Programming with the Cisco SCE Subscriber API

Published: December 23, 2013, OL-30585-01

Introduction

This chapter describes the application programming interface (API) programming structure, classes, methods, and interfaces.

API Classes

The following list presents the classes provided by the API:

Package com.scms.api.sce.prpc

Package com.scms.api.sce

Indication Listeners

Connection Monitoring

Cisco SCE Cascade Topology Support

Operation Results Handling

Package com.scms.common

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

- Login_BULK Class, page 4-10
- LoginPullResponse_BULK Class, page 4-15
- NetworkAndSubscriberID_BULK Class, page 4-14
- PolicyProfile_BULK Class, page 4-16
- SubscriberID_BULK Class, page 4-13
- SubscriberData, page 4-10 (class)
- SCAS_BB_Quota, page 4-7 (class)
- SCAS_BB_QuotaOperation, page 4-8 (class)
- Network ID Mappings, page 4-2 (NetworkID class)
- PolicyProfile Class, page 4-5

Programming Guidelines

This section provides the programming guidelines for the API methods.

Programming with Callback Methods

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

Caution Do not perform long operations within the thread of the callback method. Perform long operations from a separate thread. If you do not follow this recommendation, resource leakage can occur on the client side.

This caution notice applies to the following methods:

- LoginPullListener callback methods
- DualLoginPullListener 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 Cisco SCE (configuring the connection attributes)
- Registering/unregistering indication listeners
- Setting the connection listener
- Performing subscriber provisioning operations
- Disconnecting from the Cisco SCE

API Construction

The PRPC_SCESubscriber API 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 list presents parameters for the API constructors:

- apiName—Specifies an API name.

Note Provide a unique API name per Cisco SCE. If you construct more than one API with the same name and connect it to a single Cisco SCE, the Cisco SCE platform manages the APIs as one API client. Use this feature only when highavailability is supported. For more information about high availability, see the "Implementing High Availability" section on page 5-52.

- sceHost—Can be either an IP address or a reachable hostname.
- scePort—PRPC protocol TCP port by which to connect to the Cisco SCE (default value is 14374).
- autoReconnectInterval—Defines the interval (in milliseconds) after which reconnection is attempted, as follows:
  - If the value is 0 or lesser, the reconnection task is not activated (no auto-reconnect is attempted).
  - If the value is greater than 0 and a connection failure occurs, the reconnection task runs every <autoReconnectInterval>milliseconds.
  - Default value is 1 (no auto-reconnect is attempted).

Note To support the auto-reconnect action, call the **connect** method of the API 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();
```

Listener Setup Operations

After the API initializes, it activates listeners, which are based on the type of application that is using the API, and the topology. For more information about topologies, see the "Supported Topologies" section on page 2-6.

The following list presents the types of listener setup operations:

- Setting a connection listener. See the "Connection Monitoring" section on page 5-23:

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

- Setting a login-pull listener. See the "LoginPullListener Interface Class" section on page 5-15:

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

- Setting a dual-login-pull listener. See the "DualLoginPullListener Interface Class" section on page 5-17:

  ```
  public void registerLoginPullListener(DualLoginPullListener listener)
  ```

- Setting a logout listener. See the "LogoutListener Interface Class" section on page 5-19:

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

- Setting a quota listener. See the "QuotaListenerEx Interface Class" section on page 5-19:

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

- Setting a redundancy state listener. See the "RedundancyStateListener Interface" section on page 5-24:

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

Note The listener registration to the API causes resource allocations in the Cisco 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 Cisco SCE when the API reconnects.

Advanced Setup Operations

To customize the API, you can initialize certain internal properties. Perform the initialization by using the API init method.

Note For settings to take effect, the API must call the init method before the connect method.

You can set the following properties:

- Output queue size—The internal buffer size which sets the maximum number of requests that the API can accumulate before it sends the requests to the Cisco SCE. The default is 1024 kilobytes.
- Operation timeout—The interval of time (in milliseconds) that elapses before a timeout occurs on a non-responding PRPC protocol connection. The default is 45 seconds.

Syntax

The following example presents the syntax for the init method:

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

Parameters

- properties (java.util.Properties)—Enables setting the properties described in the "Advanced Setup Operations" section on page 5-6:
  - 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 can be invoked from listener callback. To avoid deadlocks, set this timeout to be shorter than com.scms.api.sce.prpc.regularInvocationTimeout.

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 Cisco SCE

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

To connect to the Cisco SCE, use the following method:

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

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

```
public boolean isConnected()
```

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

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 Cisco SCE that manage the reliability of the connection from the Cisco SCE to the API. If the application is restarting and you do not want to lose any messages, do not use the disconnect method.

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();
       }
}
```

Virtual Link API

Virtual link provisioning to the Cisco SCE platform is done through the Virtual Link Manager (VLM) in SM. The VLM finds the corresponding topology and provisions the virtual link details to Cisco SCE. However, an user can use the VLink API to provision the virtual link details to Cisco SCE without involving the VLM. The working of the VLink API is similar to that of the command-line interface (CLI) commands that are available in Cisco SCE for monitoring the virtual links.

Virtual Link Provisioning Operations

This section lists the Virtual Link API methods that can be used for provisioning virtual links. The description of each method includes information about its input parameters, return values, and exceptions. The following are the types of virtual link provisioning operations:

Create a Virtual Link

Syntax

The virtual link creation syntax is as follows:

```
public void createVLink(VLink vlinkData) throws VLinkDisabledException,VLinkModeException;
```

Description

This operation is used to create a virtual link in Cisco SCE.

Parameters

The create virtual link operation parameter is as follows:

vlinkData—Contains the data with which the VLink should be created. For details about the VLink class, see "VLink Class" section on page 4-19.

Return value

Null.

Exception

The following are the exceptions thrown by the create virtual link method:

- VLinkDisabledException. For more details on the VLinkdisabledException class, see the "VLinkDisabledException Class" section on page 4-20.
- VLinkModeException. For more details on the VLinkModeException class, see the "VLinkModeException Class" section on page 4-20.

Create Multiple Virtual Links

Syntax

The multiple virtual links creation syntax is as follows:

```
public void createVLink(VLink[] vlinkData) throws
VLinkDisabledException,VLinkModeException;
```

Description

This operation is used to create multiple virtual links in Cisco SCE.

Parameters

The create multiple virtual links operation parameter is as follows:

vlinkData—Describes the array of VLink objects in which each of the elements contain the data for the virtual links to be created in the Cisco SCE platform. For details about the corresponding VLink class, see the "VLink Class" section on page 4-19.

Return Value

Null.

Exception

The following are the exceptions thrown by the create multiple virtual links method:

- VLinkDisabledException. For more details on the VLinkdisabledException class, see the "VLinkDisabledException Class" section on page 4-20.
- VLinkModeException. For more details on the VLinkModeException class, see the "VLinkModeException Class" section on page 4-20.

Delete a Virtual Link

Syntax

The delete virtual link syntax is as follows:

```
public void deleteVLink(VLink vlinkData) throws VLinkDisabledException,VLinkModeException;
```

Description

This operation is used to delete a virtual link in Cisco SCE.

Parameters

The delete virtual link operation parameter is as follows:

vlinkData—Contains the data with which the VLink should be created. For details about the corresponding VLink class, see the "VLink Class" section on page 4-19.

Return Value

Null.

Exception

The following are the exceptions thrown by the delete a virtual link method:

- VLinkDisabledException. For more details on the VLinkdisabledException class, see the "VLinkDisabledException Class" section on page 4-20.
- VLinkModeException. For more details on the VLinkModeException class, see the "VLinkModeException Class" section on page 4-20.

Delete Multiple Virtual Links

Syntax

The delete multiple virtual links syntax is as follows:

```
public void deleteVLink(VLink[] vlinkData) throws
VLinkDisabledException,VLinkModeException;
```

Description

This operation is used to delete multiple virtual links in Cisco SCE.

Parameters

The delete multiple virtual link operation parameter is as follows:

vlinkData—Describes the array of VLink objects in which each of the elements contain the data for the virtual link to be created in the Cisco SCE platform. For details about the corresponding VLink class, see the "VLink Class" section on page 4-19.

Return Value

Null.

Exception

The following are the exceptions thrown by the delete multiple virtual links method:

- VLinkDisabledException. For more details on the VLinkdisabledException class, see the "VLinkDisabledException Class" section on page 4-20.
- VLinkModeException. For more details on the VLinkModeException class, see the "VLinkModeException Class" section on page 4-20.

