CHAPTER 4

Nonblocking API

Revised: November 8, 2010, OL-21085-02

Introduction

This chapter introduces features unique to the Nonblocking Application Programming Interface (API). It lists all methods of the Nonblocking API and ends with code examples.

Information About Reliability Support

The Nonblocking API can work in two different modes, reliable and nonreliable, as described below. When the mode is not specified, the default is reliable mode.

Reliable Mode

In reliable mode, the API ensures that no requests to the Subscriber Manager (SM) are lost. The API maintains an internal storage for all API requests that are sent to the SM. After a reply from the SM is received, the request is considered committed and the API can remove the request from its internal storage. In case of connection failure between the API and the SM, the API accumulates all requests in its internal storage until the connection to the SM is established. On reconnection, the API resends all noncommitted and committed requests to the SM, so that no requests are lost.

Note In reliable mode, the order of resending requests is guaranteed. The API resends the requests in the same chronological order that they were called.

Nonreliable Mode

In **nonreliable** mode, the API does not ensure that requests sent to the SM are executed. In addition, all requests that are sent by the API when the connection to the SM is down are lost unless an external reliability mechanism is implemented.

Autoreconnect Support

The Nonblocking API supports autoreconnection to the SM in case of connection failure. When this option is activated, the API can determine when the connection to the SM is lost. When the connection is lost, the API activates a reconnection task that tries to reconnect to the SM until it is successful.

Note The autoreconnect support option can be activated regardless of the reliability mode.

Multithreading Support

The Nonblocking API supports an unlimited number of threads calling its methods simultaneously.

Note In a multithreaded scenario for the Nonblocking API, the order of invocation is guaranteed. The API performs operations in the same chronological order that they were called.

Information About the ResultHandler Interface

The Nonblocking API enables setting a result handler. A result handler is an interface with two methods, **handleSuccess** and **handleError**, as outlined in the following code:

```
public interface ResultHandler {
    /**
     * handle a successful result
     */
    public void handleSuccess(long handle, Object result);

    /**
     * handle a failure result
     */
    public void handleError(long handle, Object result);
}
```

You should implement this interface if you want to be informed about the success or error results of operations performed through the API.

Note This is the only interface for retrieving results; they cannot be returned immediately after the API method has returned to the caller.

Note To be able to receive operation results, you should set the result handler of the API before calling API methods whose results you want to receive. It is a good practice to set the result handler after the API is connected (as in the example below).

Both **handleSuccess** and **handleError** methods accept two parameters:

- **Handle**—Each API operations return-value is a handle of type **long**. This handle enables correlation between operation calls and their results. When a **handle...** operation is called with a handle of value *X*, the result will match the operation that returned the same handle value (*X*) to the caller.
- **Result**—Actual result of the operation. Some operations may return a result of NULL.

ResultHandler Interface Example

The following example is a simple implementation of a result handler that prints a message to **stdout** (when the result is successful) or to **stderr** (when the result is failure). This main method initiates the API and assigns a result handler.

For correct operation of the result handler, follow the code sequence given in this example.

Note This example does not demonstrate the use of callback handles.

```
import com.pcube.management.framework.rpc.ResultHandler;
import com.pcube.management.api.SMNonBlockingApi;

public class ResultHandlerExample implements ResultHandler{

    public void handleSuccess(long handle, Object result) {
        System.out.println("success: handle="+handle+", result="+result);
    }

    public void handleError(long handle, Object result) {
        System.err.println("error: handle="+handle+", result="+result);
    }

    public static void main (String args[]) throws Exception{
        if (args.length != 1) {
            System.err.println("usage: ResultHandlerExample <sm-ip>");
            System.exit(1);
        }

        //note the order of operations!
        SMNonBlockingApi nbapi = new SMNonBlockingApi();
        nbapi.connect(args[0]);
        nbapi.setResultHandler(new ResultHandlerExample());
        nbapi.login(...);
    }
}
```

Nonblocking API Construction

In addition to the constructors described in the "API Construction" section on page 2-2, the Nonblocking API provides constructors that enable setting the reconnect period and the reliability mode.

Nonblocking API Syntax

The syntax for the additional Nonblocking API constructors is shown in the following code block:

```
public SMNonBlockingApi(long autoReconnectInterval)
public SMNonBlockingApi(boolean reliable, long autoReconnectInterval)
public SMNonBlockingApi(String legName, long autoReconnectInterval)
public SMNonBlockingApi(String legName, boolean reliable, long autoReconnectInterval)
```

Nonblocking API Arguments

The following is a description of the constructor arguments for the additional Nonblocking API constructors:

- **autoReconnectInterval**

  Defines the interval (in milliseconds) for attempting reconnection by the reconnection task, as follows:

  - If the value is 0 or less, the reconnection task is not activated (no autoreconnect is attempted).
  - If the value is greater than 0 and if there is a connection failure, the reconnection task is activated every **autoReconnectInterval** milliseconds.

