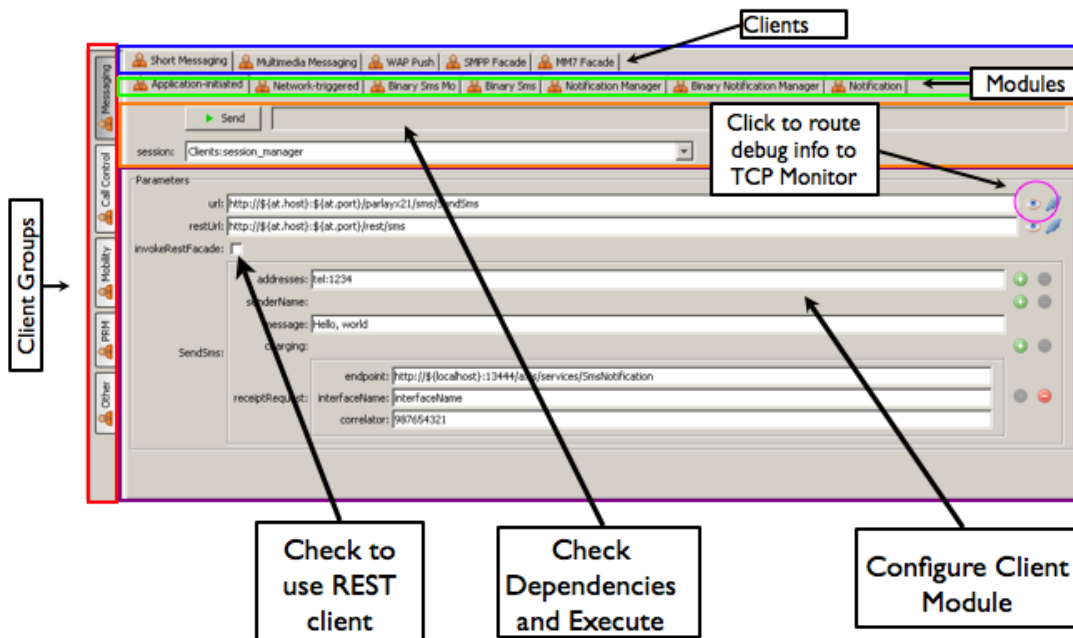
- **Database:** This tab allows you to scan your database tables, and to see the data in them. If there is a table you wish to monitor more closely, enter the name in the dialog box at the top of the left column and click **Detach**. That table then appears individually as an additional tab.
- **Command:** This tab allows you to enter any SQL command you wish directly to the database.

Figure 2-8 The Command Tab



The Clients Tool

Figure 2-9 The Clients Tool Action Panel with the Short Messaging Client Tab selected



The Clients Tool Action Panel is the most complex of the UIs for the Platform Test Environment. The display is divided into three hierarchical groups:

- **Client Groups:** Use the Clients Group column, outlined in red on the left above, to select the functional group of clients you are interested in manipulating. Your choices are:
 - Messaging
 - Clients for Short Messaging, Multimedia Messaging, WAP Push, SMPP, and MM7
 - Call Control
 - Clients for Audio Call, Third Party Call (PX 2.1 and 3.0), and Call Notification
 - Mobility
 - Clients for Terminal Location and Presence
 - PRM
 - Clients for OP and SP Partnership Relationship Management types
 - Other
 - Clients for Session Management, Subscriber Profile, Payment, and the example communication service

Note: There are RESTful versions of many of the clients. To use them instead of the SOAP versions, check the box as shown in [Figure 2-9](#).

- **Clients:** Use the Clients tabs, outlined in blue above, to select the set of clients you are interested in manipulating. In the context of the PTE, a *client* is made up of *modules*. Each module represents one operation belonging to a set of interfaces. So, in the example above, the client that is selected represents the functionality offered by the Parlay X 2.1 Short Messaging set of interfaces.

Note: This particular client also happens to include modules that belong to the Extended Web Services Binary SMS interface. This is because the two sets of interfaces share a common network node and are bundled together in the same.EAR file in Oracle Communications Services Gatekeeper.

- **Modules:** Use the Modules tabs, outlined in green above, to select the module you are interested in manipulating. A module represents a client that executes a single operation from an interface. So, in the example above, the module shown as **Application-initiated**, represents the single operation, `SendSms`. If you wanted to test the operation `SendSmsLogo`, you would need to create an additional module.

Once you have selected the module you wish to use, the display shows two windows. In the lower window, you configure the client module, setting any required parameters. In the upper right corner you will notice a small eye icon. Clicking this will route debug information to the TCP Monitor.

In the upper window, outlined in orange, dependencies are shown. In this case, the operation requires that the client acquire a session ID before sending the command. If you attempt to execute a command and the dependencies have not been set up, the PTE will offer to open the requisite modules for you. See [Figure 2-14](#) below for more information on running sessions.

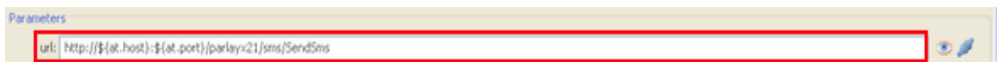
Once the dependencies are taken care of, you can simply click the **Send** button if you wish to execute the operation at that moment.

Note: This assumes you have already completed provisioning your user and associating that user with appropriate groups and SLAS, and have started the simulator to receive the request.

You can also choose to string several tests together into an automated set. See the [The Tests Tool](#) section below for more information.

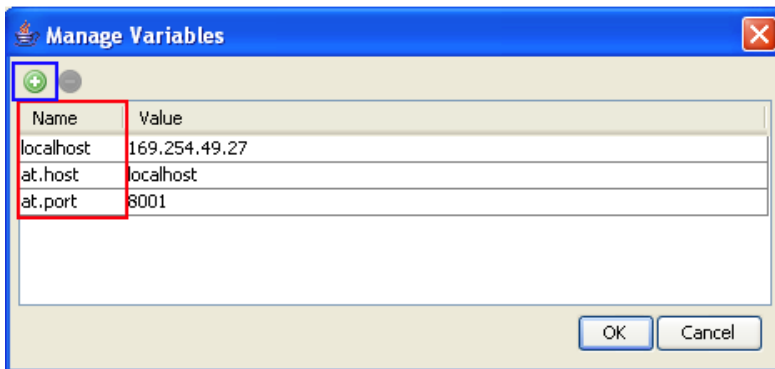
Note: In some module configuration windows, you will see URLs written out with variable values.

Figure 2-10 Variables in URLs



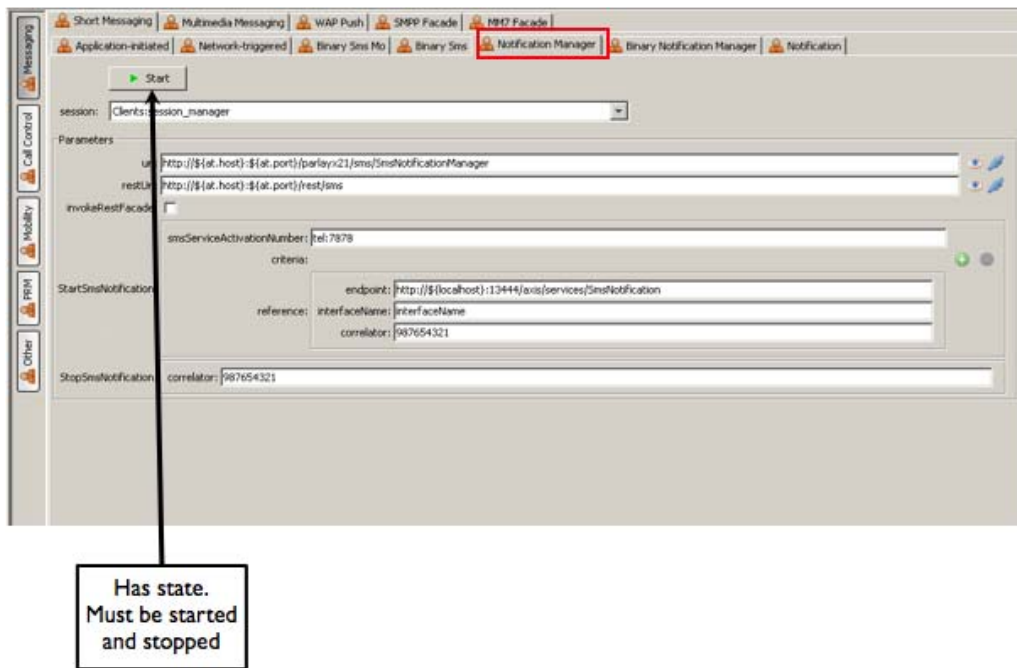
To set those variables, or to add others that are of use to you as you create and run tests, use the **Manage Variables** window. Click **Tools -> Variable Manager**. The default variables are shown outlined in red. To add a variable, click the plus button, outlined in blue. When you are finished editing, click the **OK** button.

Figure 2-11 The Manage Variables Window



Most client modules are stateless. When you click **Send**, the operation is executed and it completes. But some modules have state. They are started, and they run until they are stopped. The **Short Messaging Notification Manager** is such a module. As shown in [Figure 2-12](#), it has a **Start** button, rather than a **Send** button.

