CHAPTER 5

Programming with the Service Control Engine Subscriber Application Programming Interface

This module provides a detailed description of the Application Programming Interface (API) programming structure, classes, methods, and interfaces.

Information About API Classes

The following list maps the classes provided by the API.

Package com.scms.api.sce.prpc

PRPC_SCESubscriberApi Class, page 5-3—Main API class.

Package com.scms.api.sce

Indications Listeners

Connection Monitoring

SCE Cascade Topology Support

Operations Result Handling

Package com.scms.common

com.scms.common package contains all data types used by the API.

Programming Guidelines

Programming with Callback Methods

As described in previous sections, many of the API operations are based on callback methods. The user provides a "listener", which is called when certain events occur. The following warning defines the main guideline for programming with callback methods.

Do not perform long operations within the thread of the callback method. Long operations should be performed from a **separate thread**. Moreover, not following this recommendation might result in resource leakage on the client's side.

This caution applies to the following operations:

- LoginPullListener callback methods
- LogoutListener callback methods
- QuotaListenerEx callback methods
- ConnectionListener callback methods

PRPC_SCESubscriberApi Class

The PRPC_SCESubscriberAPI class (resides in a com.scms.sce.api.prpc package) is the main API class that provides the following functionality:

- Constructing the API
- Connecting the API to exactly one SCE (configuring the connection attributes)
- Registering/unregistering indications listeners
- Setting the connection listener
- Performing subscriber provisioning operations
- Disconnecting from the SCE

API Construction

The PRPC_SCESubscriberAPI provides the following constructors:

Syntax:

```
public PRPC_SCESubscriberApi(String apiName, String sceHost)
                                              throws UnknownHostException

public PRPC_SCESubscriberApi(String apiName,
                             String sceHost,
                             long autoReconnectInterval)
                                              throws UnknownHostException

public PRPC_SCESubscriberApi(String apiName,
                             String sceHost,
                             int scePort,
                             long autoReconnectInterval)
                                              throws UnknownHostException
```

Parameters:

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

apiName—Specifies an API name.

Note The API name should be unique per SCE. If you construct more than one API with the same name and connect it to a single SCE, the SCE platform will handle the APIs as one API client. Use this feature only when high-availability is supported. For more information about high availability, see the Implementing High-Availability section.

sceHost—Can be either an IP address or a reachable hostname.

scePort—PRPC protocol TCP port to connect to the SCE (default value is 14374)

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 auto-reconnect is attempted).
- If the value is greater than 0 and a connection failure exists, the reconnection task will be activated every <autoReconnectInterval>milliseconds.
- Default value: -1 (no auto-reconnect is attempted)

Note To enable the auto-reconnect support, the **connect** method of the API **must** be called at least once.

Examples:

The following code constructs an API with an auto-reconnection interval of 10 seconds:

```
PRPC_SCESuscriberAPI sceApi = new PRPC_SCESuscriberAPI("MyApi",
                                                       "10.1.1.1",
                                                        10000);
sceApi.connect();
```

The following code constructs an API without auto-reconnection support:

```
PRPC_SCESuscriberAPI sceApi = new PRPC_SCESuscriberAPI("MyApi",
                                                       "10.1.1.1");
sceApi.connect();
```

Listeners Setup Operations

After initializing the API, it should be set-up with the utilized listeners based on the type of application using the API, and the topology used. For more information about topologies, see the Supported Topologies section.

The listeners setup operations may include:

- Setting a connection listener, described in more detail in the Information About Connection Monitoring section:

  ```
  public void setConnectionListener(ConnectionListener listener)
  ```

- Setting a login-pull listener, described in more detail in the Information About the LoginPullListener Interface Class section:

  ```
  public void registerLoginPullListener(LoginPullListener listener)
  ```

- Setting a logout listener, described in more detail in the Information About the LogoutListener Interface Class section:

```
public void registerLogoutListener(LogoutListener listener)
```

- Setting a quota listener, described in more detail in the Information About the QuotaListenerEx Interface Class section:

```
public void registerQuotaListener(QuotaListener listener)
```

- Setting a redundancy state listener, described in more detail in the Information About the RedundancyStateListener Interface section:

```
public void setRedundancyStateListener(RedundancyStateListener listener)
```

Note The listener registration to the API causes resource allocations in the SCE to support reliable delivery of messages to the listener. Even if the application that uses the API crashes and restarts after a short time the messages are kept and sent to the SCE when the API reconnects.

Advanced Setup Operations

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

Note For 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 SCE (Default: 1024)
- Operation timeout—A suggested time interval about the desired timeout (in milliseconds) on a non-responding PRPC protocol connection (Default: 45 seconds)

Syntax

The syntax for the init method is as follows:

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

Parameters

properties (java.util.Properties)—Enables setting the properties described in Advanced Setup Operations, page 5-5:

- To set the output queue size, use **prpc.client.output.machinemode.recordnum** as a property key
- To set the operation timeout, use **com.scms.api.sce.prpc.regularInvocationTimeout** or **com.scms.api.sce.prpc.listenerInvocationTimeout** as a property key

Note **com.scms.api.sce.prpc.listenerInvocationTimeout** is used for operations that may be invoked from listener callback. This timeout should be shorter than **com.scms.api.sce.prpc.regularInvocationTimeout** to avoid deadlocks.

Customize Properties: Example

This example shows how to customize properties during initialization:

```
// API construction
PRPC_SCESuscriberAPI sceApi = new PRPC_SCESuscriberAPI("MyApi",
                                                       "10.1.1.1",10000);
// API initialization
java.util.Properties p = new java.util.Properties();
p.setProperty("prpc.client.output.machinemode.recordnum", 2048+"");

api.init(p);

// connect to the API
sceApi.connect();
```

Note The **init** method is called **before** the **connect** method.

Connecting to the SCE

After setting up the API, you should attempt to connect to the SCE. If the auto-reconnect feature is activated, the API will handle any disconnection from this point on.

To connect to the SCE, use the following methods:

```
public void connect() throws Exception
```

At any time during the API operation, you can check if the API is connected to the SCE by using the method **isConnected()** :

```
public boolean isConnected()
```

Note Every API instance supports a connection to exactly one SCE platform.

Information About getApiVersion

Syntax

```
public String getApiVersion()
```

Description

This method queries the API version. Version is a string formatted as <Major Version.Minor Version>.

API Finalization

To free the resources of both server and client, call the **disconnect** method:

```
public void disconnect()
```

The call to the disconnect method frees the resources in the SCE that manages the reliability of the connection from the SCE to the API. If the application is restarting and you do not want to lose any messages, do not use the disconnect method.

It is recommended that you use a **finally** statement in your main class. For example:

```
public static void main(String [] args) throws Exception
{
    PRPC_SCESubscriberApi sceapi = new PRPC_SCE_SubscriberApi ("myApi",
                                                               "sceHost");
    try
    {
              …
                 // Your code goes here
       }
       finally
       {
              sceapi.disconnect();
       }
}
```

Information About Indications Listeners

The SCE platform issues several types of indications when certain events occur. There are three types of indications:

- Login-pull indications
- Logout indications
- Quota indications

The indications are sent only if there are listeners that are registered to listen to those indications. For every type of indications, a separate listener may be registered. For descriptions about the events that trigger these indications, see the Application Programming Interface Events chapter.

Information About the LoginPullListener Interface Class

The **LoginPullListener** interface defines a set of callback functions that are used only in the **pull** model.

Policy Servers that are responsible for the Network ID management part of the Subscriber Provisioning process and intend to work in the pull model should register a LoginPullListener to enable to respond to the login-pull requests from the SCE and to synchronize the SCE platform.

To enable listening to those indications, the API allows a listener to be set for these types of indications:

```
public void registerLoginPullListener(LoginPullListener listener)
public void unregisterLoginPullListener(LoginPullListener listener)
```

Note The API supports one LoginPullListener at a time. Furthermore, it is strongly recommended not to have more than one API that has registered a LoginPullListener. This can lead to nonsynchronized SCE platforms if both SCEs respond to the same login-pull request.