Update a Virtual Link

Syntax

The update virtual link syntax is as follows:

```
public void updateVLink(VLink vlinkData) throws VLinkDisabledException,VLinkModeException;
```

Description

This operation is used to update a virtual link in Cisco SCE.

Parameters

The update virtual link operation parameter is as follows:

vlinkData—Contains the data with which the VLink should be updated. For details about the corresponding VLink class, see the "VLink Class" section on page 4-19.

Return Value

Null.

Exception

The following are the exceptions thrown by the update virtual link method:

- VLinkDisabledException. For more details on the VLinkdisabledException class, see the "VLinkDisabledException Class" section on page 4-20.
- VLinkModeException. For more details on the VLinkModeException class, see the "VLinkModeException Class" section on page 4-20.

Update Multiple Virtual Links

Syntax

The update multiple virtual links syntax is as follows:

```
public void updateVLink(VLink[] vlinkData) throws
VLinkDisabledException,VLinkModeException;
```

Description

This operation is used to update multiple virtual links in the Cisco SCE platform.

Parameters

The update multiple virtual links operation parameter is as follows:

vlinkData—Describes the array of VLink objects in which each of the elements contain the data for the virtual link to be updated in the Cisco SCE platform. For details about the corresponding VLink class, see the "VLink Class" section on page 4-19.

Return Value

Null.

Exception

The following are the exceptions thrown by the update multiple virtual links method:

- VLinkDisabledException. For more details on the VLinkdisabledException class, see the "VLinkDisabledException Class" section on page 4-20.
- VLinkModeException. For more details on the VLinkModeException class, see the "VLinkModeException Class" section on page 4-20.

Check Virtual Link Status

Syntax

The check virtual link status syntax is as follows:

```
public boolean isVlinkEnabled() throws VLinkModeException;
```

Description

This operation is used to check whether virtual links are enabled or disabled in the Cisco SCE platform.

Parameters

Null.

Return Value

The API returns a boolean value. The value can be either true or false.

Exception

The following is the exception thrown by the check virtual link status method:

VLinkModeException. For more details on the VLinkModeException class, see the "VLinkModeException Class" section on page 4-20.

Get Maximum Virtual Links

Syntax

The get maximum virtual links syntax is as follows:

```
public int getMaxVlinks() throws VLinkDisabledException,VLinkModeException;
```

Description

This operation is used to get the maximum number of virtual links that are supported in the Cisco SCE platform.

Parameters

Null.

Return Value

Maximum number of virtual links.

Exception

The following are the exceptions thrown by the get maximum virtual links method:

- VLinkDisabledException. For more details on the VLinkdisabledException class, see the "VLinkDisabledException Class" section on page 4-20.
- VLinkModeException. For more details on the VLinkModeException class, see the "VLinkModeException Class" section on page 4-20.

Enable the Virtual Link Mode

Syntax

The enable virtual link mode syntax is as follows:

```
public void enableVlinkMode();
```

Description

The virtual link API should be used in the user mode. This operation is used to enable the user mode before the virtual link API can be used.

Parameters

Null.

Return Value

Null.

Exception

Null.

Indication Listeners

The Cisco 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 listeners are registered to listen to the indications. For every type of indication, a separate listener can be registered. For descriptions about the events that trigger these indications, see Chapter 3, “Application Programming Interface Events.”.

LoginPullListener Interface Class

The LoginPullListener interface defines a set of callback functions that are used only in Pull Mode.

Policy servers that manage the Network ID part of the subscriber provisioning process, and that are intended to work in Pull Mode, register a LoginPullListener. This registration enables the policy server to respond to the login-pull requests from the Cisco SCE and to synchronize the Cisco SCE platform.

To enable listening to the LoginPullListener indications, the Cisco SCE Subscriber API sets a listener for LoginPullListener types of indications:

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

Note The Cisco SCE Subscriber API supports one LoginPullListener at a time. Do not have more than one API that has registered a LoginPullListener. If more than one Cisco SCE responds to the same login-pull request, the Cisco SCEs can lose synchronization.

