# Cisco SCMS SCE Subscriber API Programmer Guide

Release 3.1
May 2007

Americas Headquarters
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
http://www.cisco.com
Tel: 408 526-4000
800 553-NETS (6387)
Fax: 408 527-0883

Customer Order Number:
Text Part Number: OL-8236-04

*Cisco SCMS SCE Subscriber API Programmer Guide*

# CONTENTS

CHAPTER 4

CHAPTER 5

# About this Guide

**Revised: May 30, 2007, OL-8236-04**

The *SCMS SCE Subscriber API Programmer Guide* is used for integrations that require direct access to the SCE platform for subscriber provisioning purposes.

This introduction provides information about the following topics:



## Audience

This guide is intended for the networking or computer technician responsible for integrations involving policy servers that perform subscriber provisioning with the SCE platform.

## Document Revision History

| Cisco Service Control Release | Part Number | Publication Date |
|---|---|---|
| Release 3.1.0 | OL-8236-04 | May, 2007 |

**Description of Changes**

- Added new RedundancyStateListener interface to support cascade SCE setups. See Information About SCE Cascade Topology Support

| Cisco Service Control Release | Part Number | Publication Date |
|---|---|---|
| Release 3.0.5 | OL-8236-03 | November, 2006 |

**Description of Changes**

- Added new section on Quota State Restore Event.

- Updated SCAS_BB_Quota class.
- Updated the QuotaListenerEx interface due to deprecation of the QuotaListener interface. See Information About the QuotaListenerEx Interface Class /

| Cisco Service Control Release | Part Number | Publication Date |
|---|---|---|
| Release 3.0.3 | OL-8236-02 | May, 2006 |

**Description of Changes**

- Updated API code examples. See API Code Examples.

| Cisco Service Control Release | Part Number | Publication Date |
|---|---|---|
| Release 3.0 | OL-8236-01 | December, 2005 |

**Description of Changes**

- First version of this document.

# Organization

The major sections of this guide are as follows:

***Table 1***

| Chapter | Title | Description |
|---|---|---|
| Chapter 1 | Getting Started | Discusses the platforms on which the SCE Subscriber API can be used, and how to install, compile, and start running the API. |
| Chapter 2 | Concepts and Terms | Describes various terms and concepts that are utilized when working with the SCE Subscriber API. |
| Chapter 3 | API Events | Describes various events accessed by the SCE Subscriber API. |
| Chapter 4 | Getting Familiar with the API Data Types | Describes the various API data types. |
| Chapter 5 | Programming with the SCE Subscriber API | Provides a detailed description of the API programming structure, classes, methods, and interfaces. |

*Table 1*

| Chapter | Title | Description |
|---|---|---|
| Chapter 6 | Troubleshooting | Describes the usage of the API logging abilities for troubleshooting the integration with the API. API logging enables the user to monitor the operations being called including the received parameters both at the API client and at the SCE side. |
| Appendix A | List of Error Codes | Lists the error codes that are returned by the API. |

# Related Publications

Use this API Guide in conjunction with the following Cisco documentation:

- *Cisco SCMS Subscriber Manager User Guide*
- *Cisco Service Control Application for Broadband (SCA BB) User Guide*
- *Cisco SCE 1000 2xGBE Installation and Configuration Guide*
- *Cisco SCE 2000 4xGBE Installation and Configuration Guide*

# Document Conventions

This guide uses the following conventions:

- **Bold** is used for commands, keywords, and buttons.
- *Italics* are used for command input for which you supply values.
- `Screen` font is used for examples of information that are displayed on the screen.
- **`Bold screen`** font is used for examples of information that you enter.
- Vertical bars ( | ) indicate separate alternative, mutually exclusive elements.
- Square brackets ( [ ] ) indicate optional elements.
- Braces ( {} ) indicate a required choice.
- Braces within square brackets ( [{}] ) indicate a required choice within an optional element.

**Note** Means *reader take note*. Notes contain helpful suggestions or references to material not covered in the guide.

**Timesaver** Means the *described action saves time*. You can save time by performing the action described in the paragraph.

**Caution** Means *reader be careful*. In this situation, you might do something that could result in equipment damage or loss of data.

**Warning** **Means *danger*. You are in a situation that could cause bodily injury. Before you work on any equipment, you must be aware of the hazards involved with electrical circuitry and familiar with standard practices for preventing accidents. To see translated versions of warnings, refer to the *Regulatory Compliance and Safety Information* document that accompanied the device.**

# Obtaining Documentation, Obtaining Support, and Security Guidelines

For information on obtaining documentation, obtaining support, providing documentation feedback, security guidelines, and also recommended aliases and general Cisco documents, see the monthly *What's New in Cisco Product Documentation*, which also lists all new and revised Cisco technical documentation, at:

http://www.cisco.com/en/US/docs/general/whatsnew/whatsnew.html

CHAPTER 1

# Getting Started

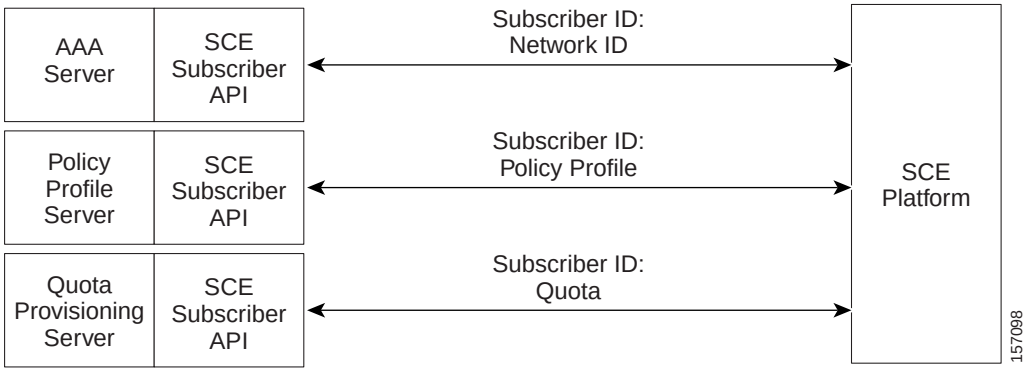
This module discusses the platforms on which the SCE Subscriber API can be used, and how to install, compile, and start running it.

The SCMS SCE Subscriber API provides the ability to external applications (policy servers) to connect directly to the SCE for the purpose of subscriber provisioning.

*Subscriber provisioning* is a process of updating the *Network IDs* , *Policy Profile* and *Quota* characteristics of the subscriber using the Subscriber ID as the correlation. For more information about the characteristics of the subscriber in the Service Control Application for Broadband (SCA BB), see the Subscriber Characteristics section.

The API can be installed and used concurrently on several policy servers and each of them can perform different parts of the subscriber provisioning process as shown in the following diagram:

***Figure 1-1 SCE Subscriber API Installed on Multiple Servers***




The API uses the PRPC (Proprietary Remote Procedure Call) protocol as a transport for the connection to the SCE. The PRPC is a proprietary RPC protocol designed by Cisco.

**Note** The API provides a connection to one SCE platform for each API instance.

# Restrictions for the SCMS SCE Subscriber API

Version 3.0.5 of the API is backward compatible with previous versions, but is not binary-compatible. You must recompile applications that use a previous version of the API in order to use the new version. Since the API is backward compatible, you do not need to make any changes to the application source code.

**Note** If you upgrade the SCE to version 3.0.5, you must upgrade the API to version 3.0.5 and recompile the application that uses it.

# Information About the SCMS SCE Subsriber API



## Platforms

The SCMS SCE Subscriber API is operable on any platform that supports Java version 1.4.

## Package Content

For brevity, the installation directory **sce-java-api-_<version>_<build-number>**is referred to as *<installdir>*.

The *<installdir>*/**javadoc** folder contains the SCE Subscriber API JAVADOC documentation.

The *<installdir>*/**lib** folder contains the **sceapi.jar** file, which is the API executable. It also contains additional jar files necessary for the API operation.

***Table 1-1 Layout of Installation Directory***

| Path | Name | Description |
|---|---|---|
| <installdir> | | |
| | README | API readme file |
| <installdir>/javadoc | | |
| | index.html | Index of all API specifications |
| | (API specification files, etc.) | API specification documents |
| <installdir>/lib | | |
| | sceapi.jar | SCE Subscriber API executable |
| | asn1rt.jar | Utility jar used by the API |
| | log4j.jar | Utility jar used by the API |
| | log4j.properties | Property file needed for the logging functionalities |
| | jdmkrt.jar | Utility jar used by the API |

# How to Extract and Install the Package

The SCMS SCE Subscriber API distribution is part of the SCMS SM-LEG distribution file and is located in the *sce_api* directory.

The SCMS SCE Subscriber API is packaged in a UNIX tar file. You can extract the SCMS SCE Subscriber API using the UNIX tar utility of most Windows compression utilities.

- Installing the Distribution on a UNIX Platform
- Installing the Distribution on a Windows Platform

## Installing the Distribution on a UNIX Platform

**Step 1** Extract the SCMS SM-LEG distribution file and locate the SCE Subscriber API distribution tar **sce-java-api-dist.tar.gz**

**Step 2** Unzip the distribution file

```
#>gunzip sce-java-api-dist.tar.gz
```

**Step 3** Extract the SCE Subscriber API package tar

**#>tar -xvf sce-java-api-dist.tar**

## Installing the Distribution on a Windows Platform

**Step 1** Use a zip extractor (such as WinZip) to unzip the package.

# How to Setup the SCE Platform

The following sections describe the configuration that is performed on the SCE platform to allow correct API functioning.