LoginPullListener is an interface that is implemented to enable to register a login-pull indications listener. It is defined as follows:

```
public interface LoginPullListener
{
     public void loginPullRequest (String anonymousSubscriberID,
                                   NetworkID networkID)

     public void loginPullRequestBulk(NetworkAndSubscriberID_BULK  subs)

     public void getSubscribersBulkResponse(
                                NetworkAndSubscriberID_BULK  subs,
                                SubscriberBulkResponseIterator iterator)
}
```

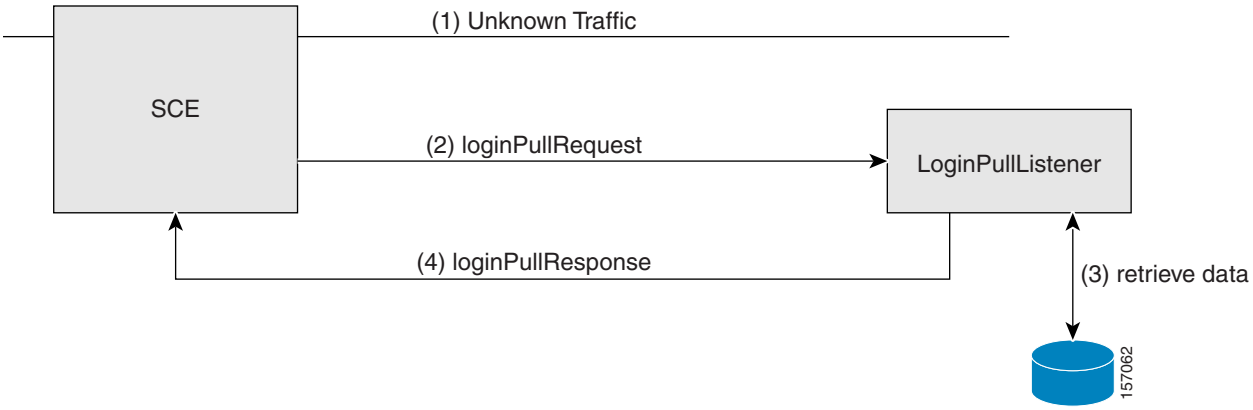
Information About the loginPullRequest Callback Method

When the SCE encounters an unknown IP address's subscriber-side traffic, it issues a request for the subscriber login information based on the IP address (see Pull Model, page 3-2). The SCE expects the policy server to respond with the configuration data of the subscriber data to which this IP was allocated.

This request is dispatched to the registered listener and triggers the **loginPullRequest** callback function. Upon this callback, the listener should retrieve the subscriber information of the subscriber matching this IP address and activate **loginPullResponse** to deliver the information to the SCE (see Information About the loginPullResponse Operation, page 5-21). If no information exists for this IP address, no response is issued.

The following diagram illustrates the **loginPullRequest** callback method:

Figure 5-1 LoginPullRequest Callback Method



Parameters

- **anonymousSubscriberID**—This anonymous subscriber ID **must** be supplied to the **loginPullResponse** operation (see Information About the loginPullResponse Operation). Also see the Anonymous Subscriber ID section.
- **networkID**—The network identifier of the unknown subscriber. See the Network ID section for more information.

Information About the loginPullRequestBulk Callback Method

This callback function is the bulk version of the **loginPullRequest** callback function that is described in the previous section.

Parameters

- **subs**—Contains pairs of NetworkIDs and anonymous IDs of several subscribers. See the Parameters section of the loginPullRequest callback method for more information.

The Policy Server can respond to this request by the loginPullBulkResponse method activation or by activating the loginPullResponse method for each NetworkID in the bulk. See Information About the loginPullResponseBulk Operation and Information About the loginPullResponse Operation. To iterate over the data contained in the subs parameter use the next() iteration method provided by the bulk class, see Bulk Iterator.

GetSubscribersBulkResponse Callback Method

This callback method is used during the SCE synchronization process in the **pull** model. For a detailed description, see Information About SCE-API Synchronization, page 5-31.

Information About the LogoutListener Interface Class

Policy Servers that are responsible for the Network ID management part of the Subscriber Provisioning process might want to register a LogoutListener to be notified when certain subscribers are actually removed from the SCE platform.

The API allows setting a **LogoutListener** to be able to receive logout indications.

```
public void registerLogoutListener(LogoutListener listener)
public void unregisterLogoutListener(LogoutListener listener)
```

Note The API supports one LogoutListener at a time.

The following sections describe callback functions of the **LogoutListener** interface.

- Information About the logoutIndication Callback Method, page 5-9
- Information About the logoutBulkIndication Callback Method, page 5-10

Information About the logoutIndication Callback Method

When the SCE platform identifies the logout of the last Network-ID of the subscriber identified by the subscriberID, it issues the logout indication. This triggers a call to the **logoutIndication** callback function of all registered logout indications listeners.

```
public void logoutIndication(String subscriberID)
```

Parameters

- **subscriberID**—A unique identifier of the subscriber. See Subscriber ID for more information. The SCE no longer handles this subscriber ID.

Information About the logoutBulkIndication Callback Method

When the SCE platform identifies the logout of the last NetworkID of the group of subscribers, it issues the logout bulk indication. This triggers a call to the **logoutBulkIndication** callback function of all registered logout indications listeners.

```
public void logoutBulkIndication(SubscriberID_BULK subs)
```

Parameters

- **subs**—Contains subscriber IDs of the subscribers that were logged out. See the SubscriberID_BULK Class section for more information.

Information About the QuotaListenerEx Interface Class

Note From version 3.0.5, the QuotaListener interface is deprecated and should be replaced with QuotaListenerEx. For backwards compatibility, the QuotaListener interface still exists, but you should use the QuotaListenerEx interface when integrating with version 3.0.5 of the API.

Policy Servers that are responsible for the Quota management operations in the Subscriber Provisioning Process should be able to receive quota-related indications issued by the SCE platform.

The API allows setting the **QuotaListener** to be able to receive quota indications.

```
public void registerQuotaListener(QuotaListener listener)
public void unregisterQuotaListener(QuotaListener listener)
```

Note The API supports one QuotaListener at a time.

Note The QuotaListener interface is used for backward compatibility, but it is recommended to pass an object that implements QuotaListenerEx.

The following sections describe the callback functions of the **QuotaListenerEx** interface.

Note The Bulk versions of the quota callback methods are not used in this release of the API.

Information About the quotaStatusIndication Callback Method

Quota status indication delivers the remaining value of the specified set of the quota buckets for a specific subscriber. This indication is issued by the SCE periodically or upon a call to the **getQuotaStatus** operation (see the Information About the getQuotaStatus Operation section) and is distributed to the registered listener by activating a **quotaStatusIndication** callback function.

```
public void quotaStatusIndication(String subscriberID,Quota quota)
```

Parameters

- **subscriberID**—The unique ID of the subscriber. See the Subscriber ID section for more information.
- **quota**—Quota of the subscriber. See Information About Subscriber Quota for more information.

Information About the quotaStatusBulkIndication Callback Method

Quota status bulk indication delivers the remaining value of the specified set of the quota buckets for a group of subscribers. This indication is issued by SCE periodically or upon a call to the **getQuotaStatusBulk** operation (see the Get Quota Status Event section) and is distributed to the registered listener by activating a **quotaStatusBulkIndication** callback function.

```
public void quotaStatusBulkIndication(Quota_BULK subs)
```

You can configure the period for periodically issued indications. For more information, see the *Cisco Service Control Application for Broadband User Guide*.

Parameters

- **subs**—Contains quota data of the bulk of the subscribers. See the Quota_BULK Class section for more information.

Information About the quotaBelowThresholdIndication Callback Method

When the quota of a subscriber drops below a pre-configured threshold, the SCE platform issues an indication that is distributed to the registered listener by activating a **quotaBelowThresholdIndication** callback function.

```
public void quotaBelowThresholdIndication(String subscriberID,Quota quota)
```

Parameters

- **subscriberID**—The unique ID of the subscriber. See the Subscriber ID section for more information.
- **quota**—Quota of the subscriber. See Information About Subscriber Quota for more information.