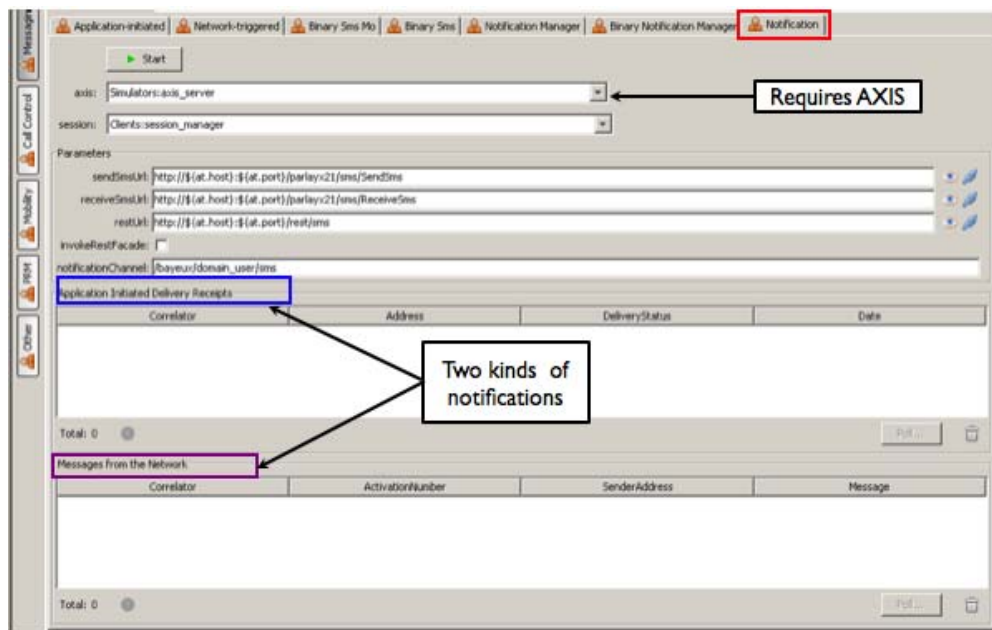
Figure 2-12 A Stateful Module



When you set up notifications, that is, when you tell Gatekeeper that your client is interested in receiving asynchronous messages from the network, the client must provide a web service to which the notifications can be delivered. These client-based web services also show up in the PTE GUI as client modules. In [Figure 2-13](#) below, the **Notifications** tab is selected. This module runs the web service to which both kinds of Short Messaging notifications can be returned: Delivery Receipts for Application-initiated messages and actual SMSs sent from the network to the client application. Notice the dependency on the Axis web server. It must be running for the web service to function. Also note that the RESTful facades use Bayeux protocol channels, running in the OWLS Publish-Subscribe Server instead of SOAP-based Web Services. For more information on the OWLS pub-sub server, see “Using the HTTP Publish-Subscribe Server” in *Developing Web*

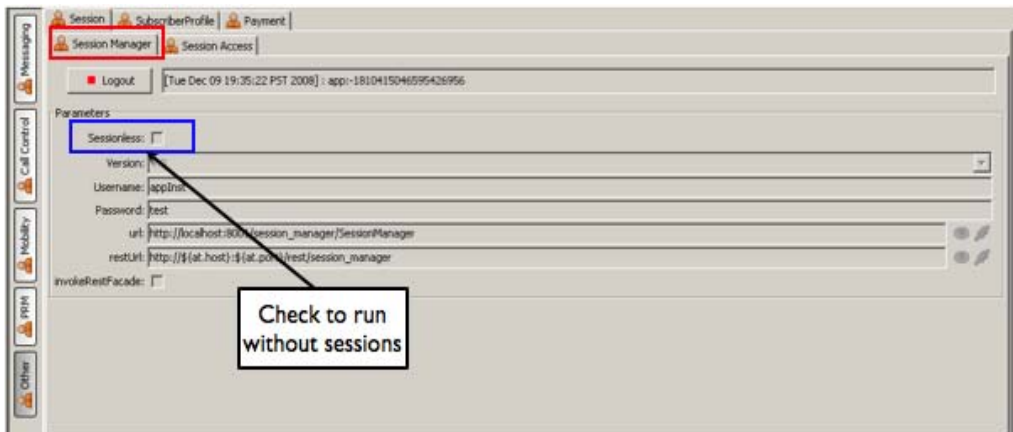
Applications, Servlets, and JSPs For Oracle WebLogic Server at http://download.oracle.com/docs/cd/E12840_01/wls/docs103/webapp/.

Figure 2-13 The Client Notification Web Service



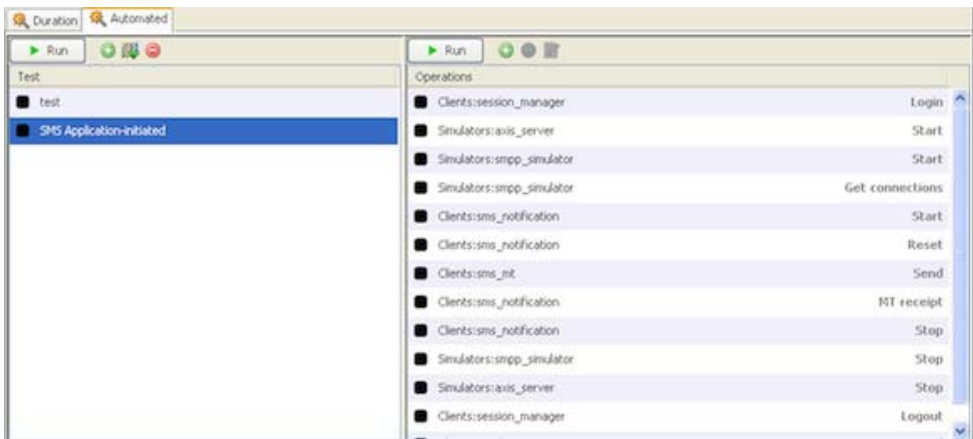
Finally there is the question of session management. The default setting in Gatekeeper is to require applications to start a session and get a Session ID before they send traffic through the system. But this requirement is configurable in Gatekeeper, and so the PTE makes it possible to turn the session requirement on and off. By selecting the Other client group and the Session client, the Session Manager module, you can simply check the Sessionless option, shown below in [Figure 2-14](#), and your clients will not be required to acquire or use a Session ID in order to run traffic.

Figure 2-14 Turning Sessions Off






The Tests Tool

Figure 2-15 Creating an Automated Test Sequence



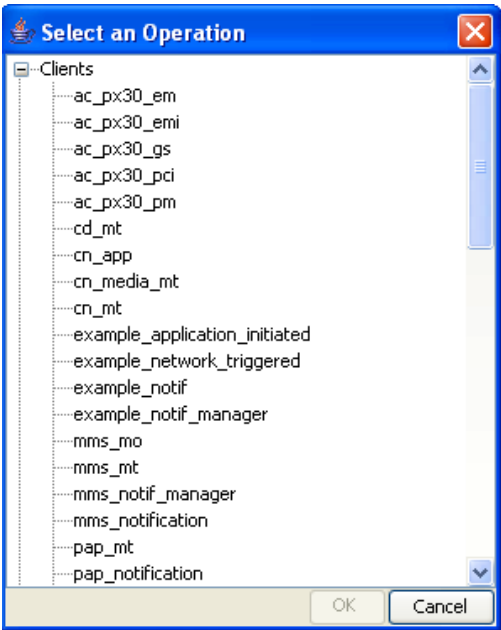
- Automated:** While it is possible to execute a single operation from the client configuration screen, seen above in [Figure 2-9](#), it is usually the case that you will want to string together a set of actions, including setting up client and simulator modules, and executing operations, into a single automated test sequence. The PTE makes this simple to do. To create a test, use the icons at the top of the left column, as shown in [Table 2-1](#):

Table 2-1 Creating tests

| Icon | Function |
|---|--|
|  | Adds a new test. When you click it, you will be prompted to give your test a name. |
|  | Allows you to select from a set of predefined tests. This can be useful for understanding test flows |
|  | Deletes the selected test. |

Once you have created your test, click the plus icon at the top of the right column to add operations. The **Select an Operation** window opens, as in [Figure 2-16](#) below.

Figure 2-16 The Select an Operation Window

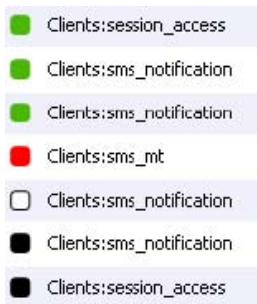


This window shows all operations available in every module in the PTE: clients, simulators, and even duration tests. Select your first operation, and then continue adding until you have completed the desired test sequence.

Note: If you have multiple clients that might be able to perform a particular operation, a popup window will appear and allow you to choose the one you wish to use.

Each test sequence that you create is automatically persisted when the PTE is shut down, so that you only need to create a test once. To run a single test from the GUI, click the **Run** button on the top of the right column. To run the entire test sequence from the GUI, click the **Run** button on the top of the left column. The status of the tests is indicated by the color of the box next to the individual test item names: see [Figure 2-17](#) below.