- Prerequisites
- Configuring the SCE in a Pull Model
- How to Configure the RDR Formatter
- How to Configure the RDR Server
- How to Configure the API Disconnection Timeout

## Prerequisites

The API connects to the PRPC server on the SCE platform. The PRPC server is a server running a proprietary RPC protocol designed by Cisco. For more information, see the *SCE User Guide* .

Before using the API, ensure that:

- The SCE must be up and running, and reachable from the machine that hosts the API.
- The PRPC server on the SCE must be started.

# Configuring the SCE in a Pull Model

To enable the SCE platform to issue a request for subscriber information when running in a Pull Model (see Pull Model ), configure the following using the SCE platform Command-Line Interface (CLI).

For more information about configuring the SCE platform, see the *Cisco SCE 1000 2xGBE Installation and Configuration Guide* or the *Cisco SCE 2000 4xGBE Installation and Configuration Guide* .

**Step 1** Configure the subscriber templates

```
(config if)#>subscriber template import   CSV file
```

For more information about the templates and the format of the CSV file, see the *Cisco Service Control Application for Broadband User Guide* .

**Step 2** Configure the unmapped-subscriber groups ranges

**a.** Use the **subscriber anonymous group import** CLI to import anonymous groups from a file.

```
(config if)#>subscriber anonymous group import   CSV file
```

**b.** Alternatively, use the **subscriber anonymous group name** CLI to manually define the anonymous group

```
(config if)#>subscriber anonymous group name   NAMEIP-range   IP RANGE
```

# How to Configure the RDR Formatter

- Configuring the RDR Formatter to Issue Quota-Related Indications
- Mapping the Quota RDR Tags to a Different Category

## Configuring the RDR Formatter to Issue Quota-Related Indications

To enable the RDR formatter to issue quota-related indications, configure the RDR formatter on the SCE platform as follows.

**Step 1** Use the **RDR-formatter destination** CLI.

```
#>RDR-formatter destination 127.0.0.1 port 33001 category number 4 priority 100
```

## Mapping the Quota RDR Tags to a Different Category

By default, Quota RDR tags are mapped to category 4. If another category is required, use the following command.

**Step 1** Use the **RDR-formatter rdr-mapping** CLI

```
#>RDR-formatter rdr-mapping tag-ID   tag numbercategory-number   number
```

**Note** For Quota RDR tag IDs, see the *Cisco Service Control Application for Broadband User Guide* .

To enable the application to issue quota-related indications, it should be enabled in the *Cisco Service Control Application for Broadband GUI* . See the *Cisco Service Control Application for Broadband User Guide* for configuration description.

# How to Configure the RDR Server

To enable the API to receive Quota indications, the RDR server should be enabled and listening on the same port that is configured in the RDR formatter.

- Verifying the RDR Server Configuration
- Enabling the RDR Server
- Changing the RDR Server Port

## Verifying the RDR Server Configuration

To verify the RDR server configuration, do the following.

**Step 1** Use the **show RDR-server** CLI.

```
#>show RDR-serverRDR server is ONLINE
RDR server port is 33001
```

## Enabling the RDR Server

To enable the RDR server, do the following

**Step 1** Use the configuration **RDR-server** CLI

```
#>configure(config)#>RDR-server Default RDR server port is 33001
```

## Changing the RDR Server Port

To change the RDR server port, do the following.

**Step 1** Use the **RDR-server port** CLI

```
#>configure(config)#>RDR-server port  port
```

# How to Configure the API Disconnection Timeout

The SCE platform allows setting the timeout for the API to reconnect to the SCE platform after it was disconnected. During this timeout, the SCE will not free the resources and no data will be lost. After the timeout has elapsed and the API did not reconnect, the SCE considers the API disconnected and frees all the resources. The default timeout value is 5 minutes.



## Configuring the API Disconnection Timeout

To configure the API disconnection timeout, do the following:

**Step 1** Use the **management-agent sce-api timeout** CLI

```
(config)# management-agent sce-api timeout    timeout-in-sec
```

## Reset the Disconnection Timeout to the Default Value

To reset the API disconnection timeout to the default value, do the following:

**Step 1** Use the **default management-agent sce-api timeout** CLI

```
(config)# default management-agent sce-api timeout
```

## Viewing the Timeout Value

To view the timeout value, do the following:

**Step 1** Use the **show management-agent sce-api** CLI

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

CHAPTER 2

# Concepts and Terms

This module describes various terms and concepts that are utilized when working with the SCMS SCE Subscriber API.

- Subscriber Characteristics
- Information About Subscriber Integration Models
- Non-blocking Model
- Indications Listeners
- Supported Topologies
- Multi-threading Support
- Auto-reconnect Support
- Reliability Support
- High Availability Support
- Synchronization
- Practical Tips

## Subscriber Characteristics

One of the fundamental entities in the Service Control Application for Broadband (SCA BB) solution is a subscriber . A subscriber is the entity that the SCA BB solution individually monitors, accounts, and enforces a service configuration. The following sections briefly describe the characteristics of the subscriber in the SCA BB. For more information about the format and usage of the subscriber's characteristics, see the Getting Familiar with the API Data Types module.

- Subscriber ID
- Anonymous Subscriber ID
- Network ID
- Policy Profile
- Quota

## Subscriber ID

Subscriber ID is a subscriber unique identifier, for example, a user name, IMSI (International Mobile Subscriber Identity), or other codes that uniquely identify a subscriber.

## Anonymous Subscriber ID

When working in the Pull Model integration, the SCE assigns each unknown subscriber IP address with a temporary Subscriber ID, Anonymous Subscriber ID, until it receives the real Subscriber ID from the Policy Server.

For more information on the Pull Model integration, see the Information About Subscriber Integration Models section.

## Network ID

The SCE correlates a certain traffic flow to a subscriber by mapping a network identifier, for example, IP address, IP range, or VLAN, to the subscriber entity.

## Policy Profile

A Policy Profile includes a set of parameters used by the SCA BB solution to define what policy is enforced on the subscriber.

## Quota

A quota includes the quota-bucket values of the service quota or quotas available for the usage of the subscriber.

# Information About Subscriber Integration Models

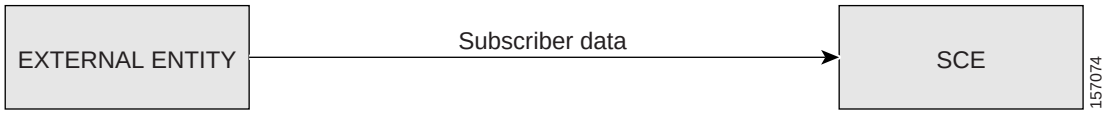
The following terms describe two models of a dynamic subscriber integration that the SCE platform supports.

- Push Model
- Pull Model

## Push Model

In push model integration, an external server introduces (pushes) the subscribers to the SCE platform. This is performed whenever a new subscriber logs in to the network or the external server presumes to know all subscribers and introduces them to the SCE box when they connect.

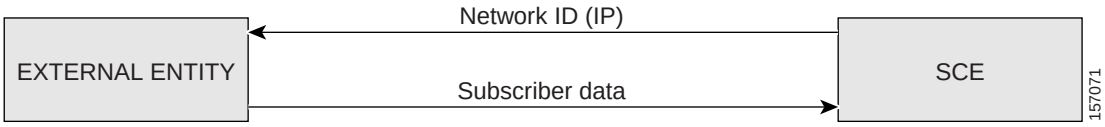
*Figure 2-1 Push Model Schematic*




# Pull Model

In pull model integration, the SCE platform requests subscriber data from the external entity when it encounters traffic of an unknown subscriber, known as an anonymous subscriber. The external entity retrieves the required subscriber information and sends it back to the SCE platform.

*Figure 2-2 Pull Model Schematic*




# Non-blocking Model

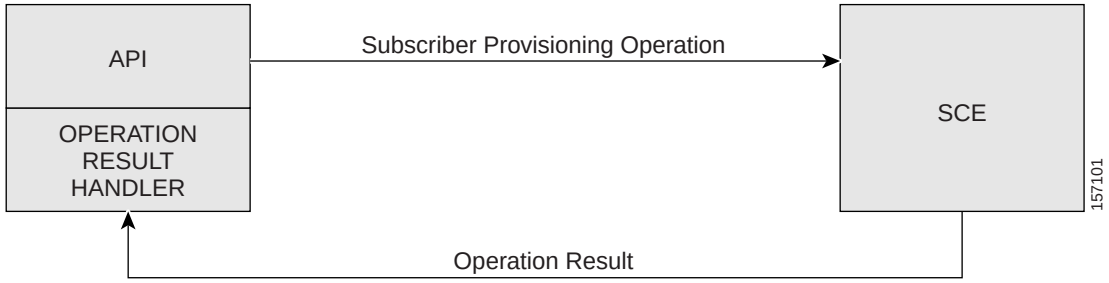
The SCE Subscriber API is implemented using a non-blocking model. Non-blocking methods return immediately, even before the completion of a subscriber provisioning operation. The Non-blocking Model method is advantageous when the operation is lengthy and involves I/O. Performing the operation in a separate thread allows the caller to continue doing other tasks and it improves overall system performance.

The operation results are either returned to an Observer object (Listener) or may not be returned at all.

The API supports retrieval of operation results using an operation result handler described in the Information About Result Handling section.