- Default value is –1 (no autoreconnect is attempted).

Note To enable the autoreconnect support, the connect method of the API must be activated at least once. For more information, see the "Nonblocking API Code Examples" section on page 4-9.

- **reliable**

  A flag that defines whether the API should work in reliable mode, as follows:

  - TRUE—The API works in reliable mode.
  - FALSE—The API works in nonreliable mode.

- Default value is TRUE (the API works in reliable mode).
- **legName** The name of the LEG, as described in the "API Construction" section on page 2-2.

Nonblocking API Examples

The following code constructs a reliable API with an autoreconnection interval of 10 seconds:

```
SMNonBlockingAPI nbapi = SMNonBlockingAPI(10000);
nbapi.connect(<SM IP address>);
```

The following code constructs a reliable API without autoreconnection support:

```
// API construction
SMNonBlockingAPI nbapi = SMNonBlockingAPI();
// Connect to the API
nbapi.connect(<SM IP address>);
```

The following code constructs a nonreliable API with autoreconnection support:

```
// API construction
SMNonBlockingAPI nbapi = SMNonBlockingAPI(false,10000);
// Initial connection - to enable the reconnect task
nbapi.connect(<SM IP address>);
```

Nonblocking API Initialization

The Nonblocking API enables initializing certain internal properties for API customization. This initialization is performed using the API **init** method.

Note For the settings to take effect, the **init** method must be called before the **connect** method.

The following properties can be set:

- Output queue size—The internal buffer size defining the maximum number of requests that can be accumulated by the API until they are sent to the SM. The default is 1024.
- Operation timeout—The desired timeout (in milliseconds) on a nonresponding PRPC protocol connection. The default is 45 seconds.

Nonblocking API Initialization Syntax

The syntax for the Nonblocking API init method is as follows:

```
public void init(Properties properties)
```

Nonblocking API Initialization Parameters

The following is a description of the parameters for the Nonblocking API **init** method:

- **properties** (**java.util.Properties**)

  Enables setting the following properties described above:

  - To set the output queue size, use **prpc.client.output.machinemode.recordnum**
  - To set the operation timeout, use **prpc.client.operation.timeout**

Nonblocking API Initialization Example

The following code illustrates how to customize properties during initialization when using the Nonblocking API. Note that the **init** method is called before the **connect** method.

```
// API construction
SMNonBlockingAPI nbapi = SMNonBlockingAPI(10000);
// API initialization
java.util.Properties p = new java.util.Properties();
p.setProperty("prpc.client.output.machinemode.recordnum", 2048);
p.setProperty("prpc.client.operation.timeout", 60000);// 1 minute
nbapi.init(p);
// initial connect to the API to enable the reconnect task
nbapi.connect(<SM API address>);
```

Information About Nonblocking API Methods

This section describes the methods of the Nonblocking API.

All methods return a handle of type **long** that can be used to correlate operation calls and their results. See the "Information About the ResultHandler Interface" section on page 4-2.

The operation results passed to the result handler are similar to the return values described in the same method in the "Blocking API" section on page 3-1, with the exception of:

- Basic types are converted to their Java class representation. For example, **int** is translated to **java.lang.Integer**.
- Return values of **Void** are translated to NULL.

Note An error is passed to the result handler **only if** the matching operation in the Blocking API throws an exception with the same arguments according to the SM database state at the time of the call.

All methods will throw a **java.lang.IllegalStateException** if called before a connection with the SM is established.

This section describes the following methods:

login

This section provides the login operation syntax information.

Syntax

The login syntax:

```
public long login(String subscriberName,
                  String[] mappings,
                  short[] mappingTypes,
                  String[] propertyKeys,
                  String[] propertyValues,
                  String domain,
                  boolean isMappingAdditive,
                  int autoLogoutTime)
```

The operation functionality is the same as the matching Blocking API operation. See the "login" section on page 3-4 for more information.

logoutByName

This section provides the logoutByName operation syntax information.

Syntax

The logoutByName syntax:

```
public long logoutByName(String subscriberName,
                         String[] mappings,
                         short[] mappingTypes)
```

The operation functionality is the same as the matching Blocking API operation. See the "logoutByName" section on page 3-7 for more information.

logoutByNameFromDomain

This section provides the logoutByNameFromDomain operation syntax information.