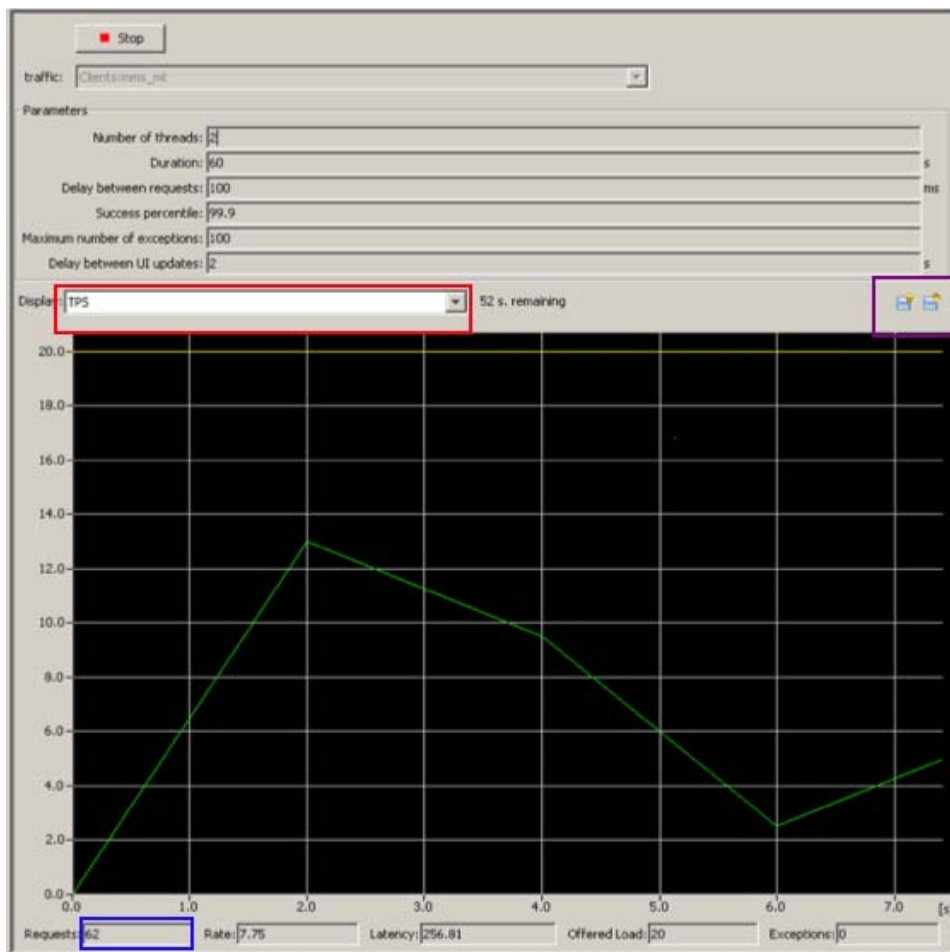
Figure 2-17 A Running Test



Green boxes indicate success; red boxes indicate failure; white boxes indicate an in-progress test; and black boxes indicate tests that have not yet run.

- **Duration:** In addition to functional testing, it is also important to see behavior over time. The PTE also makes it easy to create duration tests and includes a real-time graphing display. See [Figure 2-18](#) below.

Figure 2-18 Duration Tests



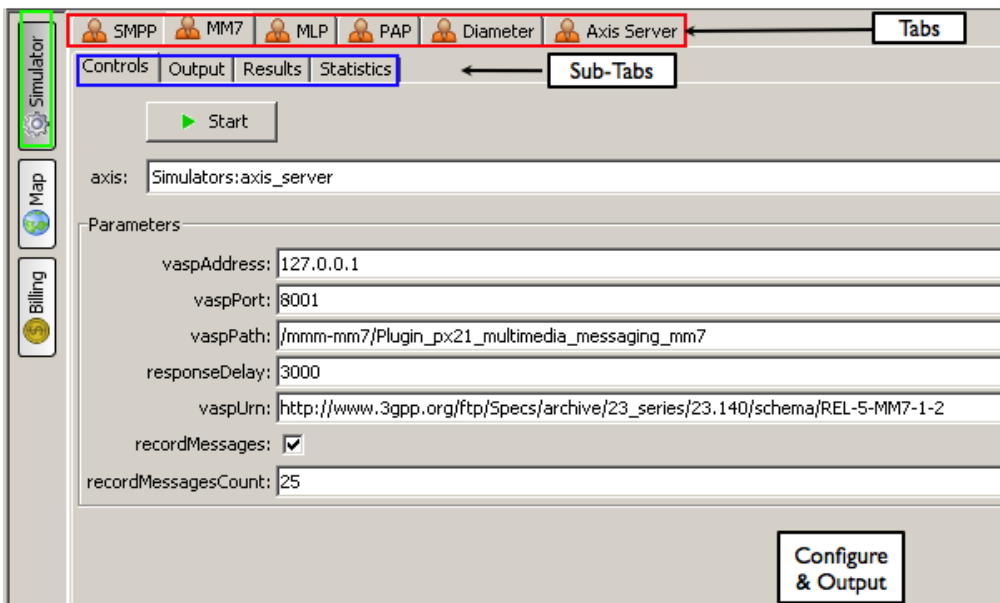
Create a new test by clicking the plus icon in the lower left corner. You are prompted for a name for the test. Configure the test in the upper portion of the right column. Select the type of traffic you wish to run, based on the client type, from the dropdown **Traffic** menu. Select what you wish to see graphed (Transactions Per Second, Exceptions, or Latency) in the **Display** dropdown menu outlined in red in the graphic above. Current statistics appear in the boxes at the bottom of the graph, like the Requests box, outlined in blue.

Make sure the appropriate simulator is running and start the test by clicking the **Start** button. The test runs in the background, so it is possible to run multiple tests in parallel.

Because duration test results are *not* saved across PTE sessions, you can choose to export results to be saved in a file and then import them back into the tool later, using the icons outlined in purple above.

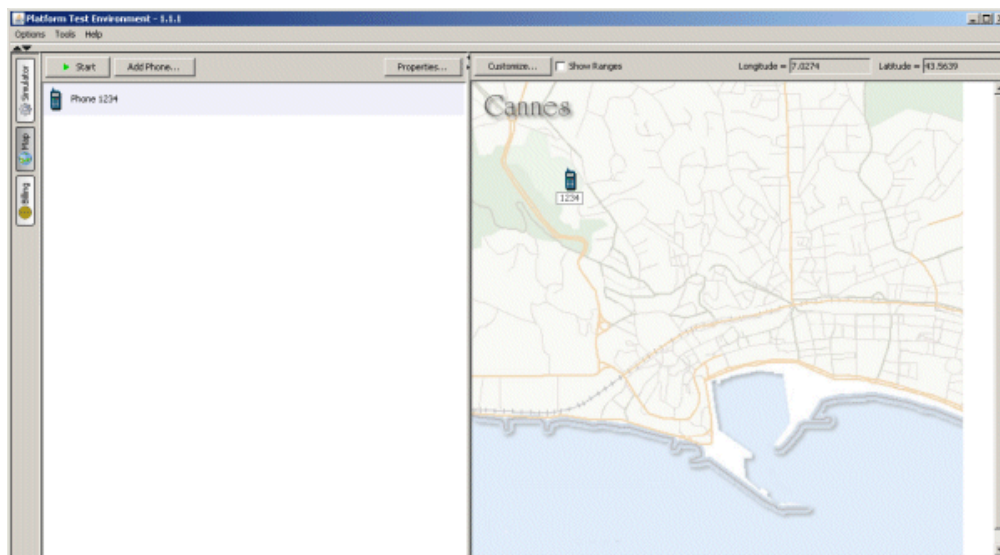
The Simulator Panel

Figure 2-19 The Simulator Panel with MM7 Selected



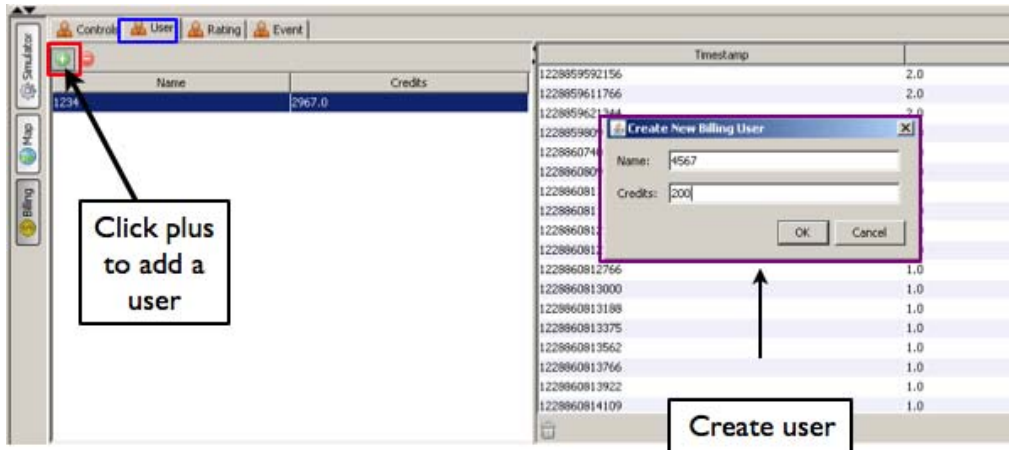
Like the Clients Tool, the Simulator panel is set up as a hierarchy. On the extreme left there are three buttons: **Simulator**, **Map** and **Billing**. Under the **Simulator** button is a set of tabs and sub-tabs. The tabs list the available simulator modules, including a simulator for the example communication service (Netex), and a separate tab and module for the Axis Server, which is required to run traffic over HTTP based protocols like MM7. See Figure 2-19 above. Under the row of tabs is the row of sub-tabs. The number of sub-tabs depends on the module selected. In all cases, there is a **Control** tab in which you can set up any necessary configurations. This area is also where the **Start** button is for each of the modules. The other tabs may allow you to see the actual content of a message or show you the statistics associated with traffic.

Figure 2-20 The Maps Panel



The Maps panel is a variant of a tool which was originally developed as part of the Application Developers SDK. It provides a map on which you can place phone terminals. This offers visual support for testing Parlay X 2.1 SMS, MMS, and Terminal Location traffic. Note the **Customize** button on the bar above the left side of the map. Clicking this button opens a dialog box that allows you to change the map you are using.