Information About the quotaBelowThresholdBulkIndication Callback Method

When the quota of a group of subscribers drops below a pre-configured threshold, the SCE platform issues an indication that is distributed to the registered listener by activating a **quotaBelowThresholdBulkIndication** callback function.

```
public void quotaBelowThresholdBulkIndication(Quota_BULK subs)
```

Parameters

- **subs**—Contains quota data of the bulk of the subscribers. See the Quota_BULK Class section for more information.

Information About the quotaDepletedIndication Callback Method

When the quota of a subscriber is depleted, the SCE platform issues an indication that is distributed to the registered listener by activating a **quotaDepletedIndication** callback function.

```
public void quotaDepletedIndication(String subscriberID, Quota quota)
```

Parameters

- **subscriberID**—The unique ID of the subscriber. See the Subscriber ID section for more information.
- **quota**—Quota of the subscriber. See Information About Subscriber Quota for more information.

Information About the quotaDepletedBulkIndication Callback Method

When the quota of a group of subscribers is depleted, the SCE platform issues an indication that is distributed to the registered listener by activating a **quotaDepletedBulkIndication** callback function.

```
public void quotaDepletedBulkIndication (SubscriberID_BULK subs)
```

Parameters

- **subs**—Contains names of the subscribers whose quota was depleted. See the SubscriberID_BULK Class section for more information.

Information About the quotaStateRestore Callback Method

When a subscriber logs in to the policy server, the policy server performs a login operation to the SCE. The SCE issues a request to the policy server to restore the subscriber quota in the SCE by activating a **quotaStateRestore** callback function. The policy server should respond to this function with a Quota Update Event.

```
public void quotaStateRestore(String subscriberID,Quota quota)
```

Parameters

- **subscriberID**—The unique ID of the subscriber. See the Subscriber ID section for more information.
- **quota**—Quota of the subscriber. See Information About Subscriber Quota for more information. The bucket IDs array is of size 0 because when this indication is created all the quota buckets are empty.

Information About the quotaStateBulkRestore Callback Method

When a group of subscribers log in to the policy server, the policy server performs a login operation to the SCE. The SCE issues a request to the policy server to restore the subscriber quota in the SCE by activating a **quotaStateBulkRestore** callback function. The policy server should respond to this function with a Quota Update Event.

```
public void quotaStateBulkRestore(SubscriberID_BULK subs)
```

Parameters

- **subs**—Contains names of the subscribers whose quota was depleted. See the SubscriberID_BULK Class section for more information.

Information About Connection Monitoring

The SCMS SCE Subscriber API monitors the connection to the SCE platform. A Policy Server requesting to perform certain operations on connection establishment or disconnection from the SCE can implement a ConnectionListener interface.

- ConnectionListener Interface, page 5-13
- Disconnect Listener: Example, page 5-13

ConnectionListener Interface

The API allows setting a connection listener.

```
setConnectionListener(ConnectionListener listener)
```

The connection listener is an interface that is defined as follows:

```
public interface ConnectionListener {

    /**
     * called when the connection with the SCE is down.
     */
    public void connectionIsDown();

    /**
     * called when the connection with the SCE is established.
     */
    public void connectionEstablished();
}
```

The connection establishment callback is used to start the SCE synchronization. See Information About SCE-API Synchronization for more information.

Disconnect Listener: Example

This example is a simple implementation of a disconnect listener that prints a message to **stdout** and returns.

```
import com.scms.api.sce.ConnectionListener;

public class MyConnectionListener implements ConnectionListener {

    public void connectionIsDown(){
        System.out.println("Message: connection is down.");
        return;
    }

    public void connectionEstablished(){
        System.out.println("Message: connection is established.");
        // activate thread that starts SCE synchronization
    }
}
```

Information About SCE Cascade Topology Support

The SCMS SCE Subscriber API supports SCE cascade topologies. A Policy Server connected to a cascade SCE platform is required to know which of the SCEs in the cascade setup is active and which is standby. The Policy Server should send logon operations **only** to the active SCE. Similarly, the Policy Server should perform subscriber synchronization with **only** the active SCE.

The standby SCE learns about the subscribers from the active SCE, which allows stateful fail-over. The Policy Server should be able to identify a fail-over event and synchronize the SCE that became active so that it will receive the most updated subscriber information.

In order to know which SCE is active, the Policy Server can implement a RedundancyStateListener interface.

- isRedundancyStatusActive Method, page 5-14
- Information About the RedundancyStateListener Interface, page 5-14
- Configuring the SCE to Ignore Cascade Violation Errors, page 5-15

isRedundancyStatusActive Method

The API provides the **isRedundnacyStatusActive** method in conjunction with the RedundancyStateListener interface in order to monitor the SCE redundancy status.

```
public boolean isRedundancyStatusActive()
```

The return value from this method has the following meaning:

- TRUE—If the SCE current status is active.
- FALSE—Otherwise.

It is recommended to use this method when first connecting to the cascade SCE in order to verify whether the SCE is active and prior to sending any logon operation to the SCE.

Information About the RedundancyStateListener Interface

In order to be able to monitor cascade SCE state changes, the API allows setting a redundancy state listener.

```
setRedundancyStateListener(RedundancyStateListener listener)
```

The redundancy state listener defines a callback method that is called when the cascade SCE redundancy status changes from active to standby and vice versa.

The redundancy state listener is an interface that is defined as follows:

```
public interface RedundancyStateListener {
    public void redundancyStateChanged(SCESubscriberApi sceApi,
                                       boolean isActive);
}
```

Note The Policy Server should perform a synchronization procedure on the SCE that became active. This should be similar to the procedure that is performed by the Policy Server on connection establishment to the SCE.

Note The API provides a connection to one SCE platform for each API instance. Therefore, for cascade setups, two SCE Subscriber API instances are required.

Parameters

- **sceApi**—The API instance that represents the SCE whose status changed. This parameter enables you to implement one listener for several SCEs.
- **isActive**—TRUE if the SCE became active. FALSE if the SCE became non-active.

Configuring the SCE to Ignore Cascade Violation Errors

The SCE 3.1.0 is configured by default to return an error when a logon operation is performed on a standby SCE. Use the `ignore-cascade-violation` CLI on the SCE in order to change this behavior.

To configure the SCE to ignore the cascade violation, use the following CLI on the SCE platform:

```
(config)#>management-agent sce-api ignore-cascade-violation
```

To view whether the the cascade violation is ignored, use the following CLI on the SCE platform:

```
#>show management-agent sce-api
```

To configure the SCE to issue the errors in case of the cascade violation, use the following CLI on the SCE platform:

```
(config)#>no management-agent sce-api ignore-cascade-violation
```

To configure the flag to the default value (to issue errors in case of the cascade violation) use the following CLI on the SCE platform:

```
(config)#>default management-agent sce-api ignore-cascade-violation
```

Note It is recommended to configure the SCE to ignore cascade violation only for backward compatibility with the existing SCE API code. In order to fully utilize the cascade feature, the SCE redundancy status should be monitored and used.

Information About Result Handling

The API enables setting a result handler for **every operation** allowing handling operations results in a different manner.

The OperationResultHandler interface's handleOperationResult callback is called when a result of an operation, which ran on the SCE, returns to the API.

If no result handling is required for a specific operation, insert **null** in the **handler** argument.

Note The same operation result handler can be passed to all operations.

Information About the OperationResultHandler Interface

This interface is implemented to receive results of operations performed through the API.

The operation result handler is called with the following single method:

```
public interface OperationResultHandler {

    /**
     * handle a result
     */
    public void handleOperationResult(Object[] result,
                                      OperationArguments handback);

}
```

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

Note The OperationResultHandler interface is the only way to retrieve results. The results cannot be returned immediately after the API method has returned to the caller. To enable to receive operation results, set the result handler of each operation at the time of the operation call (as displayed in the examples).

The following is the data returned from the OperationResultHandler interface:

- **result**—The actual result of the operation - each entry within the array can be one of the following:
  - **NULL**—indicates success of the operation.
  - **OperationException**—indicates operation failure (see below). For non-bulk operations, the result array will have only one entry.
- For bulk operations, each entry of the result array corresponds to the relevant entry in the bulk operation.
- **handback**—The API automatically provides this object to every operation call. It contains the information about the operation that was called, including all arguments that were passed at the time of the call. The input arguments of the operation are retrieved by the argument name in the API documentation. For example, this data can be used to inspect/output the parameters after the operation failed or to repeat the operation call.