Syntax

The logoutByNameFromDomain syntax:

```
public long logoutByNameFromDomain(String subscriberName,
                                   String[] mappings,
                                   short[] mappingTypes,
                                   String domain)
```

The operation functionality is the same as the matching Blocking API operation. See the "logoutByNameFromDomain" section on page 3-9 for more information.

logoutByMapping

This section provides the logoutByMapping operation syntax information.

Syntax

The logoutByMapping syntax:

```
public long logoutByMapping(String mapping,
                            short mappingType,
                            String domain)
```

The operation functionality is the same as the matching Blocking API operation. See the "logoutByMapping" section on page 3-10 for more information.

loginCable

This section provides the loginCable operation syntax information.

Syntax

The loginCable syntax:

```
public long loginCable(String CPE,
                       String CM,
                       String IP,
                       int lease,
                       String domain,
                       String[] propertyKeys,
                       String[] propertyValues)
```

The operation functionality is the same as the matching Blocking API operation. See the "loginCable" section on page 3-12 for more information.

logoutCable

This section provides the logoutCable operation syntax information.

Syntax

```
public long logoutCable(String CPE,
                        String CM,
                        String IP,
                        String domain)
```

The operation functionality is the same as the matching Blocking API operation. See the "logoutCable" section on page 3-14 for more information.

Nonblocking API Code Examples

This section illustrates a code example for logging in and logging out subscribers.

Login and Logout

The following example logs in a predefined number of subscribers to the SM and then logs them out. Note the implementation of a disconnect listener and a result handler.

```
package nonblocking;

import com.pcube.management.framework.rpc.DisconnectListener;
import com.pcube.management.framework.rpc.ResultHandler;
import com.pcube.management.api.SMNonBlockingApi;
import com.pcube.management.api.SMApiConstants;

class LoginLogoutDisconnectListener implements  DisconnectListener {
    public void connectionIsDown() {
        System.err.println("disconnect listener:: connection is down");
    }
}

class LoginLogoutResultHandler implements ResultHandler {
    int count = 0;

    //prints a success result every 100 results
    public synchronized void handleSuccess(long handle, Object result) {
        Object tmp = null;
        if (++count%100 == 0) {
            tmp = result instanceof Object[] ? ((Object[])result)[0] : result;
            System.out.println("\tresult "+count+":\t"+tmp);
        }
    }

    //prints every error that occurs
    public synchronized void handleError(long handle, Object result) {
        System.err.println("\terror: "+count+":\t"+ result);
        ++count;
    }

    //waits for result number 'last result' to arrive
    public synchronized void waitForLastResult(int lastResult) {
        while (count<lastResult) {
            try {
                wait(100);
            } catch (InterruptedException ie) {
                ie.printStackTrace();
            }
        }
    }
}
```

```
public class LoginLogout {
    public static void main (String args[]) throws Exception{
        //check arguments
        checkArguments(args);
        int numSubscribersToLogin = Integer.parseInt(args[2]);

        //instantiation
        SMNonBlockingApi nbapi = new SMNonBlockingApi();
        try {
            //initation
            nbapi.setDisconnectListener(
                new LoginLogoutDisconnectListener());
            nbapi.connect(args[0]);
            LoginLogoutResultHandler resultHandler =
            new LoginLogoutResultHandler();
            nbapi.setResultHandler(resultHandler);
            //login
            System.out.println("login of "+numSubscribersToLogin +" subscribers");
            for (int i=0; i<numSubscribersToLogin; i++) {
                nbapi.login("subscriber"+i,    //subscriber name
                            getMappings(i),    //a single ip mapping
                            new short[]{
                                SMApiConstants.MAPPING_TYPE_IP
                            },
                            null,          //no properties
                            null,
                            args[1],       //domain
                            false,         //mappings are not additive
                            -1);           //disable autologout
            }
            resultHandler.waitForLastResult(numSubscribersToLogin);
            //logout
            System.out.println("logout of "+numSubscribersToLogin +" subscribers");
            for (int i=0; i<numSubscribersToLogin; i++) {
                nbapi.logoutByMapping(getMappings(i)[0],
                                      SMApiConstants.MAPPING_TYPE_IP,
                                      args[1]);
            }
            resultHandler.waitForLastResult(numSubscribersToLogin*2);
        } finally {
            nbapi.disconnect();
        }
    }
    static void checkArguments(String[] args)  throws Exception{
        if (args.length != 3) {
            System.err.println("usage: java LoginLogout "+
                               "<SM-address> <domain> <num-susbcribers>");
            System.exit(1);
        }
    }
    //'automatic' mapping generator
    private static String[] getMappings(int i) {
        return new String[]{ "10." +((int)i/65536)%256 + "." +
            ((int)(i/256))%256 + "." + (i%256)};
    }
}
```