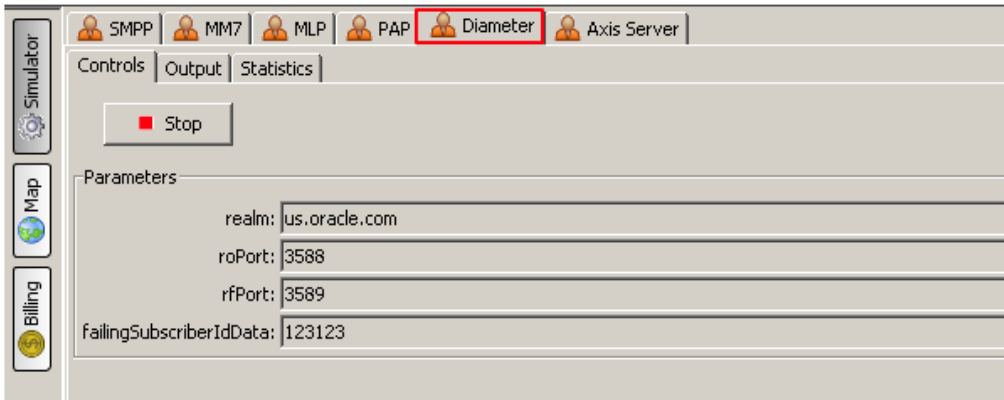
Figure 2-21 The Billing Panel



The Billing panel allows you to set up and run a billing simulator. Oracle Communications Services Gatekeeper can be integrated with billing systems that use the Diameter protocol. So to use the billing simulator with the Payment communication service, for example, you would need to create a user in the billing simulator that corresponds to the `endUserIdentifier` in the Payment client and then send a request from the client through the Diameter simulator to the Billing simulator

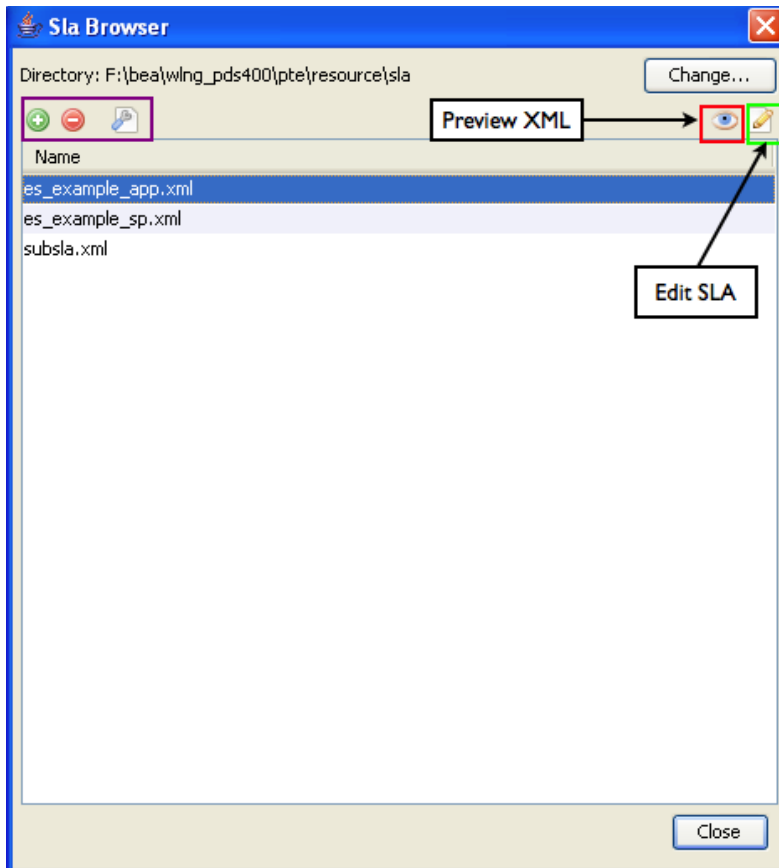
Note: Do not use a value of “localhost” when you are configuring the Diameter Origin Host parameter in the Payment communication service. You must use the actual name of the host.

Figure 2-22 The Diameter Simulator



The SLA Editor

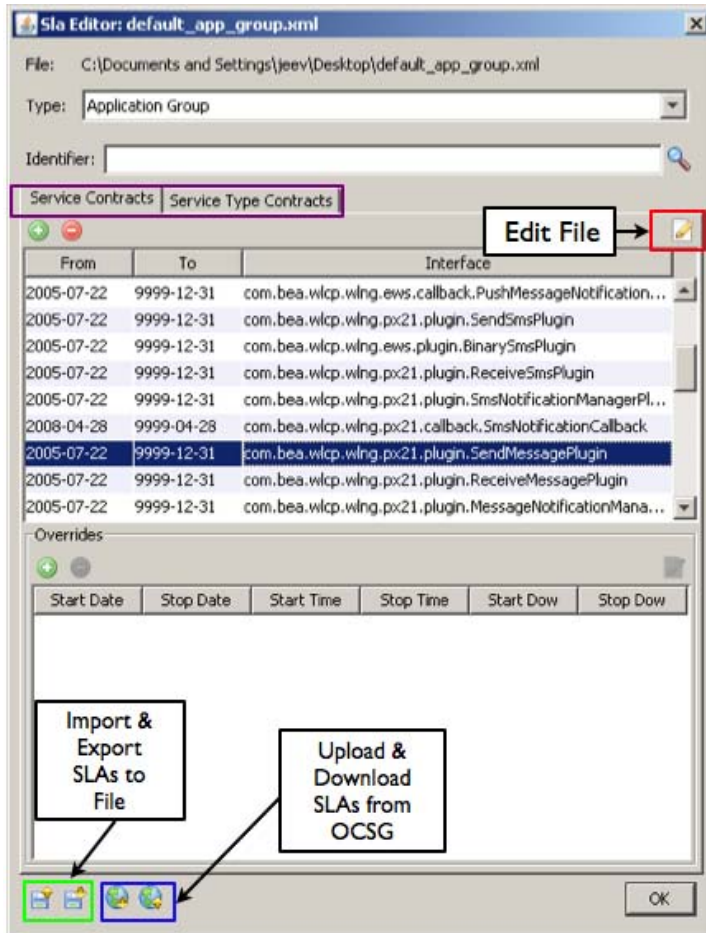
Managing large XML files can be difficult, particularly in a test environment where you may wish to change small details multiple times for various iterations of testing. To help you manage your SLAs, the PTE ships with an SLA editor, which manages the tags and validation so that you can focus on setting appropriate values. To access the SLA editor, first make sure you have selected the Server Tool and are connected to the server. Then click **Tools -> SLA Manager** in the Menu Bar. The SLAs are fetched from the file system and the SLA Browser window opens. See [Figure 2-23](#).

Figure 2-23 The SLA Browser

The SLA files that were fetched from the file system are listed in the main window. By default, the PTE looks in the `pte/resource/sla` directory. This directory has SLAs that model the various options for SLA construction. If you want to search for other files that may be on your system, click the **Change...** button and a file browser appears. To create an entirely new SLA, click the Plus button in the upper left corner, outlined in purple. You can also delete or rename SLAs using buttons in the same area.

To edit an SLA, select the one you are interested in and then click the pencil icon in the upper right corner, outlined in green. The **SLA Editor** window opens. See [Figure 2-24](#).

Figure 2-24 The SLA Editor

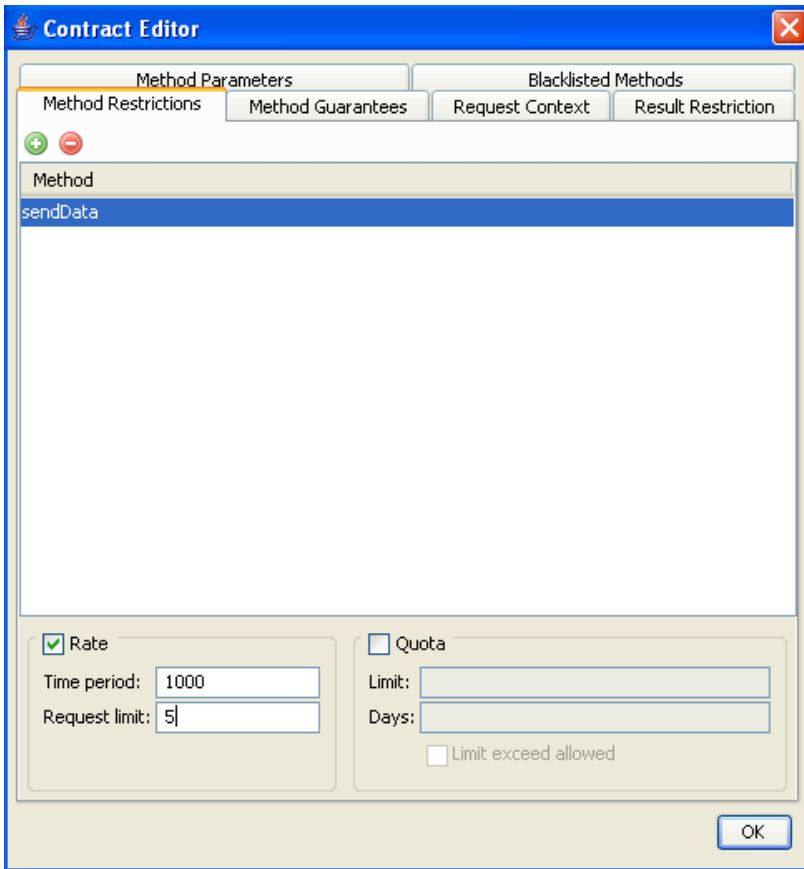


Select the SLA type using the dropdown menu and specify the group identifier. If you wish to import a different SLA from the file system, or to save changes out, use the import and export icons on the bottom left. To upload the SLAs to, or download them from, the repository in the running instance of Oracle Communications Services Gatekeeper, use the icons outlined in blue on the bottom left.