The LoginPullListener is an interface that registers a login-pull indication 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)
}
```

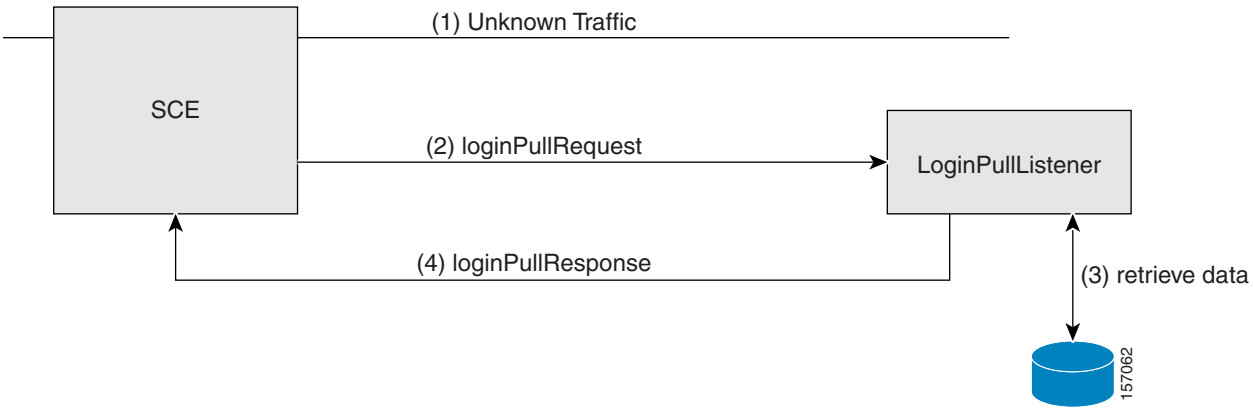
loginPullRequest Callback Method

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

This request is dispatched to the registered listener and triggers the loginPullRequest callback function. In response to this callback, the listener retrieves the subscriber information of the subscriber matching this IP address and activate loginPullResponse to deliver the information to the Cisco SCE (see the "loginPullResponse Operation" section on page 5-33). If no information exists for this IP address, no response is issued.

Figure 5-1 illustrates the loginPullRequest callback method.

Figure 5-1 loginPullRequest Callback Method



Parameters

The loginPullRequest Callback method parameters are as follows:

- anonymousSubscriberID—This anonymous subscriber ID *must* be supplied to the loginPullResponse operation (see the "loginPullResponse Operation" section on page 5-33). Also, see the "Anonymous Subscriber ID" section on page 2-2.
- networkID—The network identifier of the unknown subscriber. See the "Network ID" section on page 2-2.

loginPullRequestBulk Callback Method

This callback function is the bulk version of the loginPullRequest callback function.

Parameter

- subs—Contains pairs of Network IDs and anonymous IDs of several subscribers. See the "Parameters" section on page 5-17 of the loginPullRequest callback method.

The policy server can respond to a request by the loginPullBulkResponse method or activate the loginPullResponse method for each network ID in the bulk. See the "loginPullResponseBulk Operation" section on page 5-35 and the "loginPullResponse Operation" section on page 5-33. To retrieve the data contained in the subs parameter, use the next() iteration method provided by the bulk class. See the "Bulk Iterator" section on page 4-10.

GetSubscribersBulkResponse Callback Method

This callback method is used during the Cisco SCE synchronization process in the pull mode. For a detailed description, see the "Cisco SCE-API Synchronization" section on page 5-45.

DualLoginPullListener Interface Class

The DualLoginPullListener interface defines a set of callback functions that are used only in the pull mode.

Policy servers that manage the Network ID part of the subscriber provisioning process, and that are intended to work in the pull mode, register a DualLoginPullListener. This registration enables the policy server to respond to login-pull requests from the Cisco SCE and to synchronize the Cisco SCE platform.

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

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

Note The Cisco SCE Subscriber API supports one DualLoginPullListener at a time. Do not have more than one API that has registered a DualLoginPullListener. If more than one Cisco SCE responds to the same login-pull request, the Cisco SCEs can lose synchronization.

The DualLoginPullListener is an interface that registers a login-pull indication listener. It is defined as follows:

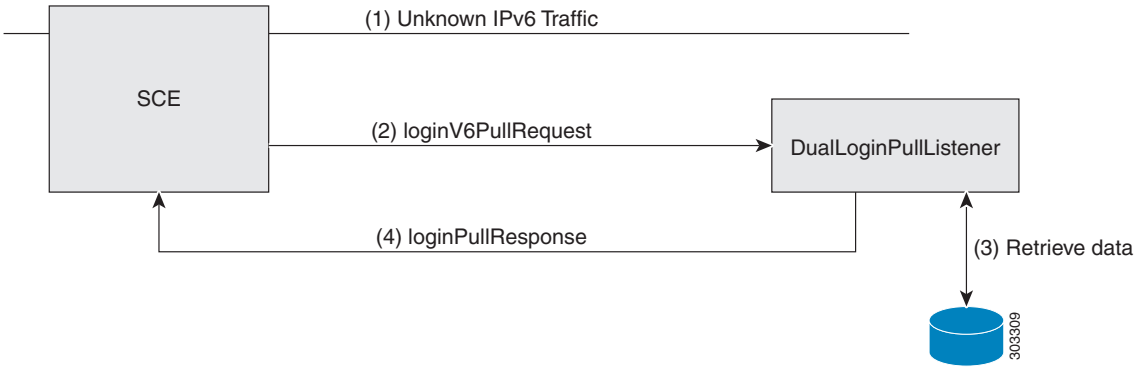
```
public interface DualLoginPullListener extends LoginPullListener
{
public void loginV6PullRequest(String anonymousSubscriberID, long higherOctetValue, long
lowerOctetValue);
}
```

Note To provision IPv4 only, IPv6 only, and dual-stack subscribers, use the DualLoginPullListener Interface class. You can use the LoginPullListener Interface class to provision only the IPv4 subscribers. The Cisco SCE Subscriber API supports the registration of either the DualLoginPullListener or the LoginPullListener interface.

loginV6PullRequest Callback Method

When Cisco SCE encounters a subscriber-side traffic with unknown IPv6 address, it issues a request for subscriber login information based on the IPv6 address. Cisco SCE expects the policy server to respond with the subscriber data for which this IP was allocated. This request is sent to the registered listener and triggers the loginV6PullRequest callback method.

Figure 5-2 loginV6PullRequest Callback Method



In response to the loginV6PullRequest callback, the listener retrieves the subscriber information of the subscriber matching this IP address and activates the loginPullResponse to deliver the information to Cisco SCE. If no information exists for this IP address, no response is issued.

Parameter

- anonymousSubscriberID—The temporary Subscriber ID assigned by Cisco SCE to each unknown subscriber Network ID when working in the pull mode. This ID is valid only until Cisco SCE receives the real subscriber ID from the policy server.
- higherOctetValue—Higher order 64-bit value
- lowerOctetValue—Lower order 64-bit value

LogoutListener Interface Class

Policy servers that are responsible for the network ID management part of the subscriber provisioning process can be configured to notify a LogoutListener when certain subscribers are removed from the Cisco SCE.

The API enables setting a LogoutListener 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:

- logoutIndication Callback Method, page 5-19
- logoutBulkIndication Callback Method, page 5-19

logoutIndication Callback Method

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

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

Parameter

- subscriberID—A unique identifier of the subscriber. See the "Subscriber ID" section on page 4-1. The Cisco SCE no longer handles this Subscriber ID.

logoutBulkIndication Callback Method

When the Cisco SCE platform identifies the logout of the last Network ID of each subscriber identified by the Subscriber ID in the group, it issues a logout indication for each subscriber. This indication triggers a call to the logoutIndication callback function of all the registered logout indication listeners.

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

Parameter

- subscriberID—A unique identifier of the subscriber. See the "Subscriber ID" section on page 4-1. Cisco SCE no longer handles this Subscriber ID.

QuotaListenerEx Interface Class

Note From Version 3.0.5, the QuotaListener interface is deprecated. Replace it with QuotaListenerEx. For backwards compatibility, the QuotaListener interface still exists, but 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 are able to receive quota-related indications issued by the Cisco SCE platform.

The API enables setting the QuotaListener 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 pass an object that implements QuotaListenerEx.

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

Note Do not use the bulk versions of the quota callback methods in this release of the API.

quotaStatusIndication Callback Method

Quota status indication returns a value that represents the number of quota buckets that remain of the set of quota buckets specified for a subscriber. The Cisco SCE issues this indication periodically or in response to a call to the getQuotaStatus operation (see the "getQuotaStatus Operation" section on page 5-43) and is distributed to the registered listener by activating a quotaStatusIndication callback function.

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

Parameters

The quotaStatusIndication Callback method parameters are as follows:

- subscriberID—The unique ID of the subscriber. See the "Subscriber ID" section on page 4-1.
- quota—Quota of the subscriber. See the "Subscriber Quota" section on page 4-6.

quotaStatusBulkIndication Callback Method

Quota status bulk indication returns a value that represents the number of quota buckets that remain of the set of quota buckets specified for a group of subscribers. The Cisco SCE issues this indication periodically or in response to the getQuotaStatusBulk operation (see the "Get Quota Status" section on page 3-6) and is distributed to the registered listener by activating a quotaStatusBulkIndication callback function.

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

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

Parameter

- subs—Contains quota data of the bulk of the subscribers. See the "Quota_BULK Class" section on page 4-17.

quotaBelowThresholdIndication Callback Method

When the quota of a subscriber drops below a preconfigured threshold, the Cisco 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

The quotaBelowThresholdIndication Callback method parameters are as follows:

- subscriberID—The unique ID of the subscriber. See the "Subscriber ID" section on page 2-2.
- quota—Quota of the subscriber. See the "Subscriber Quota" section on page 4-6.

quotaBelowThresholdBulkIndication Callback Method

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

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

Parameter

- subs—Contains quota data of the bulk of the subscribers. See the "Quota_BULK Class" section on page 4-17.

quotaDepletedIndication Callback Method

When the quota of a subscriber is depleted, the Cisco 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

The quotaDepletedIndication Callback method parameters are as follows:

- subscriberID—The unique ID of the subscriber. See the "Subscriber ID" section on page 2-2.
- quota—Quota of the subscriber. See the "Subscriber Quota" section on page 4-6.

quotaDepletedBulkIndication Callback Method

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

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

Parameter

- subs—Contains the names of the subscribers whose quota was depleted. See the "SubscriberID_BULK Class" section on page 4-13.

quotaStateRestore Callback Method

When a subscriber logs in to the policy server, the policy server performs a log in to the Cisco SCE. The Cisco SCE issues a request to the policy server to restore the subscriber quota in the Cisco SCE by activating a quotaStateRestore callback function. The policy server responds to this function with a quota update as described in the "Quota Update" section on page 3-6.

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

Parameters

The quotaStateRestore Callback method parameters are as follows:

- subscriberID—The unique ID of the subscriber. See the "Subscriber ID" section on page 2-2.
- quota—Quota of the subscriber. See the "Subscriber Quota" section on page 4-6. The bucket ID array size is 0 because when this indication is created, all the quota buckets are empty.

quotaStateBulkRestore Callback Method

When a group of subscribers logs in to the policy server, the policy server performs a login operation to the Cisco SCE. The Cisco SCE issues a request to the policy server to restore the subscriber quota in the Cisco SCE by activating a quotaStateBulkRestore callback function. The policy server responds to this function with a quota update as described in the "Quota Update" section on page 3-6.

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

Parameter

- subs—Contains the names of the subscribers whose quota was depleted. See the "SubscriberID_BULK Class" section on page 4-13.

Connection Monitoring

The Cisco SCE Subscriber API monitors the connection to the Cisco SCE platform. A policyserver that is requested to perform certain operations on connection establishment or disconnection from the Cisco SCE can implement a ConnectionListener interface.

ConnectionListener Interface

The API enables setting a connection listener.

```
setConnectionListener(ConnectionListener listener)
```

The ConnectionListener interface 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();
}
```

Use the Connection Establishment callback to start the Cisco SCE synchronization. See the "Cisco SCE-API Synchronization" section on page 5-45.

Disconnect Listener: Example

This example presents a basic 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
    }
}
```

Cisco SCE Cascade Topology Support

The Cisco SCE Subscriber API supports Cisco SCE cascade topologies. A policy server connected to a cascade Cisco SCE platform is required to know which of the Cisco SCEs in the cascade setup is active and which is standby. The policy server sends logon operations *only* to the active Cisco SCE. Similarly, the policy server performs subscriber synchronization *only* with the active Cisco SCE.

The standby Cisco SCE learns about the subscribers from the active Cisco SCE, which enables stateful failover. The policy server can identify a failover event and synchronize the Cisco SCE that becomes active so that it receives the most updated subscriber information.

To detect which Cisco SCE is active, the Policy Server can implement a RedundancyStateListener interface.

This section consists of the following topics:

isRedundancyStatusActive Method

The API provides the isRedundnacyStatusActive method with the RedundancyStateListener interface to monitor the Cisco SCE redundancy status.