Note In operations involving bulk objects, even if the operation fails for any specific element in the bulk, the processing of the bulk will continue until the end of the bulk.

Information About the OperationArguments Class

Use the following methods to retrieve the operation name:

```
public String getOperationName()
```

Use the following methods to retrieve the arguments names:

```
public String[] getArgumentNames()
```

Use the following method to retrieve the specific operation argument. Use the operation's arguments' names from the operation signature as an argument:

```
public Object getArgument(String name)
```

Examples

Sample implementation of the OperationResultHandler interface:

```
public class MyOperationHandler implements OperationResultHandler
{
  long sucessCounter = 0;
  long errorCounter = 0;

  public void handleOperationResult(Object[] result,
                                    OperationArguments handback)
  {
     for (int index=0; index < result.length; index++)
     {
       if (result[index]==null)
       {
          // success
          successCounter++;
       }
       else
       {
```

```
            // failure
            errorCounter++;

            // Extract error details
            OperationException ex = (OperationException)result[index];

            // Extract operation name
            String operationName = handback.getOperationName();

            // Print operation name and error message
            System.out.println("Error for operation "+
                               operationName +":" +
                               ex.getErrorMessage());
            // Print operation arguments
            String[] argNames = handback.getArgumentNames();
            if (argNames!=null)
            {
               for (int j=0; j<argNames.length; j++)
               {
                 System.out.println(argNames[j]+  "="+
                       handback.getArgument(argNames[j]));
               }
            }
         }
       }
    }
}
```

Note The above sample implementation can be used for both regular and bulk operations.

The following example demonstrates login operation sample result handler:

```
public class LoginOperationHandler implements OperationResultHandler
{

    public void handleOperationResult(Object[] result,
                                      OperationArguments handback)
    {
      for (int index=0; index < result.length; index++)
      {
        if (result[index]!=null)
        {

          // failure

          // Extract error details
          OperationException ex = (OperationException)result[index];

          // Print operation name and error message
          System.out.println("Error for login operation "+":" + ex.getErrorMessage());

          // Print subscriber ID parameter value
          System.out.println("subscriberID"+handback.getArgument("subscriberID"));
        }
      }
    }
}
```

OperationException Class

The com.scms.api.sce.OperationException Java class provides all of the functional errors of the SCMS SCE Subscriber API, which is contrary to the normal Java usage. This "contrary" approach was chosen because of the required "cross-language and cross-protocol" nature of the SCMS SCE Subscriber API, which should allow all future SCE API implementations to appear the same (Java, C, C++). Each OperationException exception provides the following information:

- A unique error code (**long**)
- An informative message (**java.lang.String**)
- A server-side stack trace (**java.lang.String**)

See List of Error Codes, page A-1for more details about error codes and their meaning.

Information About Subscriber Provisioning Operations

This section lists the methods of the API that can be used for Subscriber Provisioning purposes. The signature of each method is followed by a description of its input parameters and its return values.

All the methods return a **java.lang.IllegalStateException** when called before a connection with the SCE is established.

Information About the login Operation

Syntax

```
void login(String subscriberID,
           NetworkID networkID,
           boolean networkIdAdditive,
           PolicyProfile policy,
           QuotaOperation quotaOperation,
           OperationResultHandler handler) throws Exception
```

Description

This operation adds or updates the subscriber to the SCE. The operation is performed according to the following algorithm:

- If the subscriber ID does not exists in the SCE, a new subscriber is added with all the data supplied
- If the subscriber ID exists:
  - If the **networkIdAdditive** flag is set to TRUE, the supplied NetworkID is added to the existing networkIDs of the subscriber. Otherwise, the supplied networkID replaces the existing networkIDs.
  - **policy**—Policy is *updated* with the new policy values. Subscriber Policy entries that are not provided within the PolicyProfile remain unchanged or created with default values.
  - **quota**—The quota is *updated* according to the bucket values and the operations provided, see Information About Subscriber Quota, page 4-5.
- If there is a networkID congestion with another subscriber, the networkID of the other subscriber is logged out implicitly and the new subscriber is logged in.

For relevant events description, see Push Model, page 3-2.

Parameters

subscriberID—The unique ID of the subscriber. See the Subscriber ID section for the subscriber ID format description.

networkID—The network identifier of the subscriber. See Information About Network ID Mappings for more information.

networkIDAdditive—If this flag is set to TRUE, the supplied NetworkID is added to the existing networkIDs of the subscriber. Otherwise, the supplied networkID replaces the existing networkIDs.

policy—Policy profile of the subscriber. See Information About SCA BB Subscriber Policy Profile for more information.

quota—Quota of the subscriber. See Information About Subscriber Quota for more information.

handler—Result handler for this operation. See Information About Result Handling for a description of the *OperationResultHandler* interface.

Error Codes

The following is the list of error codes that this method might return:

- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_RESOURCE_SHORTAGE
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

For a description of error codes, see List of Error Codes, page A-1.

Examples

This example adds the IP address 192.168.12.5 to an existing subscriber named *john* without affecting any existing mappings:

```
login(
      "john",                           // subscriber name
      new NetworkID(new String[]{"192.168.12.5"},
      SCESubscriberApi.ALL_IP_MAPPINGS),
      true,                             // isMappingAdditive is true
      null,                             // no policy
      null);                            // no quota
```

This example adds the IP address 192.168.12.5 overriding previous mappings:

```
login(
      "john",                           // subscriber name
      new NetworkID(new String[]{"192.168.12.5"},
      SCESubscriberApi.ALL_IP_MAPPINGS),
      false,                            // isMappingAdditive is false
      null,                             // no policy
      null);                            // no quota
```

For more examples, see the Login and Logout section.

Information About the loginBulk Operation

Syntax

```
void loginBulk(Login_BULK subsBulk,
               OperationResultHandler handler) throws Exception
```

Description

This operation applies the logic described in the login operation for each subscriber in the bulk.

Parameters

subsBulk—See the Login_BULK Class section.

handler—Result handler for this operation. See Information About Result Handling for a description of the *OperationResultHandler* interface.

Error Codes

The following is the list of error codes that this method might return:

- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_RESOURCE_SHORTAGE
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

For a description of error codes, see List of Error Codes, page A-1.

Information About the loginPullResponse Operation

Syntax

```
void loginPullResponse(String subscriberID,
                       String anonymousSubscriberID,
                       NetworkID networkID,
                       PolicyProfile policy,
                       QuotaOperation quota,
                       OperationResultHandler handler) throws Exception
```

Description

This operation sends subscriber login information to the SCE as a response to a **loginPullRequest** call from the SCE or a **loginPullBulkRequest**.

For relevant events description, see Pull Model, page 3-2.

Parameters

subscriberID—The unique ID of the subscriber. See the Subscriber ID section for the subscriber ID format description.

anonymousSubscriberID—The identifier of the anonymous subscriber. This is sent by the SCE within the loginPullRequest/loginPullBulkRequest indication (see Information About the LoginPullListener Interface Class). See the Anonymous Subscriber ID section for more information.

networkID—The network identifier of the subscriber. See Information About Network ID Mappings for more information. This must include the network ID received by the loginPullRequest. If this subscriber in the SCE already has other network IDs, this network ID is added to the existing ones.

policy—Policy profile of the subscriber. See Information About SCA BB Subscriber Policy Profile for more information.

quota—Quota of the subscriber. See Information About Subscriber Quota for more information.

handler—Result handler for this operation. See Information About Result Handling for a description of the *OperationResultHandler* interface.

Error Codes

The following is the list of error codes that this method might return:

- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_RESOURCE_SHORTAGE
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

For a description of error codes, see List of Error Codes, page A-1.

Information About the loginPullResponseBulk Operation

Syntax

```
void loginPullResponseBulk(LoginPullResponse_BULK subsBulk,
                           OperationResultHandler handler) throws Exception
```

Description

This operation applies the logic described in loginPullResponse operation for each subscriber in the bulk.