The following diagram illustrates the Non-blocking Model method during a subscriber provisioning operation:

*Figure 2-3 Non-blocking Model*

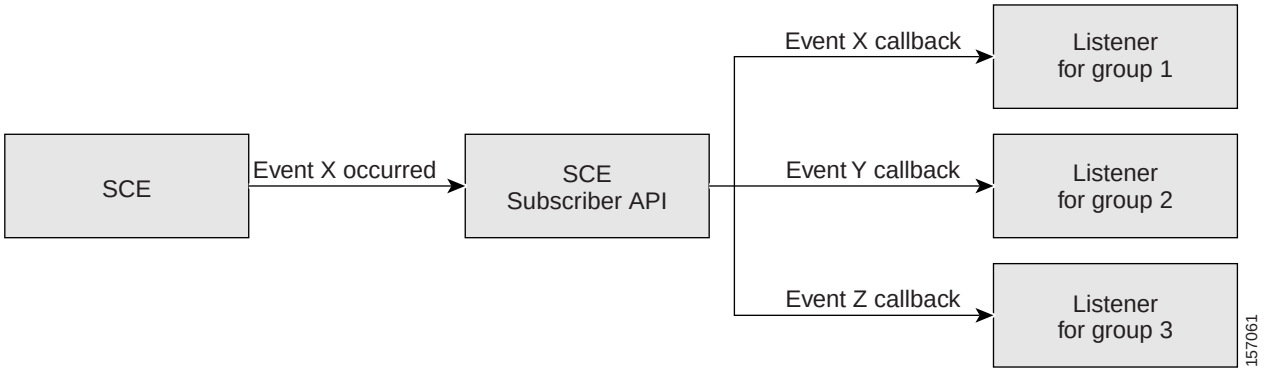


Operation results can be used for operation result error logging or for inspection of the parameters used by the operation.

# Indications Listeners

The API provides the user with the ability to receive an indication when certain events occur on the SCE platform. The API dispatches the indications received from the SCE tothe interested entities, called listeners , by activating the relevant Listener's callback methods. The indications are separated into several logical groups when only **one** listener can be defined for each group of indications.

***Figure 2-4 Indication Listeners***




To receive certain indications, you need to register a listener to the API that implements the required callback functions. After the listener is registered, the API can dispatch the required indications to the listener. The SCMS SCE Subscriber API provides three types of indications when separate listeners are registered to the following types of the indications:

- Login-pull indications
- Logout indications
- Quota indications

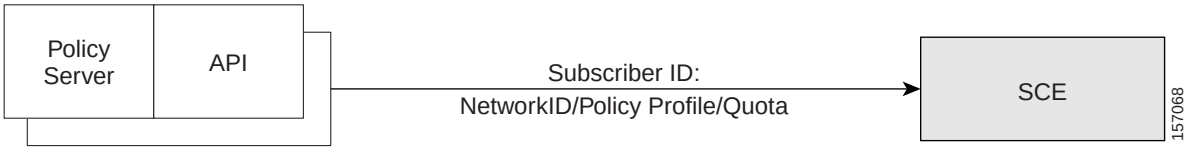
For more information about listener indications, see the API Events module.

# Supported Topologies

The following topologies are recommended to use with the SCMS SCE Subscriber API:

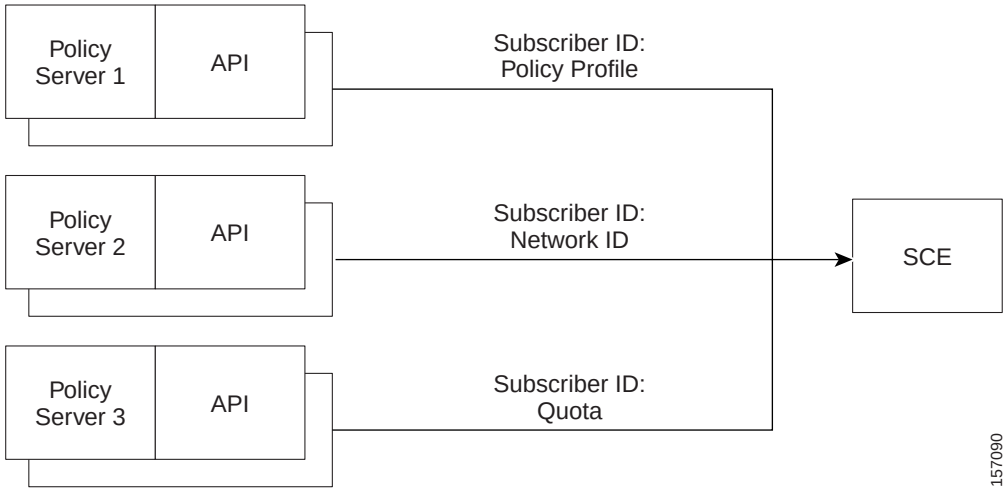
- One policy server (or two-node cluster) that is responsible for all aspects of the subscriber provisioning process:

***Figure 2-5 Supported Topologies - One Policy Server***



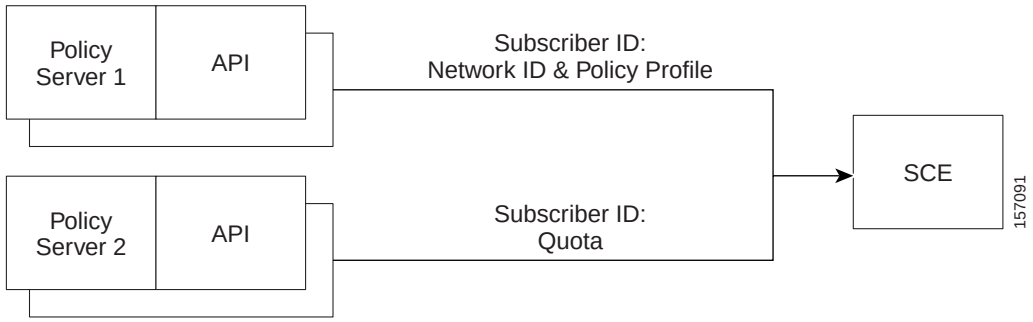
- Three policy servers (or three two-node clusters)—Every server is responsible for a different aspect of the subscriber provisioning process:

***Figure 2-6 Supported Topologies - Three Policy Servers***



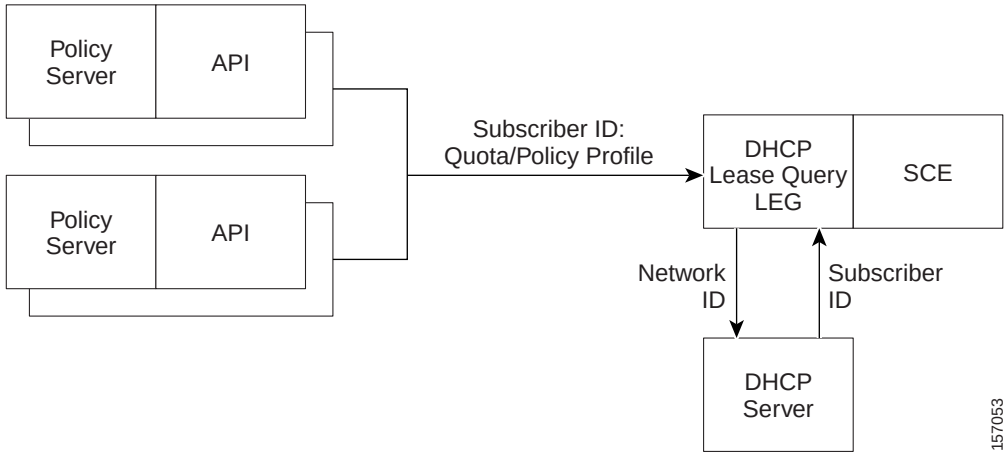

- Two policy servers (or two two-node clusters) when one of the servers is responsible for two aspects of the subscriber provisioning and the other server is responsible for one aspect only (any combination is allowed). For example:

***Figure 2-7 Supported Topologies - Two Policy Servers***



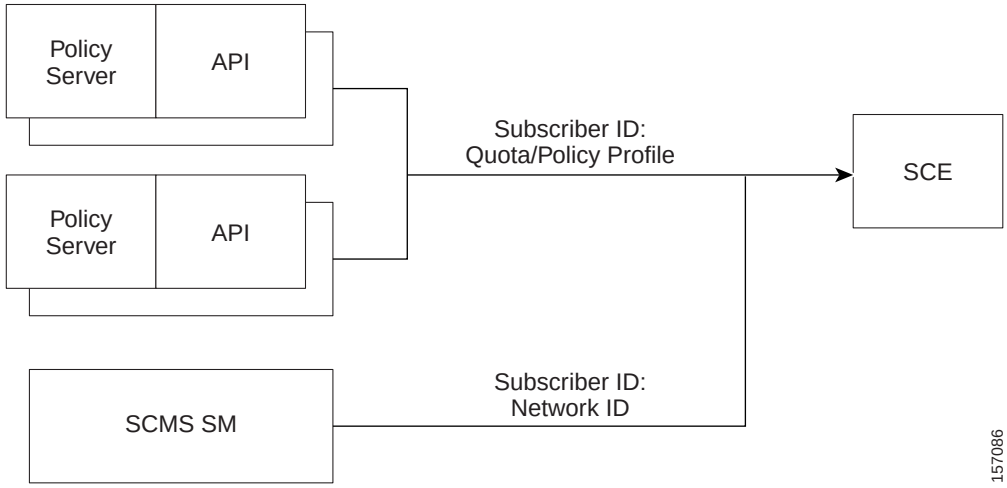

- DHCP Lease Query LEG, which is responsible for mapping a Network ID to a Subscriber ID, with one or more policy servers as described in the three policy server diagram above. The following diagram shows the DHCP Lease Query LEG:

*Figure 2-8 Supported Topologies - DHCP Lease Query LEG*




- SCMS SM, which is responsible for mapping Network ID to Subscriber ID, with one or more policy servers. The number of policy servers depends on whether the SM is used for policy profile provisioning in addition to the network ID:

*Figure 2-9 Supported Topologies - SM*




**Note** The API itself does not limit the use of any topology; however, the SCE platform does not correlate between all the entries (Policy Servers) that perform subscriber provisioning. Therefore you should be **extremely** careful when using more than one Policy Server for the **same provisioning purpose** (for example Network ID/Subscriber ID correlation). If you are not careful when using more than one Policy Server, the SCE platform may receive different information for the same subscriber from the two policy servers responsible for the same aspect of the subscriber provisioning. This may cause a loss of synchronization with at least one policy server. For example, using two policy servers that are responsible for providing Subscriber ID/Network ID correlation for the same subscriber will produce the situation where the SCE is always synchronized with the policy server that performed the last update for this subscriber.