```
public boolean isRedundancyStatusActive()
```

The return values from this method indicate the following:

- TRUE—If the Cisco SCE status is active.
- FALSE—In all other cases.

Use this method when first connecting to the cascade Cisco SCE to verify whether the Cisco SCE is active, before sending any logon operation to the Cisco SCE.

RedundancyStateListener Interface

To monitor cascade Cisco SCE state changes, the API enables setting a redundancy state listener.

```
setRedundancyStateListener(RedundancyStateListener listener)
```

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

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 performs a synchronization procedure on the Cisco SCE that becomes active. This synchronization is similar to the procedure that the Policy Server performs when a connection is established to the Cisco SCE.

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

Parameters

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

Configuring Cisco SCE to Ignore Cascade Violation Errors

By default, the Cisco SCE Release 3.1.0 is configured to return an error when a logon operation is performed on a standby Cisco SCE. Issue the ignore-cascade-violation CLI command on the Cisco SCE to change this behavior.

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

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

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

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

To configure the Cisco SCE to issue errors in response to cascade violation, use the following CLI command on the Cisco SCE platform:

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

To configure the parameter to the default value (to issue errors in case of cascade violation), issue the following CLI command on the Cisco SCE platform:

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

Note Configure Cisco SCE to ignore cascade violation only if you require backward compatibility with the existing Cisco SCE API. To use the cascade feature fully, monitor and use the Cisco SCE redundancy status.

Configuring Cisco SCE to Define the Quota Buffer Size and Quota Rate

To configure the Cisco SCE to define the size of the message queue that displays the Quota Manager (QM) indication when the QM is down, use the following CLI command in the Cisco SCE platform:

```
(config)#> management-agent sce-api quota-buffer-size 1000
```

The valid values are from1000 to 5000, and the default value is 1000.

To configure the Cisco SCE to define the quota indication rate, use the following CLI command in the Cisco SCE platform:

```
(config)#> management-agent sce-api quota-rate-control 125
```

The minimum and the default decimal value to be used is 125.

Note The Cisco SCE sends the quota Raw Data Record (RDR) and the Cisco SCE agent converts it into a quota indication message.

Result Handling

The API enables setting a result handler for every operation that enables managing results in a different manner.

The OperationResultHandler interface handleOperationResult callback is called when the result of an operation, which ran on the Cisco 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.

OperationResultHandler Interface

This OperationResultHandler interface receives 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);

}
```

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 receive operation results, set the result handler of each operation at the time of the operation call (as displayed in the examples).

The following data is returned from the OperationResultHandler interface:

- result—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. For non-bulk operations, the result array has only one entry. For bulk operations, each entry of the result array corresponds to the relevant entry in the bulk operation.
- handback—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 or output the parameters after the operation fails or to repeat the operation call.

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

OperationArguments Class

Use the following method to retrieve the operation name:

```
public String getOperationName()
```

Use the following method to retrieve the argument names:

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

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

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

Examples

The following example implements 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 You can use the previous sample implementation for both regular and bulk operations.

The following example implements the login operation 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 Cisco 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 Cisco SCE Subscriber API, which enables all future Cisco SCE API implementations to appear the same (Java, C, C++). Each OperationException exception provides the following information:

- Unique error code (long)
- Informative message (java.lang.String)
- Server-side stack trace (java.lang.String)

See Appendix A, "List of Error Codes" for details about error codes and their meanings.

Subscriber Provisioning Operations

This section lists the methods of the API that can be used for subscriber provisioning. The description of each method includes its input parameters and return values.

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

This sections consists of these subscriber provisioning operations:

Login Operation

Syntax

The login operation syntax is as follows:

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

Description

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

- If the subscriber ID does not exist in the Cisco SCE, a new subscriber is added with all the data supplied.
- If the subscriber ID exists:
  - If the networkIdAdditive parameter is set to TRUE, the supplied network ID is added to the existing network IDs of the subscriber. Otherwise, the supplied network ID replaces the existing network IDs.
  - policy—Policy is *updated* with the new policy values. Subscriber policy entries that are not provided in policy profile remain unchanged or are created with default values.
  - quota—The quota is *updated* according to the bucket values and the operations provided. See the "Subscriber Quota" section on page 4-6.
- If a networkID conflicts with another subscriber networkID, the networkID of the other subscriber is logged out implicitly and the new subscriber is logged in.

For a description of the relevant events, see the "Push Mode" section on page 3-3.

Parameters

The login operation parameters are as follows:

- subscriberID—Unique ID of the subscriber. For a description of the Subscriber ID format, see the "Subscriber ID" section on page 2-2.
- networkID—Network identifier of the subscriber. See the "Network ID Mappings" section on page 4-2.
- networkIdAdditive—If this parameter is set to TRUE, the supplied Network ID is added to the existing Network IDs of the subscriber. Otherwise, the supplied Network ID replaces the existing networkIDs.
- policy—Policy profile of the subscriber. See the "Cisco SCA BB Subscriber Policy Profile" section on page 4-5.
- quota—Quota of the subscriber. See the "Subscriber Quota" section on page 4-6.
- esa—Extended attributes associated with the subscriber and passed in the login operation.

  This new parameter was added in Cisco SCE Subscriber, Release 3.6.5 to support extended attributes in the login operation in Push mode. If there are no extended attributes for a specific subscriber, the same operation is used as in earlier releases:

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

- handler—Result handler for this operation. See the "Result Handling" section on page 5-27 for a description of the OperationResultHandler interface.

Error Codes

The following list presents the error codes that this method returns:

- 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 Appendix A, "List of Error Codes."

Examples

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

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

This example adds the IPv6 address 2000:2001:2002:abcd::/64 to an existing subscriber named alpha without affecting any of the existing mappings:

```
login(
    "alpha", // subscriber name
    new NetworkID(new String[]{"2000:2001:2002:abcd::/64"},
    NetworkID.ALL_IPV6_MAPPINGS),
    true, // isMappingAdditive is true
    null, // no policy
    null, // no quota
    null, //no esa
    handler); //operation result handler
```

For more examples, see the "Login and Logout" section on page 5-53.

loginBulk Operation

Syntax

The loginBulk operation syntax is as follows:

```
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

The loginBulk Operation parameters are as follows:

- subsBulk—See the "Login_BULK Class" section on page 4-10.
- handler—Result handler for this operation. See the "Result Handling" section on page 5-27 for a description of the OperationResultHandler interface.

Error Codes

The following list presents the error codes that this method returns:

- 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 Appendix A, "List of Error Codes".

loginPullResponse Operation

Syntax

The loginPullResponse operation syntax is as follows:

```
void loginPullResponse(String subscriberID,
                       String anonymousSubscriberID,
                       NetworkID networkID,
                       PolicyProfile policy,
                       QuotaOperation quota,
                       ExtendedAttributes esa,
```