There are three basic kinds of editing you can do - the main **Service Contracts** and **Service Type Contracts** are tabs in the upper window and any **Overrides** you have specified appear in the

lower. To edit a Service Contract, select the item you are interested in and click the pencil icon in the upper right corner. This opens the **Contract Editor**. See [Figure 2-25](#).

Figure 2-25 The Contract Editor

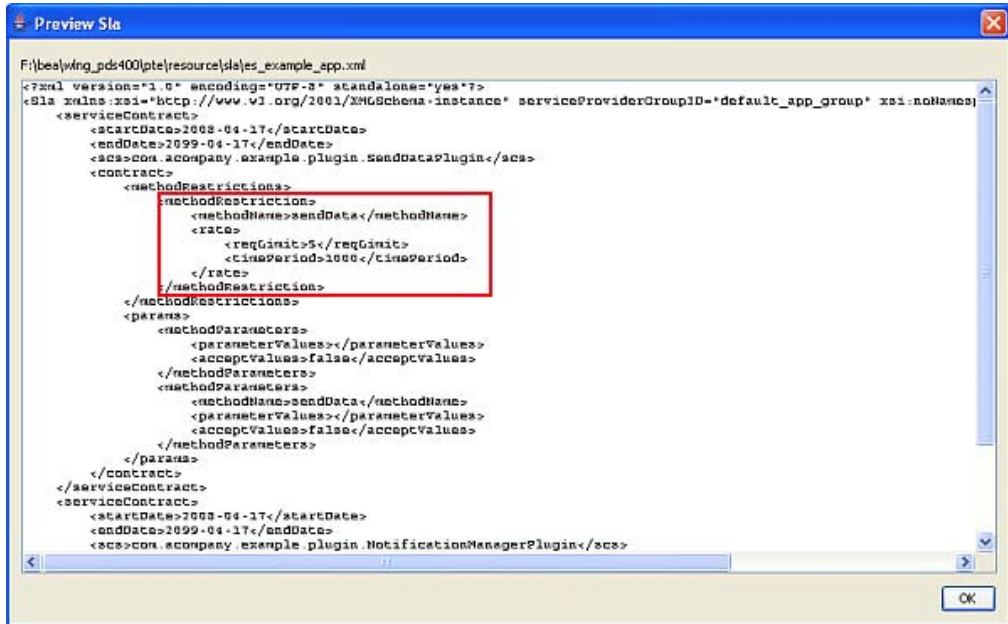


The tags that can be edited appear as tabs at the top of the window. For more information on these tags, see the “[Defining Service Provider Group and Application Group SLAs](#)” chapter in *Managing Accounts and SLAs* a separate document in this set.

In the figure, a rate limit method restriction is being added to the `sendData` operation of the sample communication service. When you have made your edits, click the **OK** button and the window closes. Click **OK** once again (on [Figure 2-24](#)) the window closes. To preview the edits

you have made in XML format, click the Eye icon at the top left of the SLA Browser (Figure 2-23). The **Preview SLA** window opens. See Figure 2-26.

Figure 2-26 The Preview SLA window



The Method Restriction rate limit that was added in Figure 2-25 is shown outlined in red.

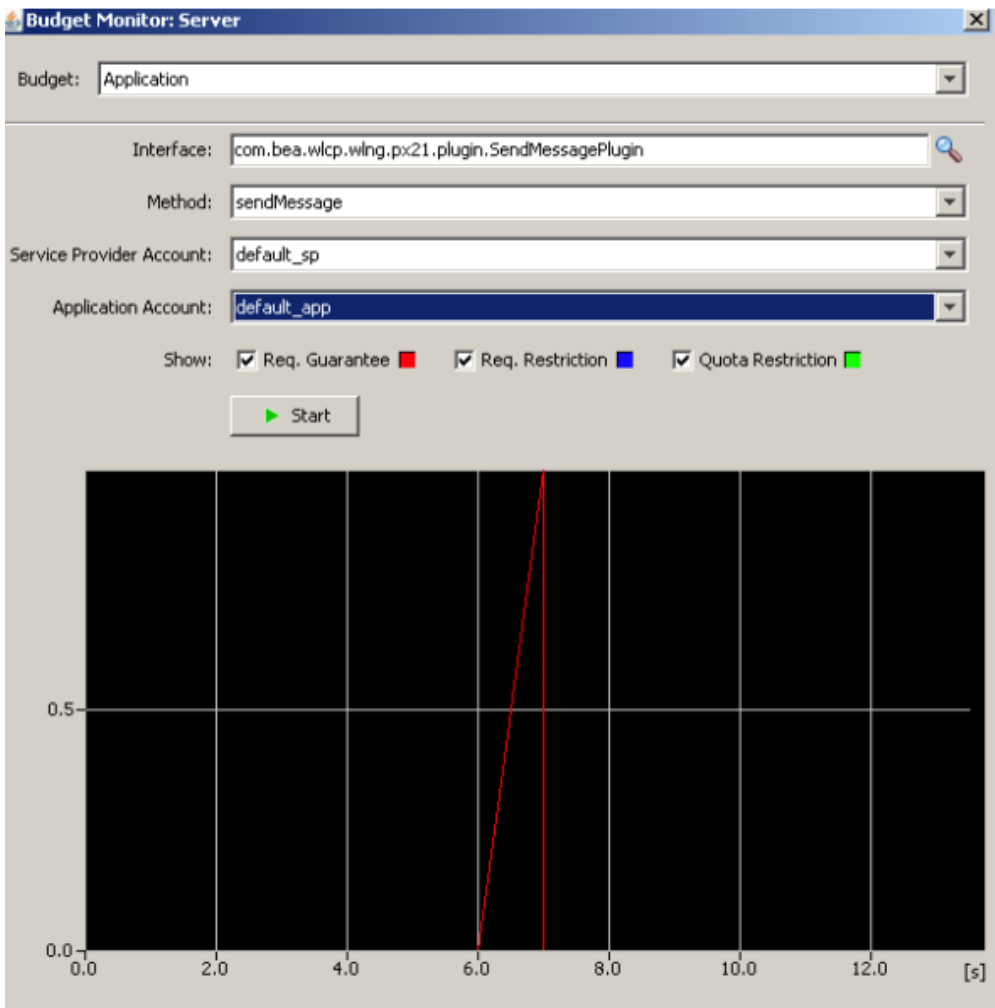
When you have completed your edits, simply click the **Close** button on the **SLA Browser** window.

The Budget Monitor

One of the key concepts to understand in dealing with Oracle Communications Services Gatekeeper is that of the budget. Based on rates, restrictions, and quotas set up in SLAs, the budget measures the level of access an application has to Oracle Communications Services Gatekeeper over time. The budget is continuously being decremented based on the application's use of Oracle Communications Services Gatekeeper while, at the same time, it is being incremented based on the contract between the service provider and the operator. If the application's use of Oracle Communications Services Gatekeeper exactly matches the SLA values, the budget level remains the same. In order to help developers visualize this process, the

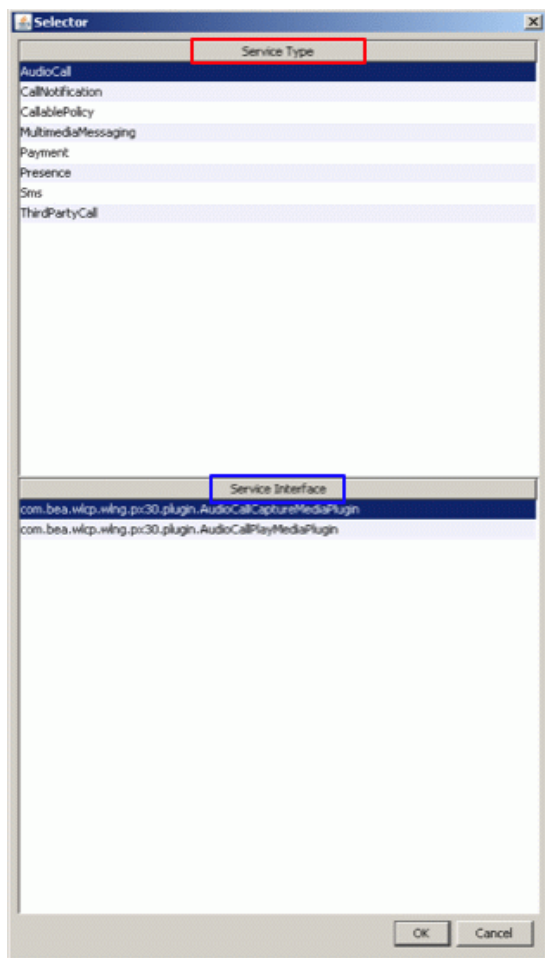
PTE includes a budget monitoring tool that tracks the state of budget usage over time. To access the budget monitor, first make sure you have selected the Server Tool and are connected to the server. Then click **Tools -> Budget Monitor** in the Menu Bar. The **Budget Monitor** window pops up. See [Figure 2-27](#).

Figure 2-27 The Budget Monitor window



Select the type of budget you wish to monitor from the **Budget** drop down list. Select the interface type by clicking the magnifying glass icon. The **Selector** appears. See [Figure 2-28](#).

Figure 2-28 The Selector



Select the **Service Type** and the specific **Service Interface** you are interested in monitoring and then click **OK**.

Note: You must create instances of certain communication services to use them in Oracle Communications Services Gatekeeper. For these communication services, only Service Types that have been instantiated show up in the Selector tool.

When the Selector window is closed, you return to the Budget Monitor window. Select the **Method**, associated **Service Provider Account** and **Application Account** information from the drop down lists. Click the **Start** button to begin monitoring.