# Multi-threading Support

The API supports an unlimited number (limited by the available memory) of threads calling its methods simultaneously.

**Note** In a multi-threaded scenario, the order of invocation is **guaranteed** : the API performs operations in the same chronological order that they were called.

# Auto-reconnect Support

The API supports auto-reconnection to the SCE in case of connection failure. When this option is activated, the API can determine when the connection to the SCE is lost. When the connection is lost, the API activates a reconnection task that tries to reconnect to the SCE again in a configurable interval time until reconnection is successful.

# Reliability Support

The SCMS SCE Subscriber API is implemented as a *reliable* API. The API ensures that no requests to the SCE are lost and no indication from the SCE is lost. The API maintains an internal storage for all API requests that were sent to the SCE. Only after receiving an acknowledgement from the SCE that the request was handled, it considers the request as **committed** and the API can remove the request from its internal storage. If a connection failure occurs between the API and the SCE, the API accumulates all requests in its internal storage until the connection to the SCE is reestablished. On reconnection, the API resends all **non-committed** requests to the SCE, ensuring that no requests are lost.

**Note** The order of resending requests is **guaranteed** : the API resends the requests in the same chronological order that they were called.

# High Availability Support

The API provides high availability support. It assumes that the high availability scheme of the policy server is a two-node cluster type where only one server is active at any given time. The other server, in standby, is not connected to the SCE. For more information, see the Implementing High Availability.

# Synchronization

The SCE and Policy Server must be kept synchronized concerning the subscribers for which the SCE is handling their internal parameters. Otherwise, the SCE might confuse one of the subscriber's traffic to another subscriber, or the subscriber's SLA (Service Level Agreement) will not be enforced because of a change in the policy that did not reach the SCE. For more information, see Information About SCE-API Synchronization.

# Practical Tips

When implementing the code that integrates the API with your application you should consider the following practical tips:

- Connect once to the SCE and maintain an open API connection to the SCE at all times, using the API many times. Establishing a connection is a timely procedure, which allocates resources on the SCE side and the API client side.
- Share the API connection between your threads - it is better to have one connection per Policy Server. Multiple connections require more resources on the SCE and client side.
- Do not implement synchronization of the calls to the API. The client automatically synchronizes calls to the API.
- If the Policy Server application has bursts of logon operations, enlarge the internal buffer size accordingly to hold these bursts (Non-Blocking flavor).
- During the integration, use the logging capabilities that are described in the SCE Logging and the API Client Logging sections to view the API operations in the SCE's client logs and to troubleshoot problems during the integration, if any.
- Use the debug mode for the Policy Server application that logs/prints the return values of the operations.
- Use the automatic reconnect feature to improve the resiliency of the connection to the SCE.

CHAPTER 3

# API Events

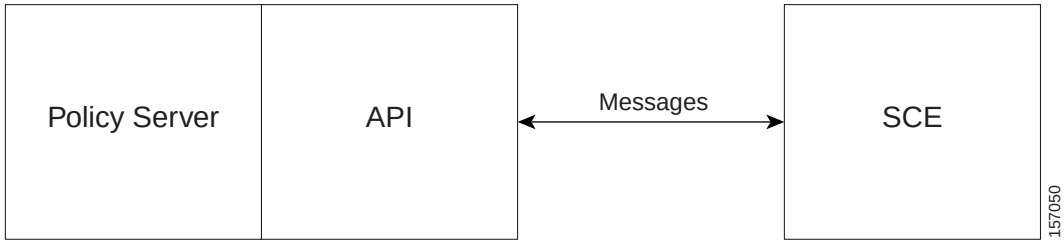
This module describes various events accessed by the SCMS SCE Subscriber API.

- Information About API Events

## Information About API Events

The API accesses a set of 'events' that are a pre-defined set of messages passed back and forth between the Policy Server and the SCE platform:

***Figure 3-1 API Events Overview***




Every message can be assigned a type according to the purpose of the message:

- *Request* —Requests information or an action to be performed. A request is not necessarily followed by a response.
- *Response* —Answers a previous request
- *Indication* —Indicates the other side that an event has occurred

Most of the events may be used for both **push** and **pull** models. See Information About Subscriber Integration Models.

The events may be divided into the following Subscriber Provisioning process groups:

- *Network ID management events* —Includes events relating to the modification of the subscriber Network ID mapping
- *Policy Profile management events* —Includes events relating to modification of the subscriber Policy Profile parameters
- *Quota management events* —Includes events relating to the management of subscriber quota

- *SCE Synchronization management events* —Includes events relating to the management of the SCE synchronization process

You can perform bulk operations, which bundle many triggers for the same event on many subscribers to one global event.

The following sections provide a general description of each type of event.



# Information About Network ID Management Events



## Information About Login Events

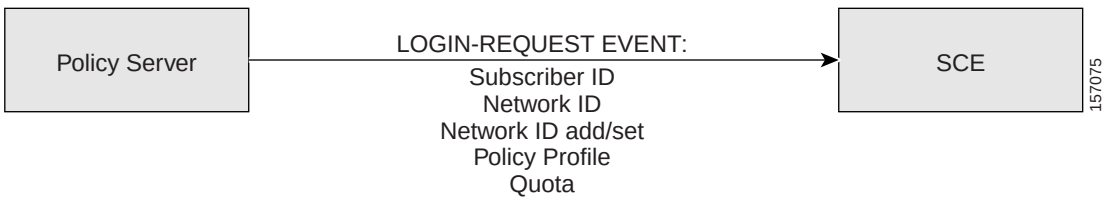
Login events occur when the subscriber connects to the network and vary for pull and push models.



### Push Model

The push integration model assumes that the Policy Server triggers the subscriber introduction to the SCE. For example, the server receives a subscriber login indication from an external entity such as AAA (Authorization, Authentication, and Accounting), extracts the required subscriber attributes, and "pushes" the information to the SCE platform:

***Figure 3-2 Login Events - Push Model***



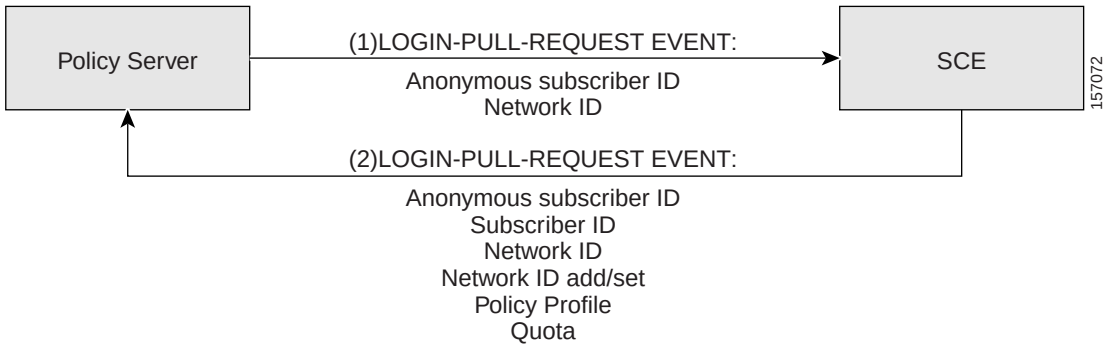

The subscriber login operation may either cause the creation of a new subscriber record in the SCE or update an existing subscriber. For example, for cable modem networks the subscriber is a cable modem and the CPEs connected to this cable modem are configured as a list of IP addresses (potentially ranges). In this case, the login of the new CPE connected to the same modem causes the CPE IP address to be added to the subscriber's Network ID list.

### Pull Model

The pull integration model assumes that the SCE discovers a new subscriber from the incoming data traffic. The new subscriber is entered in the system as an anonymous subscriber and is assigned one of the default policies. The SCE initiates a request to the external system (a login-pull request) that may either provide the subscriber login information (a login-pull reply) or is omitted if no information exists for this IP. The login information provided to the SCE replaces the anonymous subscriber with the actual subscriber and enforces the correct policy.

If the external system rejects the login and the traffic keeps coming from the anonymous subscriber, the pull request will be retried.

***Figure 3-3 Login Events - Pull Model***

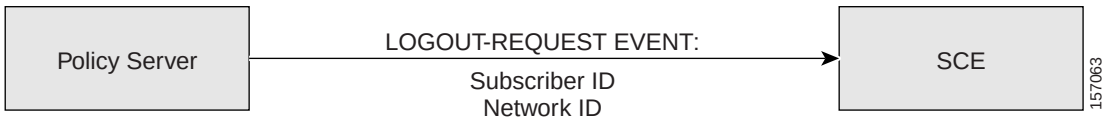



**Note** Despite being classified as "Network-ID Management Event", LOGIN-REQUEST event and LOGIN-PULL-RESPONSE event are optimized to allow sending all subscriber information to the SCE. It is recommended to use these events for Policy Profile and Quota updates when a single Policy Server performs all parts of the subscriber provisioning. For multiple Policy Servers topologies, use separate events for updating Policy Profile and Quota information described in the following sections. For more information about topologies, see the Supported Topologies.