```
                                    OperationResultHandler handler) throws Exception
```

Description

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

For relevant events description, see the "Pull Mode" section on page 3-3.

Parameters

The loginPullResponse Operation parameters are as follows:

- subscriberID—Unique ID of the subscriber. For a description of the Subscriber ID format, see the "Subscriber ID" section on page 2-2.
- anonymousSubscriberID—Identifier of the anonymous subscriber. The Cisco SCE sends this parameter within the loginPullRequest/loginPullBulkRequest indication (see the "LoginPullListener Interface Class" section on page 5-15). See the "Anonymous Subscriber ID" section on page 2-2.
- networkID—Network identifier of the subscriber. See "Network ID Mappings" section on page 4-2. This parameter must include the network ID received by the loginPullRequest. If this subscriber in the Cisco SCE already has other Network IDs, this Network ID is added.
- policy—Policy profile of the subscriber. See the "Cisco SCA BB Subscriber Policy Profile" section on page 4-5.
- quota—Quota of the subscriber. See the "Subscriber Quota" section on page 4-6.
- esa—Extended attributes associated with the subscriber and passed in the login operation.

  This new parameter was added in Cisco SCE Subscriber, Release 3.6.5 to support extended attributes in the login operation during Pull mode. If there are no extended attributes for a specific subscriber, the same operation is used as in previous releases:

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

- handler—Result handler for this operation. See the "Result Handling" section on page 5-27 for a description of the OperationResultHandler interface.

Error Codes

The following list presents the error codes that this method returns:

- 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 Appendix A, "List of Error Codes."

loginPullResponseBulk Operation

Syntax

The loginPullResponseBulk operation syntax is as follows:

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

Description

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

For a description of the relevant events, see the "Pull Mode" section on page 3-3.

Parameters

The loginPullResponseBulk operation parameters are as follows:

- subsBulk—See the "LoginPullResponse_BULK Class" section on page 4-15.
- handler—Result handler for this operation. See the "Result Handling" section on page 5-27 for a description of the OperationResultHandler interface.

Error Codes

The following list presents the error codes that this method returns:

- 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 Appendix A, "List of Error Codes".

Logout Operation

Syntax

The Logout operation is as follows:

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

Description

This operation removes the specified networkID of the subscriber from the Cisco SCE. If this networkID is the last networkID of the specified subscriber, the subscriber is removed from the Cisco SCE. If no subscriber ID is specified, the supplied network ID is removed from the Cisco SCE without regard to the network ID to which this subscriber belongs. If no network ID is supplied, all Network IDs of this subscriber are removed.

If the subscriber record is not in the Cisco SCE, the logout operation succeeds.

For a description of the relevant events, see the "Logout Events" section on page 3-4.

Parameters

The logout operation parameters are as follows:

- subscriberID—Unique ID of the subscriber. For a description of the subscriber ID format, see the "Subscriber ID" section on page 2-2.
- networkID—Network identifier of the subscriber. See the "Network ID Mappings" section on page 4-2.
- handler—Result handler for this operation. See the "Result Handling" section on page 5-27 for a description of the OperationResultHandler interface.

Error Codes

The following list presents the error codes that this method returns:

- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED

For a description of error codes, see Appendix A, "List of Error Codes".

logoutBulk Operation

Syntax

The logoutBulk operation syntax is as follows:

```
void logoutBulk(NetworkAndSubscriberID_BULK subsBulk,
```

```
                                    OperationResultHandler handler) throws Exception
```

Description

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

For a description of the relevant events, see the "Logout Events" section on page 3-4.

Parameters

The logoutBulk operation parameters are as follows:

- subsBulk—See the "NetworkAndSubscriberID_BULK Class" section on page 4-14.
- handler—Result handler for this operation. See the "Result Handling" section on page 5-27 for a description of the OperationResultHandler interface.

Error Codes

The following list presents the error codes that this method can return:

- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED

For a description of error codes, see Appendix A, "List of Error Codes".

networkIdUpdate Operation

Syntax

The networkIdUpdate operation syntax is as follows:

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

Description

This operation adds or replaces an existing subscriber network ID.

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

For a description of the relevant events, see the "Network ID Update Event" section on page 3-5.

Parameters

The networkIdUpdate operation parameters are as follows:

- subscriberID—Unique ID of the subscriber. For a description of the subscriber ID format, see the "Subscriber ID" section on page 2-2.
- networkID—Network identifier of the subscriber. See the "Network ID Mappings" section on page 4-2.
- networkIDAdditive—If this parameter is set to TRUE, the supplied network ID is added to the existing network IDs of the subscriber. Otherwise, the supplied network ID replaces the existing network IDs.

Error Codes

The following list presents the error codes that this method returns:

- 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 Appendix A, "List of Error Codes".

networkIdUpdateBulk Operation

Syntax

The networkIdUpdateBulk operation syntax is as follows:

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

Description

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

For a description of the relevant events, see the "Network ID Update Event" section on page 3-5.

Parameters

The networkIdUpdateBulk operation parameters are as follows:

- subsBulk—See the "NetworkAndSubscriberID_BULK Class" section on page 4-14.
- handler—Result handler for this operation. See the "Result Handling" section on page 5-27 for a description of the OperationResultHandler interface.

Error Codes

The following list presents the error codes that this method returns:

- 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 Appendix A, "List of Error Codes".

profileUpdate Operation

Syntax

The profileUpdate operation syntax is as follows:

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

Description

This operation modifies an existing subscriber policy profile. If the subscriber record does not exist in the Cisco SCE, this operation fails.

For a description of the relevant events, see the "Profile Update" section on page 3-5.

Parameters

The profileUpdate operation parameters are as follows:

- subscriberID—Unique ID of the subscriber. For a description of the subscriber ID format, see the "Subscriber ID" section on page 2-2.
- policy—Policy profile of the subscriber. See the "Cisco SCA BB Subscriber Policy Profile" section on page 4-5.
- handler—Result handler for this operation. See the "Result Handling" section on page 5-27 for a description of the OperationResultHandler interface.

Error Codes

The following list presents the error codes that this method returns:

- 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 Appendix A, "List of Error Codes".

profileUpdateBulk Operation

Syntax

The profileUpdateBulk operation syntax is as follows:

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

Description

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

For a description of the relevant events, see the "Profile Update" section on page 3-5.

Parameters

The profileUpdateBulk operation parameters are as follows:

- subsBulk—See the "PolicyProfile_BULK Class" section on page 4-16.
- handler—Result handler for this operation. See the "Result Handling" section on page 5-27 for a description of the OperationResultHandler interface.

Error Codes

The following list presents the error codes that this method returns:

- 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 Appendix A, "List of Error Codes".

quotaUpdate Operation

Syntax

The quotaUpdate operation syntax is as follows:

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

Description

This operation updates the subscriber quota.

For a description of the relevant events, see the "Quota Update" section on page 3-6.

Parameters

The quotaUpdate operation parameters are as follows:

- subscriberID—Unique ID of the subscriber. For a description of the subscriber ID format, see the "Subscriber ID" section on page 2-2.
- quotaOperation—Quota operation to perform on the quota of the subscriber. See the "Subscriber Quota" section on page 4-6 for more information.
- handler—Result handler for this operation. See the "Result Handling" section on page 5-27 for a description of the OperationResultHandler interface.

Error Codes

The following list presents the error codes that this method returns:

- 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 Appendix A, "List of Error Codes".

quotaUpdateBulk Operation

Syntax

The quotaUpdateBulk operation syntax is as follows:

```
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 a description of the relevant events, see the "Quota Update" section on page 3-6.

Parameters

The quotaUpdateBulk operation parameters are as follows:

- subsBulk—See the "QuotaOperation_BULK Class" section on page 4-18.
- handler—Result handler for this operation. See the "Result Handling" section on page 5-27 for a description of the OperationResultHandler interface.

Error Codes

The following list presents the error codes that this method returns:

- 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 Appendix A, "List of Error Codes".

getQuotaStatus Operation

Syntax

The getQuotaStatus operation syntax is as follows:

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

Description

This operation requests the amount of quota remaining of the specified set of quota buckets. The getQuotaStatus indication including the queried data follows this request (*asynchronously*). See the "quotaStatusIndication Callback Method" section on page 5-20.

For a description of the relevant events, see the "Get Quota Status" section on page 3-6.

Parameters

The getQuotaStatus operation parameters are as follows:

- subscriberID—Unique ID of the subscriber. For a description of the subscriber ID format, see the "Subscriber ID" section on page 2-2.

- quota—Includes the list of names (without values) of the quota buckets to retrieve. See the "Subscriber Quota" section on page 4-6 for more information about how to construct with only the bucket names .
- handler—Result handler for this operation. See the "Result Handling" section on page 5-27 for a description of the OperationResultHandler interface.

Error Codes

The following list presents the error codes that this method can 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 Appendix A, "List of Error Codes".

getQuotaStatusBulk Operation

Syntax

The getQuotaStatusBulk operation syntax is as follows:

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

Description

This operation is a bulk version of the getQuotaStatus operation described in the "getQuotaStatus Operation" section on page 5-43.

For a description of the relevant events, see the "Get Quota Status" section on page 3-6.

Parameters

The getQuotaStatusBulk operation parameters are as follows:

- subsBulk—See the "Quota_BULK Class" section on page 4-17.
- handler—Result handler for this operation. See the "Result Handling" section on page 5-27 for a description of the OperationResultHandler interface.

Error Codes

The following list presents the error codes that this method returns:

- 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 Appendix A, “List of Error Codes”.

Cisco SCE-API Synchronization

When the Cisco SCE and the policy server have a conflict in the data about a subscriber because of disconnection, loss of logon messages, or reboot, the following problems can occur:

- Misclassification of the traffic of one subscriber as if it was the traffic of another subscriber
- Enforcement of the wrong service on the subscriber traffic
- Loss of resources

It is possible to prevent such conflicts by maintaining the communication channels as reliable as synchronize the subscriber data between the Cisco SCE and the policy server using the API. The policy server always initiates the synchronization.