The X-axis indicates time in seconds and the Y-axis the amount of budget that is available. The number of units in the Y-axis depends on values assigned in the SLA for the selected budget. Rates, quotas, and restrictions can all be mapped.

Navigating the Platform Test Environment GUI

Extending the Platform Test Environment

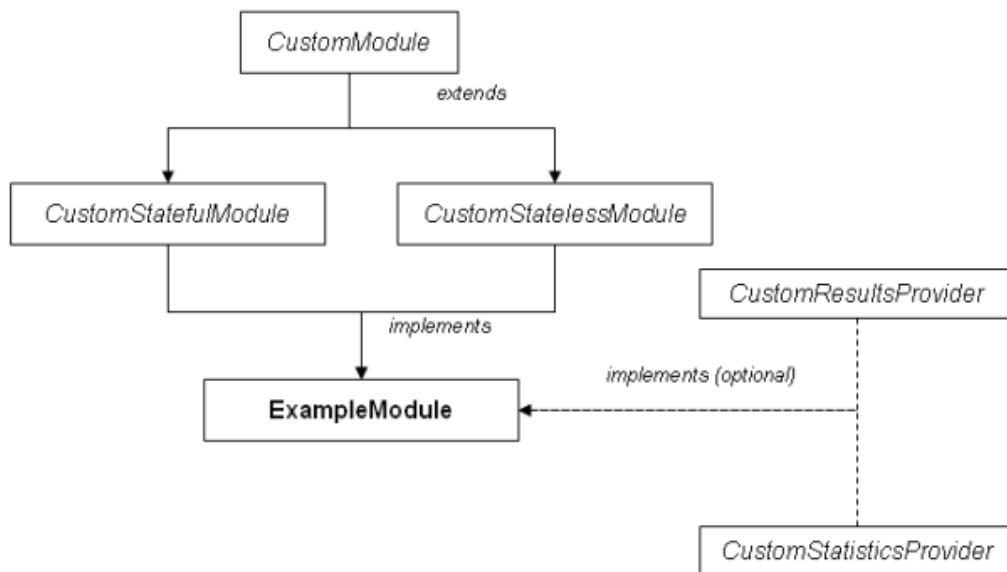
Extending the Platform Test Environment

One of the most common uses for the Platform Testing Environment is to test extension communication services. Depending on how those extensions are implemented, you may need to create one or more new modules so that the PTE can interact successfully with your new communication service. You can implement new client modules, and even new clients containing multiple modules, if support for the application-facing interface that you want your communication service to use is not already available in the PTE. You can also implement new simulators, if the network node type that you want your communication service to interact with is not available. From the point of view of the PTE, a module is a module.

The only relevant distinction in the PTE is between modules for operations that simply execute and return and those for operations that start a process which runs until it is turned off. These are called, respectively, stateless and stateful modules. See [Figure 2-12](#) for more information.

Stateless modules must implement the `CustomStatelessModule` SPI and stateful modules must implement the `CustomStatefulModule` SPI. There are two additional, optional interfaces that can be implemented if you would like your module to display results (for example, a notification, a message from the network delivered to a client Web Service) or provide statistics in the GUI. The custom module SPI hierarchy is as follows:

Figure 3-1 The Custom SPI Hierarchy

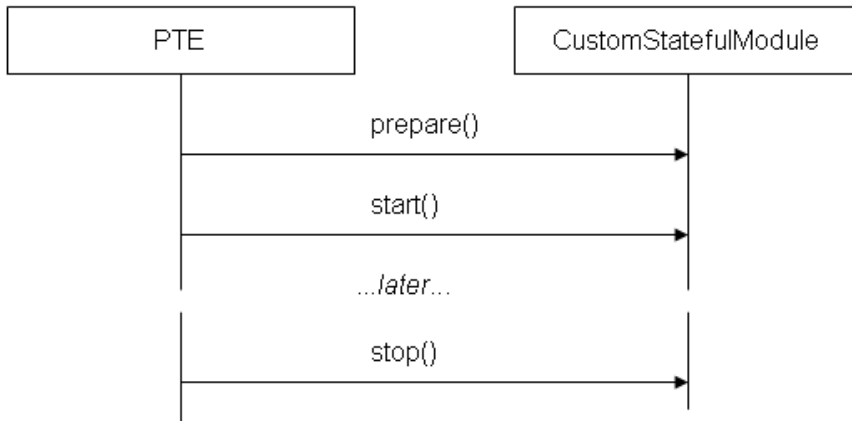


Any module that is created must be packaged as a .jar file which must be located in the \$PTE_HOME\$/lib/modules/ directory. The root of the .jar file must include a descriptor file called module.xml. All custom modules automatically load when the PTE starts up.

Note: The modules created for use with the example communication service are located in <bea_home>/wlng_pds400/example/pte_module.

The Stateful SPI

Figure 3-2 shows the execution sequence for a stateful module:

Figure 3-2 The Execution Sequence for a Stateful Module

The following listing is the SPI that must be implemented by stateful PTE modules.

Listing 3-1 CustomStatefulModule SPI

```

package com.bea.wlcp.wlng.et.spi;

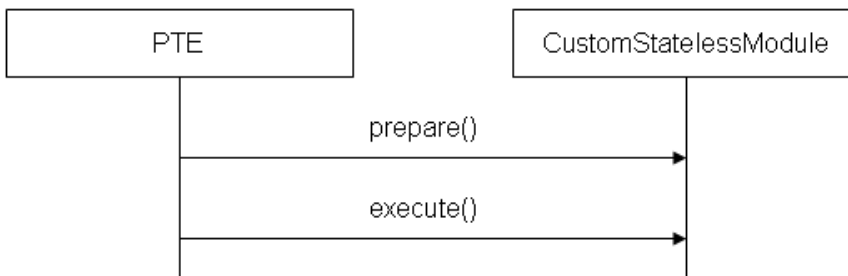
/**
 * This interface must be implemented by a custom stateful module.
 * A stateful module has a start() and a stop() method and will be
 * represented in the UI by the Start/Stop button.
 * Note: a stateful module is not used in duration tests.
 *
 * @author Copyright (c) 2008 by BEA Systems, Inc. All Rights Reserved.
 */
public interface CustomStatefullModule extends CustomModule {
/**
 * Starts the module.
 *
 * @param context The custom module context
 */
}
  
```

```
    * @return true if the module successfully started
    * @throws Exception Any exception preventing the module to start
    */
    public boolean start(CustomModuleContext context) throws Exception;
/**
    * Stops the module.
    * @param context The custom module context
    * @return true if the module successfully stopped
    * @throws Exception Any exception preventing the module to stop
    */
    public boolean stop(CustomModuleContext context) throws Exception;
}
```

The Stateless SPI

Figure 3-3 shows the execution sequence for a stateless module:

Figure 3-3 The Execution Sequence for a Stateless Module



The following listing is the SPI that must be implemented by stateless PTE modules.

Listing 3-2 CustomStatelessModule SPI

```

package com.bea.wlcp.wlng.et.spi;

/**
 * This interface must be implemented by custom stateless module.
 * A stateless module has only an execute() method and will be
 * represented in the UI by the Send button.
 *
 * @author Copyright (c) 2008 by BEA Systems, Inc. All Rights Reserved.
 */

public interface CustomStatelessModule extends CustomModule {
    /**
     * Asks the module to execute its job and return the result.
     *
     * @param context The custom module context
     * @return The result of the execution
     * @throws Exception Any exception that occurred during the execution
     */
    public Object execute(CustomModuleContext context) throws Exception;
}

```

The Custom Base SPI

This following is the base SPI for custom PTE modules. It should *not* be implemented directly. See the first comment.

Listing 3-3 The Custom Base SPI

```

package com.bea.wlcp.wlng.et.spi;

```

Extending the Platform Test Environment

```
import com.bea.wlcp.wlng.et.api.CustomModuleContext;

/**
 * This interface defines the general API a custom module must implement.
 * Note: a custom module should NOT implement this interface directly but
 * one of the subinterface like CustomStatefulModule or
CustomStatelessModule.
 *
 * @author Copyright (c) 2008 by BEA Systems, Inc. All Rights Reserved.
 */
public interface CustomModule {

    /**
     * Prepares the module with the given context. This method is invoked
before
     * the module is executed: it can be used by the module to prepare
     * any internal states needed.
     * Note: when a duration test is performed on the Platform Test
Environment,
     * prepare() is invoked only once at the beginning of the duration test.
     *
     * @param context The context of the custom module
     * @throws Exception Any exception that occurred during the module
preparation
     */
    public void prepare(CustomModuleContext context) throws Exception;
```

The Custom Results Provider SPI

The following listing is the SPI that must be implemented by modules that wish to display some sort of results in the GUI.

Listing 3-4 The CustomResultsProvider SPI

```
package com.bea.wlcp.wlng.et.spi;

/**
 * A custom module can implement this interface if it wants to provide
 * a list of results in the UI. The PTE will automatically display a list
 * and handle the user interaction with it.
 *
 * @author Copyright (c) 2008 by BEA Systems, Inc. All Rights Reserved.
 */

public interface CustomResultsProvider {

    /**
     * Clears the results.
     */

    public void clearResults();

    /**
     * Returns an array of string that will be used to create
     * the name of each column of the results table.
     * @return An array of string to create the column headers
     */

    public String[] getResultsHeaders();

    /**
     * Returns the results. Each result is composed of a map whose keys are
     * the same as the strings returned by getResultsHeaders().
     */
}
```

```
*  
  
* Note: It is up to the custom module to accumulate the results until  
* this method is invoked by the PTE.  
*  
* @return A list of results  
*/  
  
public List<Map<String,String>> getResults();  
}
```

The Custom Statistics Provider SPI

The following listing is the SPI that must be implemented by modules that wish to display statistics in the GUI.