For relevant events description, see Pull Model, page 3-2.

Parameters

subsBulk—See the LoginPullResponse_BULK Class section.

handler—Result handler for this operation. See Information About Result Handling for a description of the **OperationResultHandler** interface.

Error Codes

The following is the list of error codes that this method might return:

- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_RESOURCE_SHORTAGE
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

For a description of error codes, see List of Error Codes, page A-1.

Information About the logout Operation

Syntax

```
void logout(String subscriberID,
            NetworkID networkID,
            OperationResultHandler handler) throws Exception
```

Description

This operation removes the specified networkID of the subscriber from the SCE. If this is the last networkID of the specified subscriber, the subscriber is removed from the SCE. If no subscriber ID is specified, the supplied network ID is removed from the SCE regardless to which subscriber this network ID belongs. If no network ID is supplied, all Network IDs of this subscriber are removed.

If the subscriber record is not in the SCE, the logout operation will succeed.

For relevant events description, see Logout Events, page 3-3.

Parameters

subscriberID—The unique ID of the subscriber. See the Subscriber ID section for the subscriber ID format description.

networkID—The network identifier of the subscriber. See Information About Network ID Mappings for more information.

handler—Result handler for this operation. See Information About Result Handling for a description of the **OperationResultHandler** interface.

Error Codes

The following is the list of error codes that this method might return:

- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED

For a description of error codes, see List of Error Codes, page A-1.

Information About the logoutBulk Operation

Syntax

```
void logoutBulk(NetworkAndSubscriberID_BULK subsBulk,
                OperationResultHandler handler) throws Exception
```

Description

This operation applies the logic described in logout operation for each subscriber in the bulk.

For relevant events description, see Logout Events, page 3-3.

Parameters

subsBulk—See the NetworkAndSubscriberID_BULK Class section.

handler—Result handler for this operation. See Information About Result Handling for a description of the **OperationResultHandler** interface.

Error Codes

The following is the list of error codes that this method might return:

- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED

For a description of error codes, see List of Error Codes, page A-1.

Information About the networkIdUpdate Operation

Syntax

```
void networkIDUpdate(String subscriberID,
                     NetworkID networkID,
                     boolean networkIdAdditive,
                     OperationResultHandler handler) throws Exception
```

Description

This operation adds or replaces an existing subscriber's network ID.

Note This operation is effective only if the subscriber record exists in the SCE. Otherwise, the operation will fail.

For relevant events description, see Network ID Update Event, page 3-4.

Parameters

subscriberID—The unique ID of the subscriber. See the Subscriber ID section for the subscriber ID format description.

networkID—The network identifier of the subscriber. See Information About Network ID Mappings for more information.

networkIDAdditive—If this flag is set to TRUE, the supplied NetworkID is added to the existing networkIDs of the subscriber. Otherwise, the supplied networkID replaces the existing networkIDs.

Error Codes

The following is the list of error codes that this method might return:

- ERROR_CODE_SUBSCRIBER_NOT_EXIST
- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_RESOURCE_SHORTAGE
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

For a description of error codes, see List of Error Codes, page A-1.

Information About the networkIdUpdateBulk Operation

Syntax

```
void networkIDUpdateBulk(NetworkAndSubscriberID_BULK subsBulk,
                         OperationResultHandler handler) throws Exception
```

Description

This operation applies the logic described in networkIDUpdate operation for each subscriber in the bulk.

For relevant events description, see Network ID Update Event, page 3-4.

Parameters

subsBulk—See the NetworkAndSubscriberID_BULK Class section.

handler—Result handler for this operation. See Information About Result Handling for a description of the **OperationResultHandler** interface.

Error Codes

The following is the list of error codes that this method might return:

- ERROR_CODE_SUBSCRIBER_NOT_EXIST
- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_RESOURCE_SHORTAGE
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

For a description of error codes, see List of Error Codes, page A-1.

Information About the profileUpdate Operation

Syntax

```
void profileUpdate(String subscriberID,
                   PolicyProfile policy,
                   OperationResultHandler handler) throws Exception
```

Description

This operation modifies an existing subscriber's policy profile. If the subscriber record does not exist in the SCE, this operation will fail.

For relevant events description, see Profile Update Event, page 3-4.

Parameters

subscriberID—The unique ID of the subscriber. See the Subscriber ID section for the subscriber ID format description.

policy—Policy profile of the subscriber. See Information About SCA BB Subscriber Policy Profile for more information.

handler—Result handler for this operation. See Information About Result Handling for a description of the **OperationResultHandler** interface.

Error Codes

The following is the list of error codes that this method might return:

- ERROR_CODE_SUBSCRIBER_NOT_EXIST
- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

For a description of error codes, see List of Error Codes, page A-1.

Information About the profileUpdateBulk Operation

Syntax

```
void profileUpdateBulk(PolicyProfile_BULK subsBulk,
                       OperationResultHandler handler) throws Exception
```

Description

This operation applies the logic described in profileUpdate operation for each subscriber in the bulk.

For relevant events description, see Profile Update Event, page 3-4.

Parameters

subsBulk—See the PolicyProfile_BULK Class section.

handler—Result handler for this operation. See Information About Result Handling for a description of the **OperationResultHandler** interface.

Error Codes

The following is the list of error codes that this method might return:

- ERROR_CODE_SUBSCRIBER_NOT_EXIST
- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

For a description of error codes, see List of Error Codes, page A-1.

Information About the quotaUpdate Operation

Syntax

```
void quotaUpdate(String subscriberID,
                 QuotaOperation quotaOperation,
                 OperationResultHandler handler) throws Exception
```

Description

This operation performs an operation of updating the subscriber's quota.

For relevant event description, see Quota Update Event, page 3-5.

Parameters

subscriberID—The unique ID of the subscriber. See the Subscriber ID section for the subscriber ID format description.

quotaOperation—Quota operation to perform on the quota of the subscriber. See Information About Subscriber Quota for more information.

handler—Result handler for this operation. See Information About Result Handling for a description of the **OperationResultHandler** interface.

Error Codes

The following is the list of error codes that this method might return:

- ERROR_CODE_SUBSCRIBER_NOT_EXIST
- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

For a description of error codes, see List of Error Codes, page A-1.

Information About the quotaUpdateBulk Operation

Syntax

```
void quotaUpdateBulk(QuotaOperation_BULK subsBulk,
                     OperationResultHandler handler) throws Exception
```

Description

This operation applies the logic of the quotaUpdate operation on each subscriber in the bulk.

For relevant event description, see Quota Update Event, page 3-5.

Parameters

subsBulk—See the QuotaOperation_BULK Class section.

handler—Result handler for this operation. See Information About Result Handling for a description of the **OperationResultHandler** interface.

Error Codes

The following is the list of error codes that this method might return:

- ERROR_CODE_SUBSCRIBER_NOT_EXIST
- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

For a description of error codes, see List of Error Codes, page A-1.

Information About the getQuotaStatus Operation

Syntax

```
void getQuotaStatus(String subscriberID,
                    Quota quota,
                    OperationResultHandler handler) throws Exception
```

Description

This operation places the request to query the current remaining quota amount of the specified set of quota buckets. The getQuotaStatusIndication including the queried data follows this request (**asynchronously**). See Information About the quotaStatusIndication Callback Method, page 5-10.

For relevant events description, see Get Quota Status Event, page 3-5.

Parameters

subscriberID—The unique ID of the subscriber. See the Subscriber ID section for the subscriber ID format description.

quota—Includes the list of names (without values) of the quota buckets to retrieve. See Information About Subscriber Quota for more information about how to construct with buckets' names only.

handler—Result handler for this operation. See Information About Result Handling for a description of the **OperationResultHandler** interface.

Error Codes

The following is the list of error codes that this method might return:

- ERROR_CODE_SUBSCRIBER_NOT_EXIST
- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

For a description of error codes, see List of Error Codes, page A-1.

Information About the getQuotaStatusBulk Operation

Syntax

```
void getQuotaStatuBulk(Quota_BULK subsBulk,
                       OperationResultHandler handler) throws Exception
```

Description

This method is a bulk version of the getQuotaStatus method described in the previous section.