Caution Performing the synchronization process from several policy servers at the same time causes the subscriber information in the Cisco SCE to be inconsistent with all servers.

The following list presents the synchronization guidelines to which the policy server must adhere while implementing synchronization:

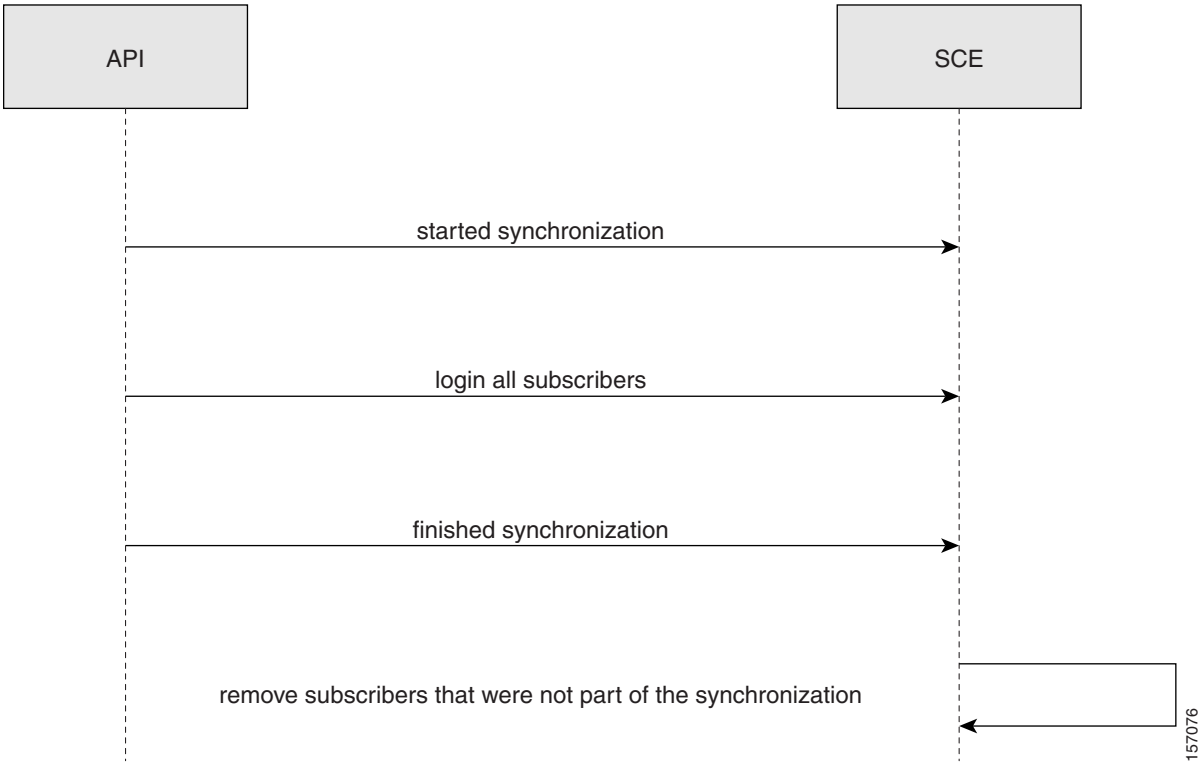
- Push Mode Synchronization Procedure, page 5-46
- Pull Mode Synchronization Procedure, page 5-47

Push Mode Synchronization Procedure

Figure 5-3 illustrates the following procedure:

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

Figure 5-3 Push Mode Synchronization



Note The regular logon operations can be performed during the synchronization process.

The following sections describe the methods provided for use in the synchronization procedure in *Push* mode:

- synchronizePushStart, page 5-47
- synchronizePushEnd, page 5-47

synchronizePushStart

Syntax

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

Description

Use this operation in Push mode only to signal the Cisco SCE that synchronization with the server is about to begin. The Cisco SCE marks all the subscriber data with a dirty-bit, which is reset if this data is applied again as part of the synchronization process. Every call to this method restarts the synchronization process.

Parameter

handler—Result handler for this operation. See the "Result Handling" section on page 5-27 for a description of the OperationResultHandler interface.

synchronizePushEnd

Syntax

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

Description

Use this operation in Push mode only to signal the Cisco SCE that synchronization with the server has ended. The Cisco SCE scans the entire subscriber database for data with the dirty-bit assigned at the synchronizePushStart indication and removes it.

Parameters

The synchronizePushEnd parameters are as follows:

- success—Indicates to the Cisco SCE that the synchronization was successful.
- handler—Result handler for this operation. See the "Result Handling" section on page 5-27 for a description of the OperationResultHandler interface.

Pull Mode Synchronization Procedure

Figure 5-4 illustrates the following procedure:

1. The Policy Server indicates to the Cisco SCE that it is starting to synchronize the Cisco SCE.
2. The Policy Server retrieves from the Cisco SCE all the subscriber IDs and network IDs that it is currently managing.
3. The Policy Server fixes any erroneous synchronization.

Algorithm

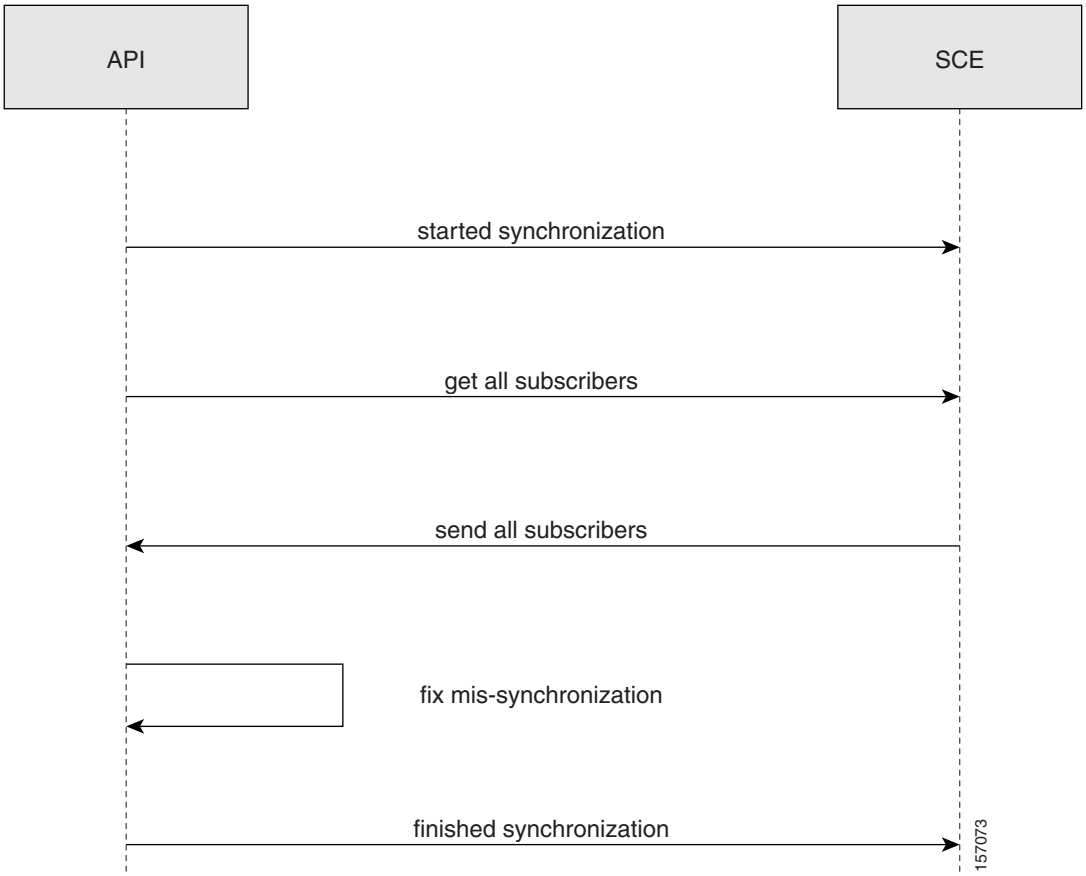
Use the following algorithm template when planning the synchronization procedure:

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 Cisco SCE.
- Otherwise, send a logout with the Subscriber IP to the Cisco SCE.

Step 2. and Step 3. are performed as a bulk simultaneously.