## Logout Events

The logout event indicates that the subscriber no longer uses a certain network ID. A logout event is not necessarily followed by the removal of the subscriber record from the SCE. For example, in cable modem networks, when there are more than one CPE connected to the same modem, the logout of one CPE may not lead to the removal of a subscriber if another CPE remains connected. The actual removal of the subscriber occurs when all of the CPEs (Subscriber's network-IDs) are disconnected.

***Figure 3-4 Logout Request Event***

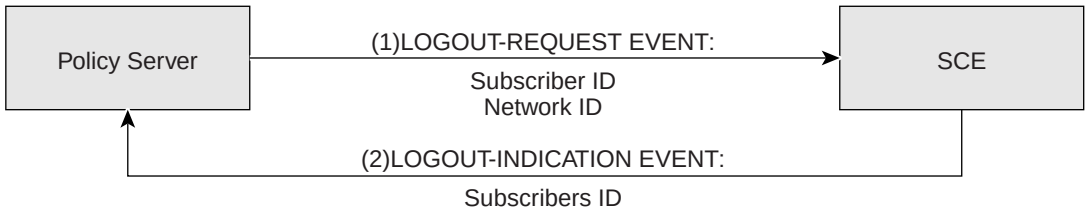



The logout event in the pull model may occur, for example, when the SCE identifies that the subscriber is not active for a specific time interval. The SCE "logs out" the subscriber and sends a LOGOUT-INDICATION event.

*Figure 3-5 Logout Indication Event*


The LOGOUT-INDICATION event may also follow the Logout operation. This occurs once a subscriber is actually removed; for example, when no more valid network mappings (IP) are associated with this subscriber.

*Figure 3-6 Logout Request Event*




## Network ID Update Event

This event is a REQUEST from the Policy Server to the SCE to update the network ID of the subscriber that already exists in the SCE platform. This event does not require any RESPONSE.

*Figure 3-7 Network ID Update Event*

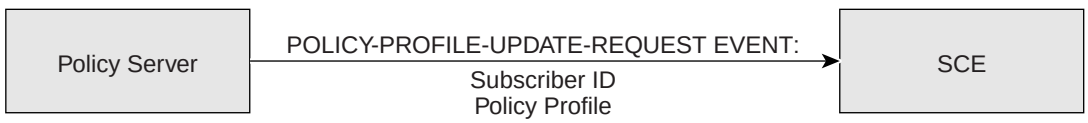

# Information About Policy Profile Management Events



## Profile Update Event

This event is a REQUEST from the Policy Server to the SCE to update the policy profile of the subscriber that already exists in the SCE platform. This event does not require any RESPONSE.

*Figure 3-8 Profile Update Event*



**Note** As described above, the LOGIN-REQUEST event and LOGIN-PULL-RESPONSE event can also update the policy profile.

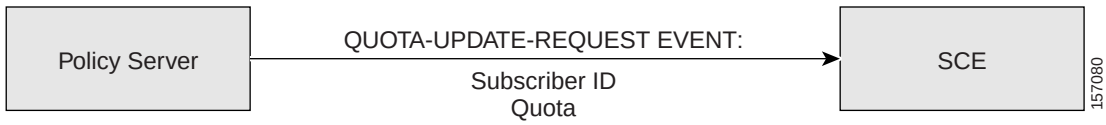
# Information About Quota Management Events



## Quota Update Event

The Quota Update Event is a REQUEST from the Policy Server to the SCE to update the quota of the subscriber that already exists in the SCE platform. This event does not require any RESPONSE event.

***Figure 3-9 Quota Update Event***

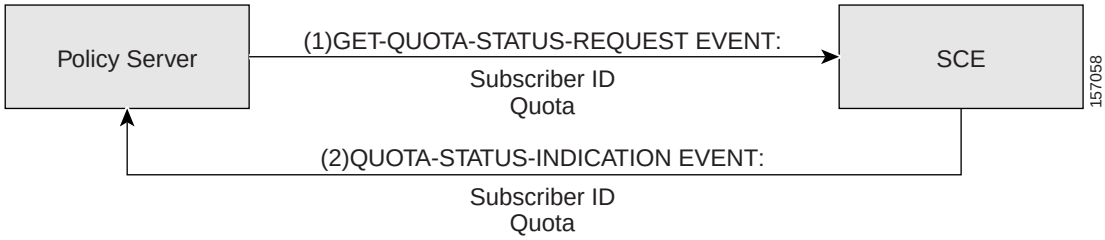



**Note** As described above, the LOGIN-REQUEST event and LOGIN-PULL-RESPONSE event can also update the quota.

## Get Quota Status Event

The Get Quota Status Event is a REQUEST from the Policy Server to the SCE to report the quota information of the subscriber that already exists in the SCE platform. A QUOTA-STATUS-INDICATION event follows this event.

***Figure 3-10 Get Quota Status Event***

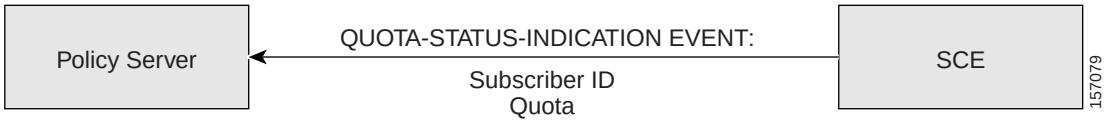


**Note** A QUOTA-STATUS-INDICATION event may be issued periodically by the SCE without a specific request from the Policy Server. See Quota Status Event.

## Quota Status Event

The SCE uses the Quota Status INDICATION event to notify the Policy Server about the remaining quota. This event is invoked periodically in a preconfigured time interval.

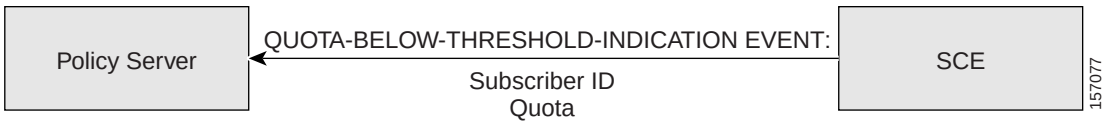
***Figure 3-11 Quota Status Event***




## Quota Below Threshold Event

The SCE uses the Quota Below Threshold INDICATION event to notify the Policy Server that the remaining quota for certain services of the specific subscriber is below the preconfigured threshold. An UPDATE-QUOTA-REQUEST event from the Policy Server to the SCE may follow this event, but it is not mandatory.

***Figure 3-12 Quota Below Threshold Event***




## Quota Depleted Event

The SCE uses the Quota Depleted INDICATION event to notify the Policy Server that the quota for certain services of the specific subscriber is depleted. An UPDATE-QUOTA-REQUEST event from the Policy Server to the SCE may follow this event.

***Figure 3-13 Quota Depleted Event***

## Quota State Restore Event

The Quota State Restore Event is an INDICATION from the SCE to the Policy Server to restore the quota of the subscriber that exists in the SCE platform. This event is invoked immediately after a subscriber is logged in to the SCE. A Quota Update event from the Policy Server may follow this event.

***Figure 3-14 Quota State Restore Event***


# Information About SCE Synchronization Procedure Events

- Start Synchronization Event
- End Synchronization Event
- Get Subscribers Events

## Start Synchronization Event

The Start Synchronization REQUEST event is used to notify the SCE that the synchronization process is about to start. The SCE uses this REQUEST to perform internal operations that are required for synchronization process preparation. This event has a push and a pull component.

***Figure 3-15 Start Synchronization Event***


## End Synchronization Event

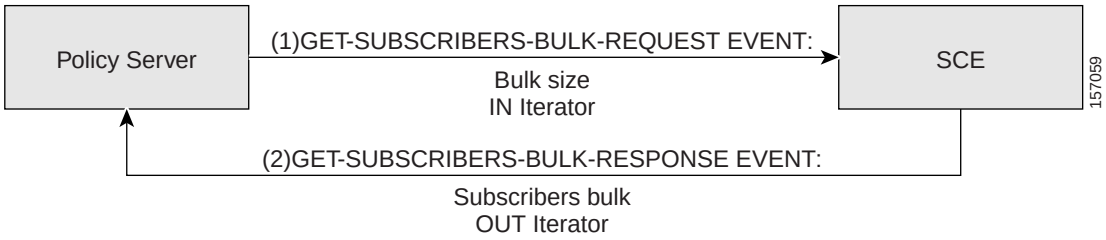
The End Synchronization REQUEST event is used to notify the SCE that the synchronization process has ended. This event has a push and a pull component.

***Figure 3-16 End Synchronization Event***

## Get Subscribers Events

During the SCE's Pull Model synchronization process, the Policy Server is required to retrieve ALL subscribers that the SCE is currently handling. The GET-SUBSCRIBERS-BULK-REQUEST event is a request from the Policy Server to the SCE to retrieve the next bulk of subscribers that the SCE is currently handling. Upon receiving this request, the SCE responds with the GET-SUBSCRIBERS-BULKRESPONSE event that supplies the subscriber names and Network-IDs.

***Figure 3-17 Get Subscribers Event***




For more information, see the Pull Model and Information About the Pull Model Synchronization Procedure.