For relevant events description, see Get Quota Status Event, page 3-5.

Parameters

subsBulk—See the Quota_BULK Class section.

handler—Result handler for this operation. See Information About Result Handling for a description of the **OperationResultHandler** interface.

Error Codes

The following is the list of error codesthat this method might return:

- ERROR_CODE_SUBSCRIBER_NOT_EXIST
- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

For a description of error codes, see List of Error Codes, page A-1.

Information About SCE-API Synchronization

In cases when the SCE and the Policy Server have a conflict in the data about a subscriber because of disconnection, loss of logon messages, or reboot, several problems can arise. These problems can cause:

- A misclassification of one subscriber's traffic as if it was another subscriber.
- Enforcement of the wrong service on the subscriber's traffic.
- A loss of resources.

It is possible to prevent such conflicts by keeping the communication channels as reliable as possible by performing synchronization of the subscribers' data between the SCE and the Policy Server using the API. The Policy Server, by using the API, is **always** the initiator of the synchronization.

Caution Performing the synchronization process from several Policy Servers at the same time will cause the subscriber information in the SCE to be inconsistent with all servers.

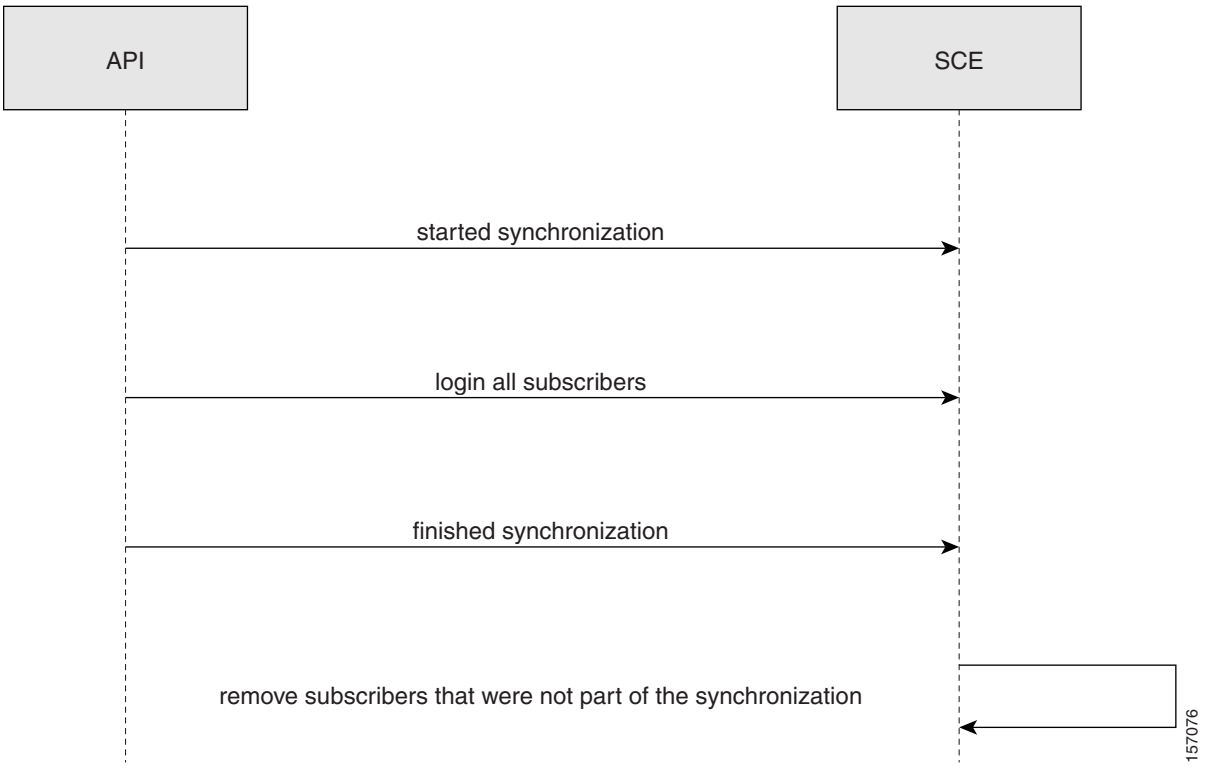
The following list describes the synchronization guidelines the Policy Server must adhere to while implementing synchronization:

- Information About the Push Model Synchronization Procedure, page 5-32
- Information About the Pull Model Synchronization Procedure, page 5-34

Information About the Push Model Synchronization Procedure

1. The Policy Server indicates to the SCE that it is starting to synchronize the SCE.
2. The Policy Server logs-in all of the subscribers the SCE should handle. Preferably, the login operations are performed in bulks.
3. The Policy Server notifies the SCE that the synchronization has ended.
4. The SCE removes all of the subscriber data that was not part of the synchronization process.

Figure 5-2 Push Model Synchronization Procedure



Note During the synchronization process, the regular logon operations can be performed.

The following sections describe the methods provided for use for the synchronization procedure in the *push* model.

Information About synchronizePushStart

Syntax

```
void synchronizePushStart(OperationResultHandler handler)
```

Description

Use this operation in the *push* model only to signal the SCE that synchronization with the server is about to begin. The SCE marks all of the subscriber data with a “dirty-bit”, which is reset if this data is re-applied as part of the synchronization process. Every call to this method restarts the synchronization process.

Parameters

handler—Result handler for this operation. See Information About Result Handling for a description of the **OperationResultHandler** interface.

Information About synchronizePushEnd

Syntax

```
void synchronizePushEnd(boolean success, OperationResultHandler handler)
```

Description

Use this operation in the *push* model only to signal the SCE that synchronization with the server has ended. The SCE scans the entire subscriber database for data with the “dirty-bit” assigned at **synchronizePushStart** and removes it.

Parameters

success—A flag indicating that the synchronization was successful to the SCE.

handler—Result handler for this operation. See Information About Result Handling for a description of the *OperationResultHandler* interface.

Information About the Pull Model Synchronization Procedure

1. The Policy Server indicates to the SCE that it is starting to synchronize the SCE
2. The Policy Server retrieves from the SCE all of the subscribers IDs and network-IDs it is currently handling
3. The Policy Server fixes any mis-synchronization.

Algorithm

Use the following algorithm template when planning the synchronization procedure:

For each retrieved subscriber (<SubscriberID, IP address>):

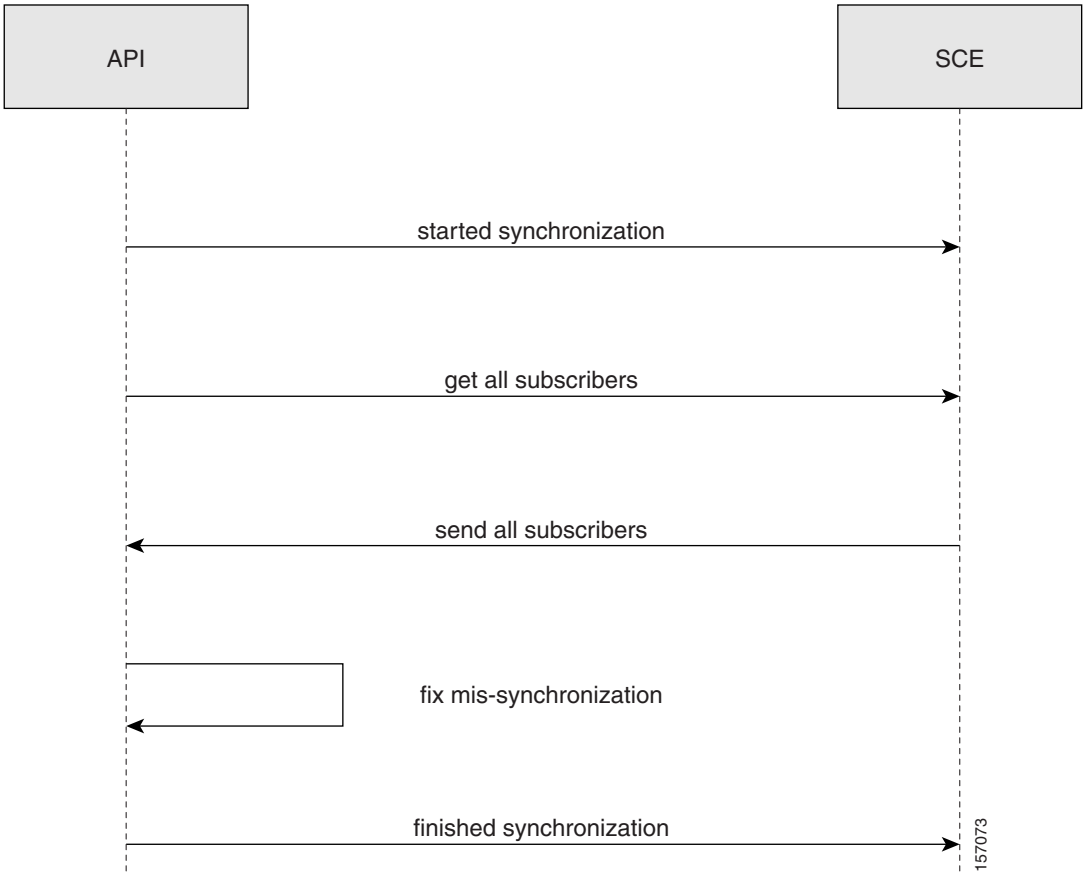
- If <SubscriberID, IP address> exists in the Policy Server database:

  send a policy profile and networkID update to the SCE
- Otherwise:

  send a logout with the Subscriber IP to the SCE

Steps 2 and 3 are performed as a bulk at one time.

Figure 5-3 Pull Model Synchonization Procedure



Note During the synchronization process, the regular logon operations can be performed.

The following sections describe the methods provided for use for the synchronization procedure in the *pull* model.

Information About synchronizePullStart

Syntax

```
void synchronizePullStart(OperationResultHandler handler)
```

Description

Use this operation in the *pull* model only to signal the SCE that synchronization with the server should be started.

Parameters

handler—Result handler for this operation. See Information About Result Handling for a description of the **OperationResultHandler** interface.

Information About synchronizePullEnd

Syntax

```
void synchronizePullEnd(boolean success, OperationResultHandler handler)
```

Description

Use this operation in the *pull* model only to signal the SCE that synchronization with the server has ended.

Parameters

handler—Result handler for this operation. See Information About Result Handling for a description of the **OperationResultHandler** interface.

success—A flag to the SCE indicating that the synchronization was successful.

Information About getSubscribersBulk

Syntax

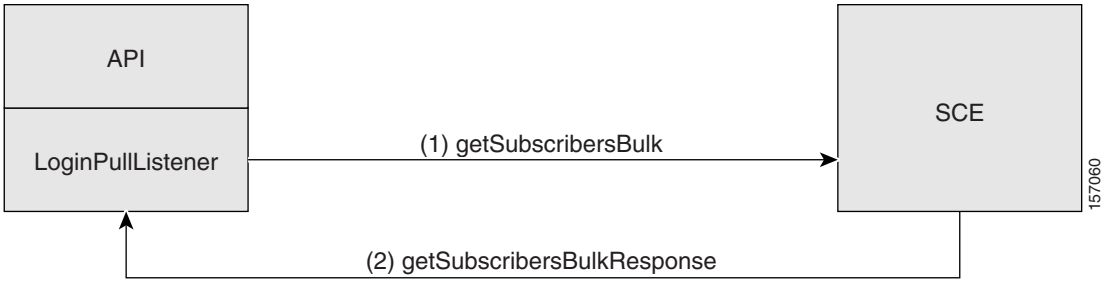
```
void getSubscribersBulk(int bulkSize,
                        SubscribersBulkResponseIterator iterator,
                        OperationResultHandler handler)
```

Description

Use this operation in the *pull* model synchronization process to retrieve a bulk of subscribers the SCE is currently handling (see Information About the Pull Model Synchronization Procedure, page 5-34).

Upon receiving this request (**getSubscribersBulk**), the SCE asynchronously issues the **getSubscribersBulkResponse** indication containing subscriberIDs and corresponding NetworkIDs (see the Information About the LoginPullListener Interface Class section). This method supplies an iterator that is passed to the next call of **getSubscribersBulk**. To signal the end of iterations, the iterator of the last bulk is null.

Figure 5-4 Get Subscribers Bulk Description



Parameters

bulkSize—The size of the bulk to retrieve. Maximum bulk size is limited to 100 entries.

iterator—Iterator of the subscribers at the SCE side. This iterator is received in getSubscribersBulkResponseIndication and it should be passed to the next call to getSubscribersBulk method. When calling the getSubscribersBulk method for the first time, use **null** as an iterator (using **null** indicates that you want to start from the beginning).

handler—Result handler for this operation. See Information About Result Handling for a description of the **OperationResultHandler** interface.

Information About Advanced API Programming

Implementing High-Availability

High-availability support provided by the API assumes that the high-availability scheme of the policy server is a type of two-node cluster where only one server is active at any given time. The other server (standby) is not connected to the SCE.

When the active server fails, it is the responsibility of the user's two-node cluster scheme to perform a fail-over to the standby server.

Note **High-availability can be implemented separately for every policy server provisioning the SCE platform at the same time.**

In order to implement high-availability with the SCMS SCE Subscriber API, you must do the following:

- Set up a two-node cluster for two policy servers.
- Construct two API instances *with the same API name* each one on the different server (node) within the cluster (For constructors description, see the API Construction section). During cluster runtime, only one API instance should be connected to the SCE platform. When a fail-over occurs, the failed server should disconnect from the SCE and the standby server should become active and re-connect to the SCE within the pre-defined timeout (see the Configuring the API Disconnection Timeout section). Because of identical API names, the SCE will behave as if the same API was re-connected and no information will be lost.

Note Do not call the **unregisterXXXListener** methods implicitly in the API used on the failed policy server as this will cause the loss of data. Calling the **disconnect()** method does not unregister the listeners.

API Code Examples

This section gives several code examples for the API usage:

- Login and Logout, page 5-38
- Login-pull Request and login-pull Response, page 5-41

Login and Logout

The following example logs in a predefined number of subscribers to the SCE, and then logs them out. This example uses autoreconnect support; therefore, it does not define a connection listener.

The following code outline contains a sample implementation of a **result handler** that counts success and failure results:

```
// Class responsible for operations result handling
import com.scms.api.sce.OperationArguments;
import com.scms.api.sce.OperationException;
import com.scms.api.sce.OperationResultHandler;

public class MyOperationResultHandler implements OperationResultHandler
{
    long count = 0;

    public void handleOperationResult(Object[] result, OperationArguments handback)
    {
        for (int index=0; index < result.length; index++)
        {
            count++;
            if (result[index]==null)
            {
                //print success every 100 operations
                //if (++count%100 == 0)
                {
                    System.out.println("\tsuccess "+count);
                }
            }
            else // error - print every error
            {
                // failure
                count++;
                // Extract error details
                OperationException ex = (OperationException)result[index];

                // Extract operation name
                String operationName = handback.getOperationName();

                // Print operation name and error message
                System.out.println("Error for operation "+
                                   operationName+": "+
                                   ex.getMessage());
            }
        }
    }
    public synchronized void waitForLastResult(int lastResult)
    {
        while (count<lastResult)
        {
            try
            {
                wait(100);
            }
            catch (InterruptedException ie)
            {
                ie.printStackTrace();
            }
        }
    }
}
```

Class that contains a simple LogoutListener implementation that counts the number of received logout indications:

```
import com.scms.api.sce.LogoutListener;
import com.scms.common.NetworkAndSubscriberID_BULK;
import com.scms.common.SubscriberID_BULK;

class MyLogoutListener implements LogoutListener
{
    long count = 0;

    public void logoutIndication(String subscriberID)
    {
        increaseCounter(1);
    }

    synchronized void increaseCounter(long value)
    {
        count = count + value;
    }

    synchronized long getCounter()
    {
        return 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 void logoutBulkIndication(SubscriberID_BULK subs)
    {
        increaseCounter(subs.getSize());
    }
}
```