Listing 3-5 The CustomStatisticsProvider SPI

```
package com.bea.wlcp.wlmg.et.spi;  
  
/**  
 * A custom module can implement this interface if it wants to provide  
 * some statistics in the UI. The PTE will automatically display a list  
 * and handle the user interaction with it.  
 * @author Copyright (c) 2008 by BEA Systems, Inc. All Rights Reserved.  
 */  
  
public interface CustomStatisticsProvider {  
  
    /**  
     * Clears the statistics.  
     */  
  
    public void clearStatistics();  
}
```

```

/**
 * Returns a map of statistics. Each key represent a particular statistic
 * and the value the value of the statistic.
 * @return The map of statistics
 */
public Map<String,String> getStatistics();
}

```

The Context API

The following listing is the API that allows modules to acquire context.

Listing 3-6 The Context API

```

package com.bea.wlcp.wlng.et.api;

/**
 * This interface defines the context available to a custom module.
 *
 * @author Copyright (c) 2008 by BEA Systems, Inc. All Rights Reserved.
 */
public interface CustomModuleContext {

    /**
     * Returns the custom module data object as described in the module.xml
     * @return The custom module data object
     */
    public Object getData();
}

```

Extending the Platform Test Environment

```
/**
 * Returns the module of the specified type that this module depends on.
 * If there are many modules of the same type, the one chosen by the user
 * in the UI will be chosen.
 *
 * @param type The type of module
 * @return The module instance of the specified type
 */
public CustomModule getDependency(String type);

/**
 * Prepares the stub that the module will use to send a request. The PTE
 * will perform various changes to the stub depending on the UI settings,
 * like TCP Monitor or Override Endpoint.
 *
 * @param stub The stub to prepare
 * @param path The path to the parameter declared in module.xml that
corresponds
 * to the stub url. Use null if it doesn't have any corresponding parameter.
 */
public void prepareStub(Stub stub, String path);

/**
 * Deploy (or undeploy) a service using a specific WSDD file.
 *
 * @param wsddFile The WSDD file that the axis server will execute
 * @throws Exception Any exception when executing the command
 */
```



```

    */
    public void axisDeploy(String wsddFile) throws Exception;

```

The Module.xml Descriptor File

Every module is packaged in a .jar file with a descriptor file, `module.xml`, in its root. What is in the file depends on the nature of the module.

The following is the listing for a client module and the simulator module supplied with the example communication service:

Listing 3-7 The example module.xml

```

<module-factory xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://www.bea.com/ns/wlng/40/et">

    <module name="example_application_initiated"
        type="client"
        class="com.bea.wlcp.wlng.et.example.SendDataModule"
        version="1.0"
        depends="session"
        uiPanel="client"
        uiTabs="Other,Example,Application-Initiated"
    >

    <data>
        <parameter name="Parameters"
            class="com.bea.wlcp.wlng.et.example.SendDataData"
            occurs="1">

```

Extending the Platform Test Environment

```
<parameter name="url"
            class="java.lang.String"
            occurs="1"
            default="http://${at.host}:${at.port}/example/SendData"
            monitor="true"/>

<parameter name="data"

class="com.acompany.schema.example.data.send.local.SendData"
            occurs="1">

<parameter name="address"
            class="java.net.URI"
            occurs="1"
            default="tel:1234"/>

<parameter name="data"
            class="java.lang.String"
            occurs="1"
            default="Hello, world"/>
</parameter>
</parameter>

</data>
</module>

...
```

```
<module name="example_simulator"
    type="netex"
    class="com.bea.wlcp.wlng.et.example.SimulatorModule"
    version="1.0"
    uiPanel="simulator"
    uiTabs="Netex"
    >
<data>
    <parameter name="Parameters"
        class="com.bea.wlcp.wlng.et.example.SimulatorData"
        occurs="1">

        <parameter name="port"
            class="int"
            occurs="1"
            default="5001"/>

    </parameter>
</data>
</module>

</module-factory>
```

Below is the entire .xsd file for module.xml:

Listing 3-8 The module.xsd File

```
<?xml version="1.0"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified">

    <!-- Main element that describes one or more modules -->
    <xs:element name="module-factory">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="module" minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <!-- Defines a single module -->
    <xs:element name="module">
        <xs:complexType>
            <xs:sequence>
                <!-- Optional data of the module -->
                <xs:element ref="data" minOccurs="0" maxOccurs="1"/>
            </xs:sequence>
            <!-- Name of the module. It will be used also for the display -->
            <xs:attribute name="name" type="xs:string" use="required"/>
            <!-- Type of the module -->
            <xs:attribute name="type" type="xs:string" use="required"/>
            <!-- Class of the module (fully qualified) -->
            <xs:attribute name="class" type="xs:string" use="required"/>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

```

<!-- Version of the module -->
<xs:attribute name="version" type="xs:string" use="required"/>
<!-- Name of the module this module depends on.
Predefined types are:
- session : session module
- axis : axis server module
The PTE will make sure that before this module is started, the
dependent module is running. -->
<xs:attribute name="depends" type="xs:string" use="optional"/>
<!-- UI panel where the module will be located (see ui-panels) -->
<xs:attribute name="uiPanel" type="ui-panels" use="required"/>
<!-- Location of the module in the panel tabs.
The location is a list of UI tab names separated by comma. For example:
    "Other,Example,SendData"
means that the module will be in a tab named "SendData"
located in the tab "Example" located in tab "Other".
The name of each tab is available in the UI.
If a tab doesn't exist for a particular name, it will be created.-->
<xs:attribute name="uiTabs" type="xs:string" use="required"/>
</xs:complexType>
</xs:element>

<!-- Available UI panels -->
<xs:simpleType name="ui-panels">
  <xs:restriction base="xs:string">
    <!-- Client panel -->
    <xs:enumeration value="client"/>

```

Extending the Platform Test Environment

```
<!-- Simulator panel -->
<xs:enumeration value="simulator"/>
</xs:restriction>
</xs:simpleType>

<!-- Data of the module -->
<xs:element name="data">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="parameter" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<!-- A single parameter -->
<xs:element name="parameter">
  <xs:complexType>
    <xs:sequence>
      <!-- Can contain other parameters too -->
      <xs:element ref="parameter" minOccurs="0" maxOccurs="unbounded"/>
      <!-- Values restriction of the parameter (see restricted) -->
      <xs:element ref="restricted" minOccurs="0" maxOccurs="1"/>
      <!-- Internal use only -->
      <xs:element ref="instance" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <!-- Name of the parameter. It will be used to access the member
    of the parameter (MBean-style) -->
```

```

<xs:attribute name="name" type="xs:string" use="required"/>
<!-- Fully qualified name of the parameter -->
<xs:attribute name="class" type="xs:string" use="required"/>
<!-- Occurrences of the parameter (see parameter-occurs) -->
<xs:attribute name="occurs" type="parameter-occurs" use="required"/>
<!-- Default value of the parameter -->
<xs:attribute name="default" type="xs:string" use="optional"/>
<!-- Set to true if this parameter represent an URL to a stub. If true,
it can be monitored by TCP monitor and have other properties -->
<xs:attribute name="stub" type="xs:boolean" use="optional"/>
<!-- Set to true if this parameter must be instanciated at creation
time.
    This is only useful if the parameter is optional. -->
<xs:attribute name="instanciate" type="xs:boolean" use="optional"/>
<!-- Internal use only -->
<xs:attribute name="preview" type="xs:boolean" use="optional"/>
<!-- Internal use only -->
<xs:attribute name="help" type="xs:boolean" use="optional"/>
<!-- Internal use only -->
<xs:attribute name="multiline" type="xs:integer" use="optional"/>
<!-- Internal use only -->
<xs:attribute name="timebase" type="parameter-timebase"
use="optional"/>
<!-- Optional display string to use instead of the name in the UI -->
<xs:attribute name="display" type="xs:string" use="optional"/>
</xs:complexType>
</xs:element>

```

```
<!-- The value the parameter is restricted to -->
<xs:element name="restricted">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="value" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```
<!-- A value has a content only -->
<xs:element name="value">
  <xs:complexType>
    <xs:attribute name="content" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>
```

```
<!-- The occurrences of a parameter -->
<xs:simpleType name="parameter-occurs">
  <xs:restriction base="xs:string">
    <!-- required (one and only one) -->
    <xs:enumeration value="1"/>
    <!-- optional -->
    <xs:enumeration value="?" />
    <!-- one or more -->
    <xs:enumeration value="+" />
    <!-- zero or more -->
    <xs:enumeration value="*" />
  </xs:restriction>
</xs:simpleType>
```



```

        <!-- tree of parameter-->

        <xs:enumeration value="t"/>

    </xs:restriction>
</xs:simpleType>

<!-- Internal use only -->
<xs:element name="instance">
    <xs:complexType>
        <xs:attribute name="v1" type="xs:string" use="required"/>
        <xs:attribute name="v2" type="xs:string" use="optional"/>
        <xs:attribute name="v3" type="xs:string" use="optional"/>
    </xs:complexType>
</xs:element>

<!-- Internal use only -->
<xs:simpleType name="parameter-timebase">
    <xs:restriction base="xs:string">
        <xs:enumeration value="ms"/>
        <xs:enumeration value="s"/>
        <xs:enumeration value="min"/>
        <xs:enumeration value="h"/>
    </xs:restriction>
</xs:simpleType>