CHAPTER 4

# Getting Familiar with the API Data Types

This module describes the various API data types used in the SCMS SCE Subscriber API.



## Subscriber ID

Most methods of the SCE Subscriber APIs require the subscriber ID to be used as an input parameter. The Subscriber ID is a string representing a subscriber name or a CM MAC address. This section lists the formatting rules of a subscriber ID.

The subscriber name is *case-sensitive* . It may contain up to 64 characters. All printable characters with an ASCII code between 32 and 126 (inclusive) can be used; except for 34 ("), 39 ('), and 96 (`).

For example:

```
String subID1="john";
String subID2="john@yahoo.com";
```

## Information About Network ID Mappings

A network ID is a network identifier that the SCE device relates to a specific subscriber record. A typical example of a network ID mapping is an IP address. Currently, the Cisco Service Control Engine (SCE) supports IP address, IP range, and VLAN types of mappings.

The NetworkID class represents various types of subscriber network identification.

The API supports the following subscriber mapping types:

- IP addresses or IP ranges
- VLAN tags

**Note** Mixing IP addresses/IP ranges with VLAN tags for the same subscriber is not supported.

When using subscriber operations that involve network ID, the caller is requested to provide a NetworkID parameter.

NetworkID class constructors are defined as follows:

```
public NetworkID(String mapping,short mappingType) throws Exception
public NetworkID(String[] mappings,short[] mappingTypes) throws Exception
```

Parameters of the NetworkID constructors are:

- A **java.lang.String** mapping identifier or array of mapping identifiers
- A short mapping type or array of mapping types

When passing arrays, the mappingTypes array must contain either the same number of elements as the mappings array, or a single element.

- Use NetworkID.TYPE_IP or NetworkID.TYPE_VLAN constants if the array contains more than one element
- Use NetworkID.ALL_IP_MAPPINGS or NetworkID.ALL_VLAN_MAPPINGS constants when a single array element is used

# Specifying IP Address Mapping

The string format of an IP address is the commonly used decimal notation:

```
IP-Address=[0-255].[0-255].[0-255].[0-255]
```

**Example:**

- `216.109.118.66`

  The mapping type of an IP address is provided in the class NetworkID:

- `com.scms.common.NetworkID.TYPE_IP:`

  **com.scms.common.NetworkID.ALL_IP_MAPPINGS** specifies that all the entries in the mapping identifiers array are IP mappings.

# Specifying IP Range Mapping

The string format of an IP range is an IP address in decimal notation and a decimal specifying the number of **1** s in a bit mask: **IP-Range=[0-255].[0-255].[0-255].[0-255]/[0-32]**.

**Examples:**

- **10.1.1.10/32** is an IP range with a full mask, that is, a regular IP address.
- **10.1.1.0/24** is an IP range with a 24-bit mask, that is, all of the addresses ranging between **10.1.1.0** and **10.1.1.255**.

**Note** The mapping type of an IP Range is identical to the mapping type of the IP address.

# Specifying VLAN Tag Mapping

The string format for VLAN tag mapping is a decimal number in the following range: **[2-2046]**

The **com.scms.common.NetworkID** class provides the VLAN mapping type:

- The mapping type of an IP address is provided in the class NetworkID:
- `com.scms.common.NetworkID.TYPE_VLAN:`
- **com.scms.common.NetworkID.ALL_VLAN_MAPPINGS** specifies that all the entries in the mapping identifiers array are VLAN mappings.

## Network ID Mappings Examples

Construct NetworkID with a single IP address:

```
NetworkID nid = new NetworkID("1.1.1.1",NetworkID.TYPE_IP)
```

Construct NetworkID with a range of IP addresses:

```
NetworkID nid = new NetworkID("1.1.1.1/24",NetworkID.TYPE_IP)
```

Construct NetworkID with multiple IP addresses:

```
NetworkID nid = new NetworkID(new String[]{"1.1.1.1","2.2.2.2","3.3.3.3"},
NetworkID.ALL_IP_MAPPINGS)
```

Construct NetworkID with a single VLAN address:

```
NetworkID nid = new NetworkID("23",NetworkID.TYPE_VLAN)
```

# Information About SCA BB Subscriber Policy Profile

The Policy Profile describes the subscriber policy information. A policy profile is generally comprised of two main parts including a statically defined policy that is identified by the policy package and a set of subscriber policy properties that might have a dynamic nature. The package ID identifies the policy package. Most of the rules enforced on the subscriber traffic are derivedfrom the package ID.

Subscriber policy property in SCA BB is a **key-value** pair that affects the way the SCE analyzes and reacts to network traffic generated by the subscriber.

More information about properties can be found in the Cisco Service Control Application Suite for Broadband User Guide.

SCA BB version 3.0 contains the following properties:

- packageId—Defines the package ID of the subscriber
- monitor—Indicates whether to issue an Raw Data Record (RDR) for each transaction of this subscriber

## PolicyProfile Class

The API provides a PolicyProfile class to format subscriber policy profiles required for the API operations.

The following method constructs the PolicyProfile class based on the array of policy properties:

```
public PolicyProfile(String[] policy)
```

**Note** The encoding of each string within the array **must** be as follows:

```
property_name=property_value
```

The following method allows adding a policy property to the profile according to the format described above:

```
public void addPolicyProperty(String policyProperty)
```

**Note** This method is not optimized for performance. For best performance results, use the PolicyProfile constructor.

**Example:**

```
PolicyProfile pp = new PolicyProfile(new
          String[]{"packageId=22", "monitor=1"})
```

# Information About Subscriber Quota

The quota provisioning in SCA BB is prepared using subscriber quota buckets. Each subscriber has 16 buckets, and you can define each bucket for volume or sessions. When a subscriber uses a particular service, the amount of consumed volume or number of sessions is subtracted from one of the buckets. The service configuration, which is defined in the general policy definition by using the SCA BB Console, determines which bucket to use for each service. Consumption for the volume buckets is counted in units of L3 kilobytes and consumption for the session buckets is the number of sessions. For example, it is possible to define that the Browsing and E-mail services consume quota from Bucket number 1, P2P service consumes quota from Bucket number 2, and that all other services are not bound to any particular bucket.

Quota bucket comprises from the following components:

- Bucket ID—Unique identifier of the bucket **(String)** as defined in the predefined policy. Valid values are numbers in range [1-16]
- Bucket value—Quota bucket value **(long)**

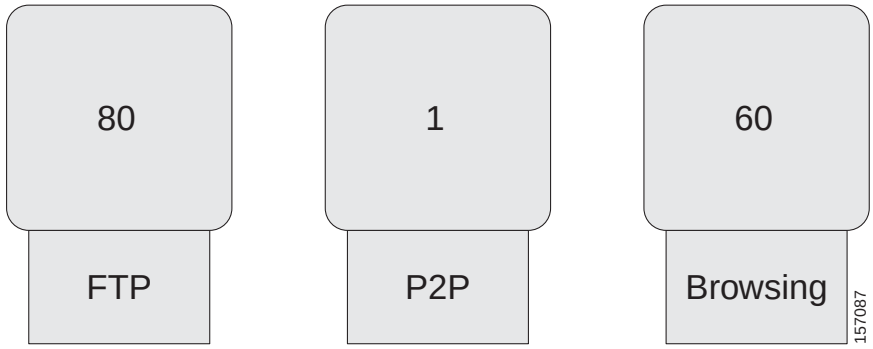
Quota Operation dynamically modifies a subscriber's quota buckets. There are two types of quota operations:

- ADD_QUOTA_OPERATION—Adds the new quota value to the current value of the bucket residing on the SCE platform
- SET_QUOTA_OPERATION—Replaces the value of the quota bucket residing on the SCE platform with the new value

**Examples**

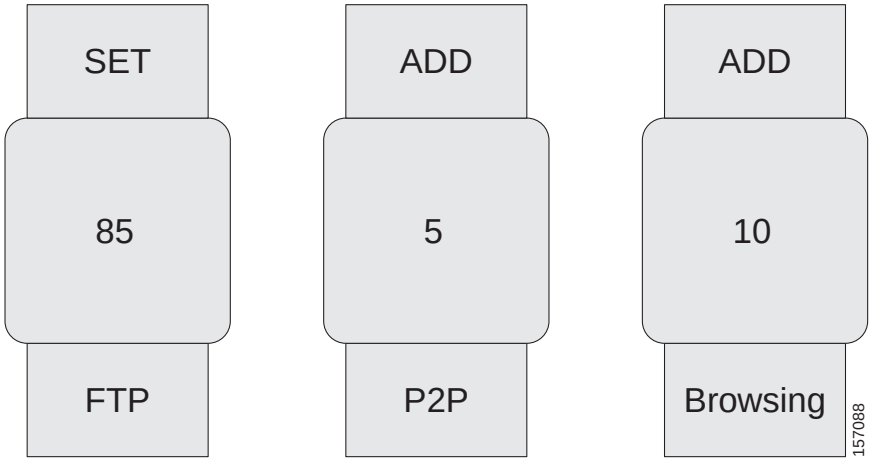
Current values of subscriber A's quota at the SCE are as follows:

***Figure 4-1 Subscriber Quota - Current Values***



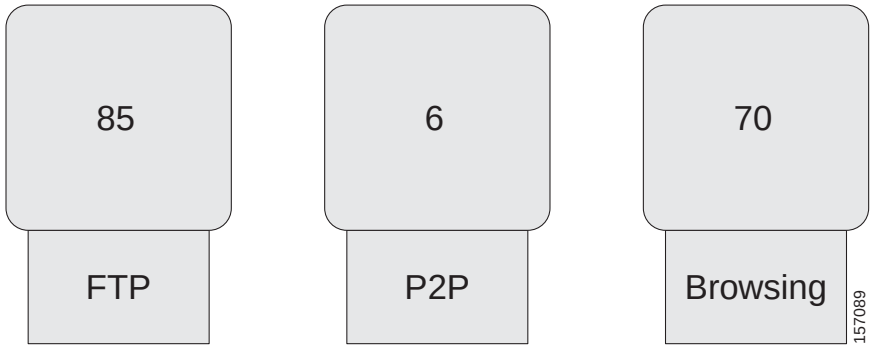
We want to apply the following actions to the existing quota:

***Figure 4-2 Subscriber Quota - Actions to Apply***




After performing the quota actions, the result is:

***Figure 4-3 Subscriber Quota - Results***




For additional information about Subscriber Quota, see the *Cisco Service Control Application for Broadband User Guide* .