Class that contains the **main** method:

```
import com.scms.api.sce.prpc.PRPC_SCESubscriberApi;
import com.scms.common.*;

public class LogonPolicyServer {

    public static void main (String args[]) throws Exception
    {
        int numSubscribersToLogin = 500;

        //instantiate an API with reconnect interval of 5 seconds
        PRPC_SCESubscriberApi api = new PRPC_SCESubscriberApi(
                                                "myAPI",
                                                args[0], // IP of the SCE
                                                5000);
```

```
        try {
            // instantiate operation result handler
            // we will use one handler for all operations
            MyOperationResultHandler resultHandler = new MyOperationResultHandler();

            // instantiate logout listener
            MyLogoutListener listener = new MyLogoutListener();

            // register to logout indications
            api.registerLogoutListener(listener);

            // connect to the SCE
            api.connect();

            //login
            System.out.println("login of "+numSubscribersToLogin+" subscribers");

            PolicyProfile pp = new PolicyProfile(new String[]{"packageId=1","monitor=1"});

            for (int i=0; i<numSubscribersToLogin; i++)
            {
                api.login("sub"+i,
                          new NetworkID(getMappings(i), // generate ip
                          NetworkID.ALL_IP_MAPPINGS),
                          true,                         // additive flag
                          pp,                           // policy
                          null,                         // no quota
                          resultHandler);
            }
            // wait for subscribers to log in
            resultHandler.waitForLastResult(numSubscribersToLogin);

            // logout all subscribers
            System.out.println("logout of "+numSubscribersToLogin+" subscribers");

            for (int i=0; i<numSubscribersToLogin; i++)
            {
                NetworkID nid = new NetworkID(getMappings(i), NetworkID.ALL_IP_MAPPINGS);

                api.logout("sub"+i,nid,resultHandler);
            }

            // wait for all subscribers to be logged out -
            // but this time use
            // logout listener to count the results
            listener.waitForLastResult(numSubscribersToLogin);
        }
        finally
        {
            api.unregisterLogoutListener
            api.disconnect();
        }
    }

    //'automatic' mapping generator for the sample program
    private static String[] getMappings(int i) {
        return new String[]{ "10." +((int)i/65536)%256 + "." +
            ((int)(i/256))%256 + "." + (i%256)};
    }
}
```

Login-pull Request and login-pull Response

The following code fragment demonstrates a login-pull request and login-pull response manipulations:

This class is a sample implementation of the listener for the logout and login pull indications:

```
import java.util.Iterator;

// result handler from the previous example
import MyOperationResultHandler;

import com.scms.api.sce.*;
import com.scms.common.*;

class MyListener implements LoginPullListener, LogoutListener
{
    // indications counters
    long logoutCount = 0;
    long pullCount=0;

    // api instance – used to send login-pull responses to the SCE
    PRPC_SCESubscriberApi api = null;

    // construct operation handler –
    // from previous (Login and Logout) example
    MyOperationResultHandler h = new MyOperationResultHandler();

    public MyListener(PRPC_SCESubscriberApi api)
    {
        this.api = api;
    }

    // Increase logout counter
    public void logoutIndication(String subscriberID)
    {
        increaseLogoutCounter(1);
        System.out.println("Got logout notification " + getLogoutCounter());
    }

    // Increase logout counter
    public void logoutBulkIndication(SubscriberID BULK subs)
    {
        System.out.println("Got logout notification");
        increaseLogoutCounter(subs.getSize());
    }

    public void loginPullRequest (String anonymousSubscriberID, NetworkID networkID)
    {
        try
        {
            increasePullCounter(1);
            System.out.println("Got pull request" + getPullCounter());

            // prepare policy
            PolicyProfile pp = new PolicyProfile(new String[]{"packageId=1","monitor=1"});

            // Answer with pull response
            // retrieve subscriber name – for example from your
            // policy server database
            // In this example we use fixed names based on the
            // subscribers counter
            api.loginPullResponse(anonymousSubscriberID,
                                  "sub"+getPullCounter(),
```

```
                                networkID,
                                pp,     // policy
                                null,   // no quota
                                h); // handler from previous example
        }
        catch (Exception ex)
        {
            System.out.println(ex.getMessage());
        }
    }

    public void loginPullRequestBulk(NetworkAndSubscriberID BULK subs)
    {
        try
        {
            increasePullCounter(subs.getSize());
            System.out.println("Got pull request" + getPullCounter());
            // Answer with pull response in bulk form
            PolicyProfile pp = new PolicyProfile(new String[]{"packageId=1","monitor=1"});

            LoginPullResponse_BULK responseBulk = new LoginPullResponse_BULK();

            Iterator subsIterator = subs.getIterator();

            // iterate of the received bulk (IPs and anonymous IDs)
            // and build a response bulk
            int count=0;
            while(subsIterator.hasNext())
            {
                // retrieve subscriber name – for example from your
                // policy server database
                // In this example we use fixed names based on the
                // subscribers counter
                String subName = "sub_"+count;

                SubscriberData sub = (SubscriberData)subsIterator.next();

                // Extract subscriber mappings from the bulk and
                // constract a new NetworkID based on those mappings
                NetworkID subNetId = new NetworkID(sub.getMappings(),
                                                   NetworkID.ALL_IP_MAPPINGS);

                responseBulk.addEntry(sub.getAnonymousSubscriberID(),
                                      subName,
                                      subNetId,
                                      true,
                                      pp,
                                      null);
                count++;
            }

            //use the bulk constructed above in the bulk response
            //use handler from the previous example
            api.loginPullBulkResponse(responseBulk,h);
        }
        catch (Exception ex)
        {
            System.out.println(ex.getMessage());
        }
    }

    public void getSubscribersBulkResponse(
                        NetworkAndSubscriberID BULK subs,
                        SubscruberBulkResponseIterator iterator)
```

```
    {
        //  not implemented in this example
    }

    synchronized void increaseLogoutCounter(long value)
    {
        logoutCount = logoutCount + value;
    }

    synchronized void increasePullCounter(long value)
    {
        pullCount = pullCount + value;
    }

    synchronized long getPullCounter()
    {
        return pullCount;
    }

    synchronized long getLogoutCounter()
    {
        return logoutCount;
    }

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

    public synchronized void waitForLogoutResult(int lastResult) {
        while (logoutCount<lastResult) {
           try {
                wait(100);
            } catch (catch (InterruptedException ie) {
                ie.printStackTrace();
            }
        }
    }
}
```

Class that contains the **main** method:

```
import java.util.Iterator;

import com.scms.api.sce.*;
import com.scms.common.*;

public class LogonPolicyServer {
    static PRPC_SCESubscriberApi api = null;

    // This sample program waits for pull requests from the SCE
    // and answers to them with pull response
    // The program exists after all 500 were logged in
    public static void main (String args[]) throws Exception
    {
        int numSubscribersToLogin = 500;

        //instantiate an API with reconnect interval of 5 seconds
```

```
            api = new PRPC_SCESubscriberApi("myAPI","1.1.1.1",5000);

            // construct an operation result handler (from the
            // previous example
            MyOperationResultHandler handler = new MyOperationResultHandler();

            // instantiate logout and login-pull listener
            MyListener listener = new MyListener(api);

            try
            {
                // register to logout indications
                api.registerLogoutListener(listener);
                api.registerLoginPullListener(listener);

                // connect to the SCE
                api.connect();

                // wait for login-pull requests from the SCE
                // they will be issued if you have traffic for unknown
                // subscribers at the SCE

                System.out.println("Waiting for pull requests for "+
                                                    numSubscribersToLogin+
                                                   " subscribers");

                // wait for all subscribers to be logged in
                listener.waitForPullResult(numSubscribersToLogin);

                //logout all subscribers
                System.out.println("logout of "+numSubscribersToLogin+" subscribers");

                for (int i=0; i<numSubscribersToLogin; i++)
                {
                    api.logout("sub"+i,null,handler);
                }

                // wait for all subscribers to be logged out
                listener.waitForLogoutResult(numSubscribersToLogin);
            }
            finally
            {
                api.unregisterLoginPullListener();
                api.unregisterLogoutListener();
                api.disconnect();
            }
        }
    }
```