</xs:schema>

```

Extending the Platform Test Environment

Using the Unit Test Framework with the Platform Test Environment

Unit tests are a core part of any testing cycle. Data are input into the system and the results are retrieved from the system and compared to expected values, all programmatically. This chapter describes the PTE Unit Test Framework. It consists of:

- [Creating and Running the Unit Test](#)
- [The Example Unit Test](#)

Creating and Running the Unit Test

The Unit Test Framework allows you to create unit tests for the PTE easily. You implement your test class based on the abstract class `WlngBaseTestCase` and it manages the mechanics of using JMX and JMS to connect to the PTE for you. A `test.properties` file located in the same directory can be used to define commonly changed properties of the test.

Note: The `WlngBaseTestCase` class is located in
`<bea_home>/wlng_pds400/lib/wlng/pte_api.jar`.

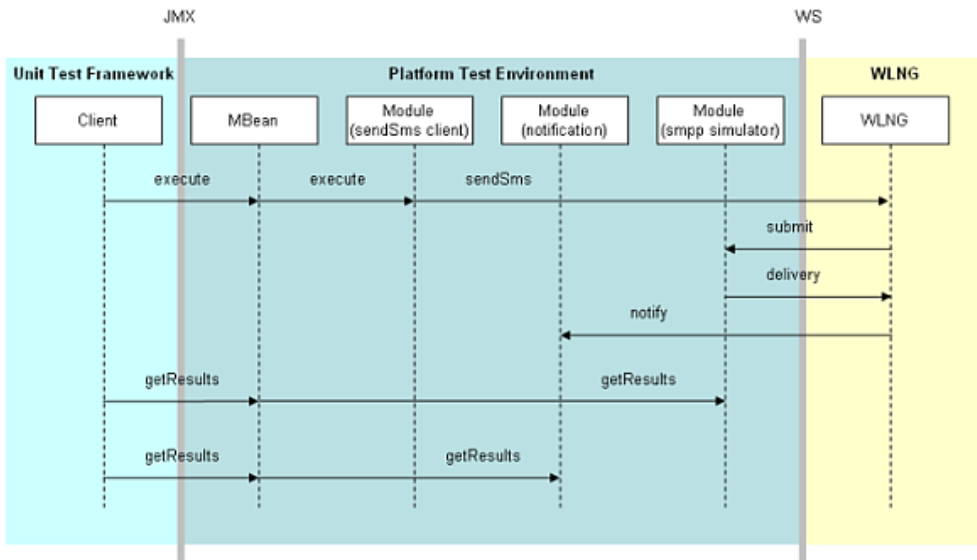
There are five basic steps to creating a unit test for the PTE:

1. Create any necessary client or simulator modules for the PTE using the required SPI and XML configuration files
2. Implement a test class based on the abstract class `WlngBaseTestCase`
3. Provision Gatekeeper
4. Start the Platform Test Environment and make sure the modules are correctly loaded

5. Run the test class

Note: The PTE should be running in Console (non-GUI) mode when you run your test. See [Installing and Running the Platform Test Environment](#) for more information on starting in Console mode.

Figure 4-1 An SMS Unit Test Sequence



The test sequence flow is as follows:

1. The test client calls `execute` on the PTE's Module Management MBean. The mechanics of the JMX call are taken care of by the base class.
2. The MBean calls `execute` on the specified Module, in this case `sendSMS`. This request includes a request for delivery receipts.
3. The `sendSMS` Module sends the request to Gatekeeper.
4. Gatekeeper processes it and submits it to the network simulator module, in this case the SMPP module
5. The simulator module returns a Delivery Receipt to Gatekeeper
6. Gatekeeper sends the receipt on to the Notification module (which represents the client Web Service implementation)

7. The test client retrieves the result of Gatekeeper's `submit` from the SMPP simulator
8. The test client retrieves the Delivery Receipt from the Notification module

The Example Unit Test

To help you understand more clearly how all this works, there is an example unit test, which tests the example communication service, using the example clients and simulator. In standard installations, it is located in

`<bea_home>/wlng_pds400/example/unit_test/src/com/bea/wlcp/wlng/pds/example`
 . See [Listing 4-1](#) below.

Listing 4-1 A Unit Test for the Example Communication Service

```
package com.bea.wlcp.wlng.et.example;

import com.bea.wlcp.wlng.et.api.WlngBaseTestCase;

import java.util.List;
import java.util.Map;

/**
 * This class illustrates how to use the Unit Test Framework to
 * test the Communication Service Example. A few things are assumed before
 * running this class:
 *
 * - the Services Gatekeeper should be running and configured properly
 * - the CS example should be deployed and ready
 *
 * Note: this example uses also the wlngJmx to be able to access the Services
 * Gatekeeper
 *
 * MBeans to ask the CS example plugin to connect to the Netex simulator.
 */
```

Using the Unit Test Framework with the Platform Test Environment

```
*/

public class TestSendData extends WlngBaseTestCase {

    private static final String SEND_DATA_MBEAN =
"com.bea.wlcp.wlng.ptc:group=traffic,name=SendData";

    private static final String NETWORK_TRIGGERED_MBEAN =
"com.bea.wlcp.wlng.ptc:group=traffic,name=NetworkTriggered";

    private static final String NOTIF_MANAGER_MBEAN =
"com.bea.wlcp.wlng.ptc:group=client,name=NotificationManager";

    private static final String NOTIF_MBEAN =
"com.bea.wlcp.wlng.ptc:group=client,name=Notification";

    private static final String NETEX_SIMULATOR_MBEAN =
"com.bea.wlcp.wlng.ptc:group=netex,name=Simulator";

    private static final String EXAMPLE_PLUGIN_MBEAN =
        "com.bea.wlcp.wlng:AppName=es_example_nt#4.0," +
        "InstanceName=example_netex_plugin," +

"Type=com.acompany.plugin.example.netex.management.ExampleMBean";

    public TestSendData() throws Exception {
    }

    @Override
    protected void setUp() throws Exception {
        super.setUp();

        wlngJmx.open("localhost", 8001, "weblogic", "weblogic");
        start(NETEX_SIMULATOR_MBEAN);
    }
}
```

```

@Override

protected void tearDown() throws Exception {

    wlngJmx.close();

    stop(NETEX_SIMULATOR_MBEAN);

    super.tearDown();

}

public void testSendData() throws Exception {

    assertTrue(isRunning(NETEX_SIMULATOR_MBEAN));

    resetStatistics(NETEX_SIMULATOR_MBEAN);

    wlngJmx.invokeOperation(EXAMPLE_PLUGIN_MBEAN, "connect");

    String data = "Hello at " + System.currentTimeMillis();

    String to = "tel:1234";

    putParameter(SEND_DATA_MBEAN, "url",
"http://localhost:8001/example/SendData");

    putParameter(SEND_DATA_MBEAN, "data.data", data);

    putParameter(SEND_DATA_MBEAN, "data.address", to);

    start(SESSION_MBEAN);

    assertTrue(isRunning(SESSION_MBEAN));

    execute(SEND_DATA_MBEAN);

    Thread.sleep(2000);

```

Using the Unit Test Framework with the Platform Test Environment

```
stop(SESSION_MBEAN);

Map<String,String> stats = listAllStatistics(NETEX_SIMULATOR_MBEAN);
System.out.println("Simulator statistics: "+stats);
assertEquals("MessageReceived", "1", stats.get("MessageReceived"));
assertEquals("MessageSent", "0", stats.get("MessageSent"));
}

public void testSendNetworkTriggeredData() throws Exception {
    String data = "Hello at " + System.currentTimeMillis();
    String from = "tel:1234";
    String to = "tel:7878";
    String correlator = "1234567890";

    assertTrue(isRunning(NETEX_SIMULATOR_MBEAN));
    resetStatistics(NETEX_SIMULATOR_MBEAN);

    wlngJmx.invokeOperation(EXAMPLE_PLUGIN_MBEAN, "connect");

    start(SESSION_MBEAN);
    assertTrue(isRunning(SESSION_MBEAN));

    putParameter(NOTIF_MANAGER_MBEAN, "url",
"http://localhost:8001/example/NotificationManager");
    putParameter(NOTIF_MANAGER_MBEAN, "start.address", "tel:7878");
    putParameter(NOTIF_MANAGER_MBEAN, "start.correlator", correlator);
}
```



```

    putParameter(NOTIF_MANAGER_MBEAN, "start.endpoint",
"http://localhost:13444/axis/services/Notification");

    putParameter(NOTIF_MANAGER_MBEAN, "stop.correlator", correlator);
    start(NOTIF_MANAGER_MBEAN);

    start(NOTIF_MBEAN);

    putParameter(NETWORK_TRIGGERED_MBEAN, "data", data);
    putParameter(NETWORK_TRIGGERED_MBEAN, "fromAddress", from);
    putParameter(NETWORK_TRIGGERED_MBEAN, "toAddress", to);

    clearResults(NOTIF_MBEAN);

    execute(NETWORK_TRIGGERED_MBEAN);

    Thread.sleep(2000);

    stop(NOTIF_MBEAN);
    stop(NOTIF_MANAGER_MBEAN);
    stop(SESSION_MBEAN);

    Map<String,String> stats = listAllStatistics(NETEX_SIMULATOR_MBEAN);
    System.out.println("Simulator statistics: "+stats);
    assertEquals("MessageReceived", "0", stats.get("MessageReceived"));
    assertEquals("MessageSent", "1", stats.get("MessageSent"));

    List<Map<String,String>> results = listAllResults(NOTIF_MBEAN);
    System.out.println("Notification results: "+results);

```

Using the Unit Test Framework with the Platform Test Environment

```
        assertEquals("Correlator", correlator,
results.get(0).get("Correlator"));

        assertEquals("From Address", from, results.get(0).get("From Address"));

        assertEquals("Data", data, results.get(0).get("Data"));
    }

}
```