The following sections describe the classes the API provides for operations that include the subscriber quota management operations.

# SCAS_BB_Quota

The SCAS_BB_Quota class implements the Quota interface, which the QuotaListenerEx interface uses in all callback functions. See Information About the QuotaListenerEx Interface Class.

The following method constructs the SCAS_BB_Quota based on the array of IDs and values:

```
public SCAS_BB_Quota (String[] bucketIDs,
long[] bucketValues)
```

The following method constructs the SCAS_BB_Quota based on the array of IDs and values, the profile ID, the reason, and the timestamp:

```
public SCAS_BB_Quota (String[] bucketIDs,
long[] bucketValues,
int quotaProfileId,
int reason,
long timestamp)
```

The following method allows retrieving of the quota buckets' IDs:

```
public String[] getBucketIDs()
```

The following method allows retrieving of the quota buckets' values:

```
public long[] getBucketValues()
```

The **quotaProfileId** parameter is the identifier for the quota profile, which is the package ID. The following method allows retrieving of the quota profile ID:

```
public int getQuotaProfileId()
```

The **reason** parameter is relevant only for quota status events and has three possible values:

- 0—The configured time was reached, for example, every two minutes
- 1—The quota status event was triggered by a subscriber logout
- 2—The quota status event was triggered by a package change

The following method allows retrieving of the reason:

```
public int getReason()
```

The **timestamp** parameter contains the time (in the SCE) at which the event was generated. It is calculated as the number of seconds from January 1, 1970 00:00 GMT.

The following method allows retrieving of the timestamp:

```
public long getTimestamp()
```

# SCAS_BB_QuotaOperation

The SCAS_BB_QuotaOperation class implements the QuotaOperation interface, which is used for Subscriber Provisioning operations that include the subscriber's quota such as login operation (see Information About the login Operation ) and update quota operation (see Information About the quotaUpdate Operation ).

The following method constructs the SCAS_BB_QuotaOperation based on the array of IDs, values and actions:

```
public SCAS_BB_QuotaOperation (String[] IDs,
long[] values,
short[] actions)
```

The following method allows retrieving of the quota buckets' IDs:

```
public String[] getBucketIDs()
```

The following method allows retrieving of the quota buckets' values:

```
public long[] getBucketValues()
```

The following method allows retrieving of the quota buckets' actions:

```
public short[] getBucketActions()
```

# Information About Bulk Operations Data Types

Use bulk classes and operations when performing the same method for many subscribers each with its own parameters. The API provides the bulk classes for result handling of bulk operations and for bulk indications from the SCE. The bulk classes are passed to the bulk methods such as **loginBulk** and **logoutBulk**.

The following is a list of considerations when using the bulk operations:

- All bulk classes are inherited from the common BulkBase class.
- Due to the memory constraints of the SCE, the bulk size is limited to a maximum of 100 entries.

## Bulk Iterator

The BulkBase class provides an iterator to view the data contained in the bulk.

The following is the syntax for the Bulk Iterator:

```
Iterator getIterator()
```

This iterator can be used for iteration over the bulks received from the SCE in various indications (for example, logoutBulkIndication, loginPullBulkResponseIndication, and so forth) or for inspecting the data you provided to various operations in case an operation has failed.

The iterator provides the following methods for data retrieval:

```
public Object next()
public boolean hasNext()
```

The next() method returns a SubscriberData object.

The SubscriberData class is used for retrieving the information of a single subscriber contained within the bulk.

### SubscriberData

The **SubscriberData** class represents all of the operations that can be performed on a specific subscriber. The SubscriberData class contains the following utility methods for information retrieval:

```
public String getSubscriberID()
public String getAnonymousID()
public String[] getMappings()
public short[] getTypes()
public boolean getAdditiveFlag()
```

The following sections describe various bulk data types that are available for different API operations.

## Login_BULK Class

This class represents bulk of subscribers and it includes all data required for the **loginBulk** operation.

## Constructor

To construct the **Login_BULK** filled with the data use the following constructor:

```
public Login_BULK(String[] subscriberIDs,
NetworkID[] networkIDs,
boolean[]networkIDsAdditive,
PolicyProfile[] policy,
QuotaOperation[] quota)
```

### Parameters

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

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

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

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

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

To construct an empty Login_BULK, use the following method:

```
public Login_BULK()
```

## addBulkEntry Method

Use the following method to add entries to the bulk:

```
public void addBulkEntry(String subscriberID,
NetworkID networkID,
boolean networkIdsAdditive,
PolicyProfile policy,
QuotaOperation quota)
```

### Parameters

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

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

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

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

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

## Examples

- Login_BULK Object Usage: Example
- Manipulating Login_BULK: Example

#### Login_BULK Object Usage: Example

This example demonstrates the usage of the Login_BULK object:

```
// Prepare all data for the bulk construction
String[] names = new String[5];
NetworkID[] mappings = new NetworkID[5];
boolean[] additive = new boolean[5];
PolicyProfile[] policy = new PolicyProfile[5];

for (int i=0; i<5; i++)
{
names[i]="sub_"+i;
mappings[i] = new NetworkID("1.1.1."+i,NetworkID.TYPE_IP);
additive[i] = true;
policy[i] = new PolicyProfile(new String[]{"packageId="+(i+1)});
}
// construct the bulk object
Login_BULK bulk = new Login_BULK(names,mappings,additive,policy,null);
// Now it can be used in loginBulk operation
sceApi.loginBulk(bulk,null);
```

#### Manipulating Login_BULK: Example

This example demonstrates an alternative way of manipulating the Login_BULK object:

```
// Construct the empty bulk
Login_BULK bulk = new Login_BULK ();
// Fill the bulk using addBulkEntry method:
for (int i=0; i<20; i++)
{
String name ="sub_"+i;
NetworkID mappings = new NetworkID(i+1);
boolean additive = true;
PolicyProfile policy = new PolicyProfile(
new String[]{"packageId="+(i+1)});
QuotaOperation quota = new SCAS_BB_QuotaOperation(
new String[]{"1","2","3"},
new long[]{80,80,0}
new short[]{SCAS_BB_QuotaOperation.ADD_QUOTA_OPERATION,
SCAS_BB_QuotaOperation.ADD_QUOTA_OPERATION,
SCAS_BB_QuotaOperation.SET_QUOTA_OPERATION});
bulk.addBulkEntry(name,mappings,additive,policy,quota);
}
// Now it can be used in loginBulk operation
sceApi.loginBulk(bulk,null);
```

# SubscriberID_BULK Class

The **logoutBulkIndication** callback function that requires only subscriber IDs to be entered uses the SubscriberID_BULK class. See Information About the logoutBulkIndication Callback Method.

- Constructors
- addBulkEntry Method

## Constructors

To construct the **SubscriberID_BULK** with Subscriber IDs data, use the following constructor:

```
public SubscriberID_BULK(String[] subscriberIDs)
```

To construct an empty **SubscriberID_BULK** , use the following method:

```
public SubscriberID_BULK()
```

#### Parameters

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

### addBulkEntry Method

Use the following method to add entries to the SubscriberID bulk:

```
addBulkEntry(String subscriberID)
```

#### Parameters

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

## NetworkAndSubscriberID_BULK Class

Use the NetworkAndSubscriberID_BULK class in bulk operations that require Subscriber IDs and NetworkIDs in the following operations:

- **getSubscribersBulkResponse** callback (see the Information About the LoginPullListener Interface Class section)
- **logoutBulk** operation (see Information About the logoutBulk Operation )
- **networkIDUpdateBulk** operation (see Information About the networkIdUpdateBulk Operation )

### Constructors

To construct the **NetworkAndSubscriberID_BULK** with the SubscriberID and NetworkID data, use the following constructor:

```
public NetworkAndSubscriberID_BULK(String[] subscriberIDs,
NetworkID[] networkIDs,
boolean[] netIdAdditive)
```

To construct an empty **NetworkAndSubscriberID_BULK** , use the following method:

```
public NetworkAndSubscriberID_BULK()
```

#### Parameters

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

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

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

### addBulkEntry Method

Use the following method to add entries to the bulk:

```
addBulkEntry(String subscriberID,
NetworkID networkID,
boolean netIdAdditive)
```

#### Parameters

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

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

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

# LoginPullResponse_BULK Class

This class represents a bulk of subscribers and includes all data required for the **loginPullResponseBulk** method.