Figure 5-4 Pull Mode Synchronization



Note The regular logon operations can be performed during the synchronization process.

The following sections describe the methods provided for use in the synchronization procedure in *Pull* mode:

- synchronizePullStart, page 5-49
- synchronizePullEnd, page 5-49
- getSubscribersBulk, page 5-49

synchronizePullStart

Syntax

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

Description

Use this operation in *Pull* mode only to signal the Cisco SCE that synchronization with the server is about to start.

Parameter

handler—Result handler for this operation. See the "Result Handling" section on page 5-27 for a description of the OperationResultHandler interface.

synchronizePullEnd

Syntax

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

Description

Use this operation in *Pull* mode only to signal the Cisco SCE that synchronization with the server has ended.

Parameters

The synchronizePullEnd parameters are as follows:

- handler—Result handler for this operation. See the "Result Handling" section on page 5-27 for a description of the OperationResultHandler interface.
- success—Indicates to the Cisco SCE that the synchronization was successful.

getSubscribersBulk

Syntax

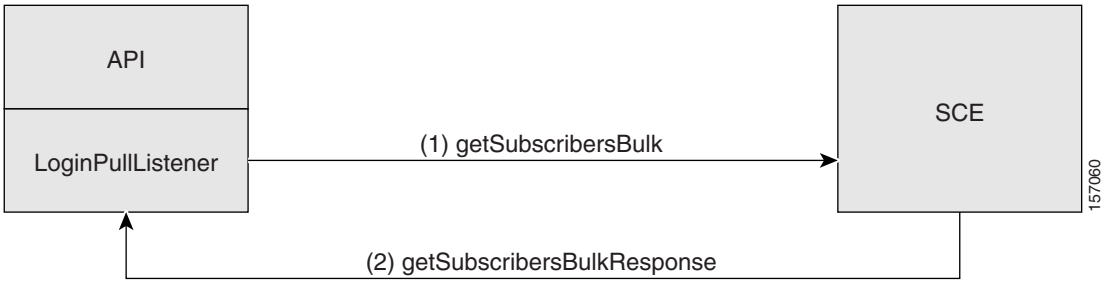
```
void getSubscribersBulk(int bulkSize,
                        SubscribersBulkResponseIterator iterator,
                        OperationResultHandler handler)
```

Description

Use this operation in the *Pull* mode synchronization process to retrieve a bulk of subscribers that the Cisco SCE is currently managing (see the "Pull Mode Synchronization Procedure" section on page 5-47).

Upon receiving this request (getSubscribersBulk), the Cisco SCE asynchronously issues the getSubscribersBulkResponse indication containing subscriber IDs and corresponding network IDs (see the "LoginPullListener Interface Class" section on page 5-15) as shown in Figure 5-5. This method supplies an iterator that is passed to the next call of the getSubscribersBulk indication. To signal the end of iterations, the iterator of the last bulk is null.

Figure 5-5 getSubscribersBulk Description



Parameters

The getSubscribersBulk parameters are as follows:

- bulkSize—Size of the bulk to retrieve. Maximum bulk size is limited to 100 entries.
- iterator—Iterator of the subscribers at the Cisco SCE side. This iterator is received in the getSubscribersBulkResponse indication and it is passed to the next call to the 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 the "Result Handling" section on page 5-27 for a description of the OperationResultHandler interface.

getSubscribersBulkIPv6

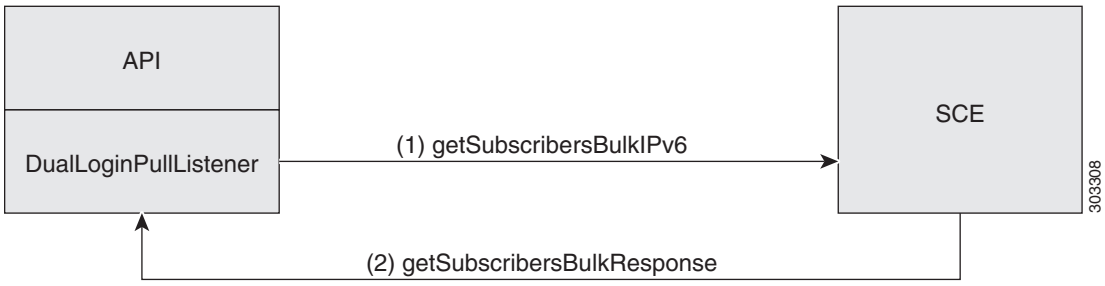
Syntax

```
public void getSubscribersBulkIPv6(int bulkSize,
    com.scms.common.SubscribersBulkResponseIterator iterator,
    OperationResultHandler handler)
    throws java.lang.Exception
```

Description

Use this operation in the *pull* mode synchronization process to retrieve the bulk of IPv6 subscribers that Cisco SCE is currently managing (see the "Pull Mode Synchronization Procedure" section on page 5-47).

Figure 5-6 getSubscribersBulkIPv6 Description



On receiving a getSubscribersBulkIPv6 request, Cisco SCE asynchronously issues the getSubscribersBulkResponse indication containing the subscriber IDs and the corresponding Network IDs (see the "DualLoginPullListener Interface Class" section on page 5-17). One of the indication parameters is the isLastBulk flag that is checked before moving to the next call to the getSubscribersBulkIPv6. The isLastBulk flag is specified by getSubscribersBulkIPv6 in the interface SCESubscriberApi.

Parameters

- bulkSize—Number of entries in the bulk. The maximum number of entries is 100.
- iterator—Iterator of the subscribers on the Cisco SCE side. This iterator is received in getSubscribersBulkIndication and is passed to the next call to getSubscribersBulkIPv6 method. When calling the getSubscribersBulkIPv6 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 getSubscribersBulkIPv6 operation. See the "Result Handling" section on page 5-27 for a description of the OperationResultHandler interface.

Note To retrieve both the IPv4 and IPv6 mappings of dual-stack subscribers, both the getSubscribersBulk method and the getSubscribersBulkIPv6 method must be called.

Advanced API Programming

This section provides details on advanced API programming using Cisco SCE Subscriber API including implementing high svailability.

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 a time. The other server (standby) is not connected to the Cisco SCE.

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

Note You can implement high-availability separately for every policy server that is provisioning the Cisco SCE platform simultaneously.

To implement high-availability with the Cisco SCE Subscriber API, perform the following:

- Set up a two-node cluster for two policy servers.
- Construct two API instances with the same API name. Each one must be on a different server (node) within the cluster. (For constructor description, see the "API Construction" section on page 5-5). During cluster run time, only one API instance is connected to the Cisco SCE platform. When a fail-over occurs, the failed server disconnects from the Cisco SCE and the standby server becomes active and reconnects to the Cisco SCE within the predefined timeout (see the "Resetting the Disconnection Timeout to the Default Value" section on page 1-8). Because of identical API names, the Cisco SCE behaves as if the same API was reconnected and no information is lost.

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

API Code Examples

This section presents several API code examples:

Login and Logout

The following example logs in a predefined number of subscribers to the Cisco 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();
            }
        }
    }
}
```

The following class contains a basic 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());
    }
}
```

The following class 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)};
    }
}
```

The following example logs in a predefined number of subscribers to Cisco SCE, and then logs them out. Because this example uses autoreconnect support, it does not define a connection listener. The following example also 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();
            }
        }
    }
}
```

The following class contains a basic 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());
    }
}
```

The following class 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(getIPv6Mappings(i), // generate ip
                NetworkID.ALL_IPV6_MAPPINGS),
                true, // additive flag
                pp, // policy
                null, // no quota
                null, no esa
                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(getIPv6Mappings(i),
NetworkID.ALL_IPV6_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[] getIPv6Mappings(int i)
    {
        return new String[]{ "abcd:" +((int)i/65536)%256 + ":" +
            ((int)(i/256))%256 + ":" + (i%256) + "::/64" };
    }
}
```

login-pull Request and login-pull Response

The following code fragment demonstrates a login-pull request and login-pull response manipulation. 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
                // construct 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();
            }
        }
    }
}
```

The following class 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();
        }
    }
}
```

The following code fragment demonstrates a login-pull request and login-pull response manipulation for IPv6. This class is a sample implementation of the listener for the logout and login pull indications.

```
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();
        }
    }
}
```

The following is an example of the result handler from the previous example:

```
import java.util.Iterator;
// result handler from the previous example
import MyOperationResultHandler;
import com.scms.api.sce.*;
import com.scms.common.*;
class MyListener implements DualLoginPullListener, 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 loginV6PullRequest(String anonymousSubscriberID, long arg1, long arg2)
        {
            try
            {
                increasePullCounter(1);
                System.out.println("MyListener - Got IPv6 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
    String ipv6Address = IPV6Utilities.getIPv6As64BitsString(IPV6Utilities.getBigIntIPV6(arg1,
    arg2));
    NetworkID networkID = new NetworkID(new String[] {ipv6Address+"/64"}, new short[]
    {NetworkID.TYPE_IPV6});
                api.loginPullResponse("sub"+getPullCounter(),
                anonymousSubscriberID,
                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
            // construct 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();
            }
        }
    }
}
```

The following class demonstrates the pull synchronization for IPv6:

```
import java.util.Iterator;
// result handler from the previous example
import com.scms.api.sce.*;
import com.scms.common.*;


public class myDualLoginPullListener implements DualLoginPullListener, LogoutListener {
    // indications counters
    public long logoutCount = 0;
    public long pullCount=0;
    public long myNumOfBulkInPullSynchProcess=0;
    public boolean synchEnded = false;
    // 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 myResultHandler = new MyOperationResultHandler();
        public myDualLoginPullListener(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
            myResultHandler); // handler from previous example
            }
            catch (Exception ex)
            {
            System.out.println(ex.getMessage());
            }
        }
        public void loginV6PullRequest(String anonymousSubscriberID, long arg1, long arg2)
{
            try
            {
            increasePullCounter(1);
            System.out.println("MyListener - Got IPv6 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

            String ipv6Address =
IPV6Utilities.getIPv6As64BitsString(IPV6Utilities.getBigIntIPV6(arg1, arg2));
           NetworkID networkID = new NetworkID(new String[] {ipv6Address+"/64"}, new
short[] {NetworkID.TYPE_IPV6});
            api.loginPullResponse("sub"+getPullCounter(),
            anonymousSubscriberID,
            networkID,
            pp, // policy
            null, // no quota
            myResultHandler); // handler from previous example
            }
            catch (Exception ex)
            {
            System.out.println(ex.getMessage());
            }

        }

    public void loginPullRequestBulk(NetworkAndSubscriberID_BULK subs)
    {
        // not implemented in this example
     }
            public void getSubscribersBulkResponse(NetworkAndSubscriberID_BULK
subs,SubscribersBulkResponseIterator iterator)
            {
                Iterator it = subs.getIterator();
                PolicyProfile pp = new PolicyProfile(new
String[]{"packageId=1","monitor=1"});
                while(it.hasNext())
                 {

                     SubscriberData sub = (SubscriberData)it.next();
```

```
                    String subName = sub.getSubscriberID();
                    if(!subName.equals("N/A"))
                    {
                        try
                        {
                                api.policyProfileUpdate(sub.getSubscriberID(),pp,
myResultHandler);
                                System.out.println("Policy Updated for
"+sub.getSubscriberID());

                        }
                        catch(Exception e)
                        {
                            e.printStackTrace();
                        }
                    }
                    else
                    {
                        System.out.println("Party N/A in the Bulk !!!!!");
                    }

                }
                 if (iterator != null)
                    {
                        try
                        {   myNumOfBulkInPullSynchProcess++;
                            api.getSubscribersBulkIPv6(subs.getSize(), iterator,
myResultHandler);
                        }
                        catch (Exception e)
                        {

                            e.printStackTrace();
                        }

                    }
                    else// null indicates all parties were sent to the api
                    {

                        synchEnded = true;
                    }


            }
            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 (InterruptedException ie) {
                ie.printStackTrace();
                }
            }
        }

        public void waitTillSynchEnded(int count)
        {
            while (myNumOfBulkInPullSynchProcess < count)
            {
                try {
                    Thread.sleep(100);
                } catch (InterruptedException ie) {
                ie.printStackTrace();
                }
            }
        }



    }
```

The following class contains the main method of the pull synchronization:

```
import java.util.Iterator;
import com.scms.api.sce.*;
import com.scms.common.*;
public class pullSynch {
    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 = 5;
    int bulkSize=5;
    SubscribersBulkResponseIterator myIt = null;
    //instantiate an API with reconnect interval of 5 seconds
    api = new PRPC_SCESubscriberApi("myAPI","10.78.241.51",5000);
    // construct an operation result handler (from the
    // previous example
    MyOperationResultHandler handler = new MyOperationResultHandler();
    // instantiate logout and login-pull listener
    myDualLoginPullListener listener = new myDualLoginPullListener(api);
    try
    {
    // register to logout indications
    api.registerLogoutListener(listener);
    api.registerLoginPullListener(listener);
    // connect to the SCE
```

```
    api.connect();
    //Login Subscribers
    for (int i=0; i<numSubscribersToLogin; i++)
    {
        api.login("test"+i,
                new NetworkID("aaaa:"+i+"::/64", // generate ip
                NetworkID.TYPE_IPV6),
                true, // additive flag
                null, // policy
                null, // no quota
                handler);
    }

    api.synchronizePullStart(handler);

     Thread.sleep(100);

    api.getSubscribersBulkIPv6(bulkSize, myIt, handler );

        Thread.sleep(100);
    listener.waitTillSynchEnded(numSubscribersToLogin/bulkSize);

    api.synchronizePullEnd(true, handler);
     Thread.sleep(100);

    }
    finally
    {
    api.unregisterLoginPullListener();
    }
    }
}
```

iVirtual Link Operations

The following sample program creates, updates and deletes VLinks in Cisco SCE:

```
i

import java.net.UnknownHostException;
import com.scms.api.sce.prpc.PRPC_SCESubscriberApi;
import com.scms.api.sce.prpc.VLink;
import com.scms.api.sce.prpc.Channel;
import com.scms.api.sce.prpc.VLinkDisabledException;

public class VLinkAPI extends PRPC_SCESubscriberApi {

    static PRPC_SCESubscriberApi Api = null;

    public static void main(String args[]) throws Exception {

        try {

            //building vlink with AL
            VLink v1 = new VLink(100, "vlink1", VLink.UPSTREAM, 0, 2432, 20, -1);
            VLink v2 = new VLink(102, "vlink2", VLink.DOWNSTREAM, 0, 2432, 16,
                    -1);
            VLink v3 = new VLink(105, "vlink3", VLink.UPSTREAM, 0, 2432, 16, -1);

            //building vlink without AL
```

```
        VLink v4 = new VLink(105, "vlink3", VLink.UPSTREAM, 0, 2432, 18);

        //building channel with AL
        Channel ch1 = new Channel(101, "channel1", 2432, 18, 5);
        Channel ch2 = new Channel(103, "channel2", 2432, 18, 5);

        //building channel without AL
        Channel ch3 = new Channel(104, "channel3", 2432, 18);

        Channel[] c = new Channel[] { ch2, ch3 };

        //addchannel with addChannel(Channel channel)
        v1.addChannel(ch1);

        //addchannel with addChannels(Channel[] channels)
        v2.addChannel(c);

        VLink[] v = new VLink[] { v1, v2 };

        //instantiate an Api
        Api = new PRPC_SCESubscriberApi("testSCEAPI", "1.1.1.1" , -1);

        //Enable user mode to use vlink API
        Api.enableVlinkMode();

        //connect to SCE
        Api.connect();

        //vlink status in SCE
        System.out.println(Api.isVlinkEnabled());

        //printing max number of vlinks
        System.out.println(Api.getMaxVlinks());

        //creating vlink with createVLink(VLink vlinkData)
        Api.createVLink(v3);

        //creating vlink with createVLink(VLink[] vlinkData)
        Api.createVLink(v);

        //updating vlink with updateVLink(VLink vlinkData)
        Api.updateVLink(v4);

        v1.setVlinkCIR(20);
        v2.setVlinkPIR(30);
        VLink[] vv = new VLink[] { v1, v2 };

        //updating vlink with updateVLink(VLink[] vlinkData)
        Api.updateVLink(vv);

        //deleting vlink with deleteVLink(VLink vlinkData)
        Api.deleteVLink(v4);

        //deleting vlink with deleteVLink(VLink[] vlinkData)
        Api.deleteVLink(vv);

    }

    catch (VLinkDisabledException vde) {
        System.out.println(vde);
    }

    catch (Exception e) {
        e.printStackTrace();
```

```
        }

        finally {
            Api.disconnect();
        }

}
    public VLinkAPI() throws UnknownHostException {
        super("VLinkAPI", "1.1.1.1" , -1);
    }

}
```