- Constructors
- addBulkEntry Method

## Constructors

To construct the **LoginPullResponse_BULK** containing the relevant data, use the following constructor:

```
public LoginPullResponse_BULK(String[] anonymousSubscriberIDs,
String[] subscriberIDs,
NetworkID[] networkIDs,
boolean[] networkIdsAdditive,
PolicyProfile[] policy,
QuotaOperation[] quota)
```

To construct an empty **LoginPullResponse_BULK** , use the following method:

```
public LoginPullResponse_BULK()
```

- Parameters

#### Parameters

**anonymousSubscriberID** —The identifier of the anonymous subscriber. This is sent by the SCE within the loginPullRequest/loginPullBulkRequest indication (see Information About the loginPullRequest Callback Method and Information About the loginPullRequestBulk Callback Method ). See Information About Subscriber Integration Models for more information.

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

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

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

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

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

## addBulkEntry Method

Use the following method to add entries to the bulk:

```
public addBulkEntry(String anonymousSubscriberID,
String subscriberID,
NetworkID networkID,
boolean networkIdAdditive,
PolicyProfile policy,
QuotaOperation quota)
```

### Parameters

**anonymousSubscriberID** —The identifier of the anonymous subscriber. This is sent by the SCE within the loginPullRequest/loginPullBulkRequest indication (see Information About the loginPullRequest Callback Method and Information About the loginPullRequestBulk Callback Method ). See Information About Subscriber Integration Models for more information.

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

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

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

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

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

# PolicyProfile_BULK Class

The **updatePolicyProfileBulk** operation uses this class that represents a bulk of subscriber IDs and subscriber policy profiles.

- Constructors
- addBulkEntry Method

## Constructors

To construct the **PolicyProfile_BULK** containing the relevant data, use the following constructor:

```
public PolicyProfile_BULK(String[] subscriberIDs, PolicyProfile[] policy)
```

To construct an empty **PolicyProfile_BULK** , use the following method:

```
public PolicyProfile_BULK()
```

### Parameters

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

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

## addBulkEntry Method

Use the following method to add entries to the bulk:

```
public addBulkEntry(String subscriberID,  PolicyProfile policy)
```

### Parameters

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

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

# Quota_BULK Class

The following operations use this class that represents a bulk of subscribers IDs and subscriber quota buckets:

- getQuotaStatusBulk operation (only the bucket IDs are to be provided)
- quotaStatusBulkIndication callback method
- quotaDepletedBulkIndication callback method
- quotaBelowThresholdIndication callback method

## Constructors

To construct the **Quota_BULK** containing the relevant data, use the following constructor:

```
public Quota_BULK(String[] subscriberIDs, Quota[] subscribersQuota)
```

To construct an empty Quota_BULK, use the following method:

```
public Quota_BULK()
```

### Parameters

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

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

## addBulkEntry Method

Use the following method to add entries to the bulk:

```
public addBulkEntry(String subscriberID,Quota quota)
```

### Parameters

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

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

# QuotaOperation_BULK Class

The **QuotaUpdateBulk** operation and the login operation use this class that represents a bulk of subscribers IDs and subscriber Quota operations.



## Constructors

To construct the **QuotaOperation_BULK** containing the relevant data, use the following constructor:

```
public QuotaOperation_BULK(String[] subscriberIDs,
QuotaOperation[]quotaOperations)
```

To construct an empty QuotaOperation_BULK, use the following method:

```
public QuotaOperation_BULK()
```

### Parameters

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

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

## addBulkEntry Method

Use the following method to add entries to the bulk:

```
addBulkEntry(String subscriberID, QuotaOperation quotaOperation)
```

### Parameters

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

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

CHAPTER 5

# Programming with the SCE Subscriber API

This module provides a detailed description of the API programming structure, classes, methods, and interfaces.



## Information About API Classes

The following list maps the classes provided by the API.



### Package com.scms.api.sce.prpc

PRPC_SCESubscriberApi Class —Main API class.

### Package com.scms.api.sce

- SCE Cascade Topology Support
- Operations Result Handling

### Indications Listeners

- Information About the LoginPullListener Interface Class (interface)
- Information About the LogoutListener Interface Class (interface)
- Information About the QuotaListenerEx Interface Class (interface)

### Connection Monitoring

- ConnectionListener Interface (interface)

### SCE Cascade Topology Support

- Information About the RedundancyStateListener Interface (interface)

### Operations Result Handling

- OperationException Class (class)
- SCESubscriberApi (interface)—Contains error codes constants that can be received inside OperationException
- Information About the OperationArguments Class (class)
- Information About the OperationResultHandler Interface (interface)

## Package com.scms.common

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

- Login_BULK Class
- LoginPullResponse_BULK Class
- NetworkAndSubscriberID_BULK Class
- PolicyProfile_BULK Class
- SubscriberID_BULK Class
- SubscriberData (class)
- SCAS_BB_Quota (class)
- SCAS_BB_QuotaOperation (class)
- Information About Network ID Mappings NetworkID class
- PolicyProfile Class

# Programming Guidelines



## Programming with Callback Methods

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

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

This caution applies to the following operations:

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

# PRPC_SCESubscriberApi Class

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

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

## API Construction

The PRPC_SCESubscriberAPI provides the following constructors:

**Syntax:**

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

**Parameters:**

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

**apiName** —Specifies an API name.

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

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

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

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

- If the value is 0 or less, the reconnection task is not activated (no auto-reconnect is attempted).
- If the value is greater than 0 and a connection failure exists, the reconnection task will be activated every <autoReconnectInterval>milliseconds.
- Default value: -1 (no auto-reconnect is attempted)

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

**Examples:**

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

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

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

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

## Listeners Setup Operations

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

The listeners setup operations may include:

- Setting a connection listener, described in more detail in Information About Connection Monitoring section:
- `public void setConnectionListener(ConnectionListener listener)`
- Setting a login-pull listener, described in more detail in Information About the LoginPullListener Interface Class section:
- `public void registerLoginPullListener(LoginPullListener listener)`
- Setting a logout listener, described in more detail in Information About the LogoutListener Interface Class section:

- `public void registerLogoutListener(LogoutListener listener)`
- Setting a quota listener, described in more detail in Information About the QuotaListenerEx Interface Class section:
- `public void registerQuotaListener(QuotaListener listener)`
- Setting a redundancy state listener, described in more detail in Information About the RedundancyStateListener Interface section:
- `public void setRedundancyStateListener(RedundancyStateListener listener)`

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

## Advanced Setup Operations

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

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

The following properties can be set:

- Output queue size—The internal buffer size defining the maximum number of requests that can be accumulated by the API until they are sent to the SCE (Default: 1024)
- Operation timeout—A suggested time interval about the desired timeout (in milliseconds) on a non-responding PRPC protocol connection (Default: 45 seconds)

### Syntax

The syntax for the init method is as follows:

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

### Parameters

**properties** (java.util.Properties)—Enables setting the properties described in Advanced Setup Operations :

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

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

### Customize Properties: Example

This example shows how to customize properties during initialization:

```
// API construction
PRPC_SCESuscriberAPI sceApi = new PRPC_SCESuscriberAPI("MyApi",
"10.1.1.1",10000);
```

```
// API initialization
java.util.Properties p = new java.util.Properties();
p.setProperty("prpc.client.output.machinemode.recordnum", 2048+"");
api.init(p);
// connect to the API
sceApi.connect();
```

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

## Connecting to the SCE

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

To connect to the SCE, use the following methods:

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

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

```
public boolean isConnected()
```

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

## Information About getApiVersion

- Syntax
- Description

### Syntax

```
public String getApiVersion()
```

### Description

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

## API Finalization

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

```
public void disconnect()
```

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

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

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

```
        {
                sceapi.disconnect();
        }
}
```

# Information About Indications Listeners

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

- Login-pull indications
- Logout indications
- Quota indications

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

## Information About the LoginPullListener Interface Class

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

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

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

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

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

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

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

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

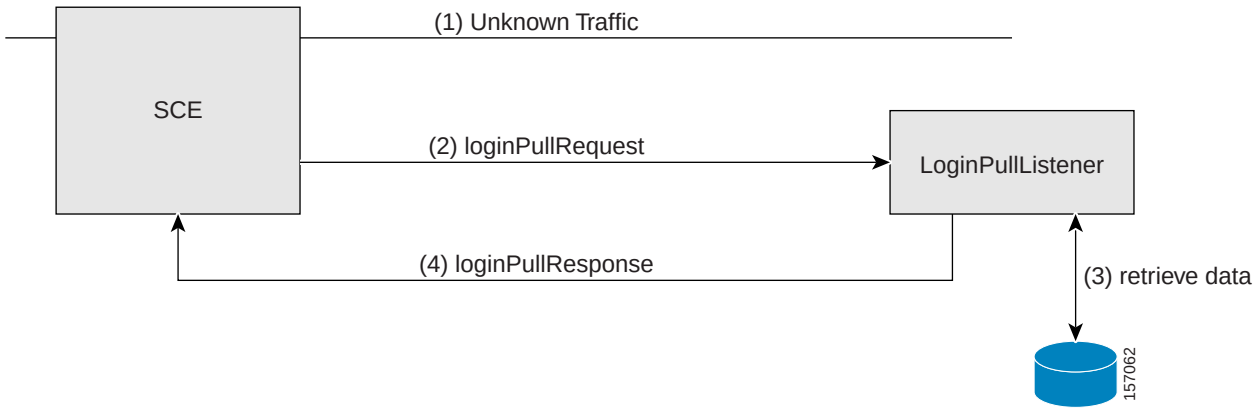
### Information About the loginPullRequest Callback Method

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

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

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

***Figure 5-1 LoginPullRequest Callback Method***




**Parameters**

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

## Information About the loginPullRequestBulk Callback Method

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

**Parameters**

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

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

## GetSubscribersBulkResponse Callback Method

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

# Information About the LogoutListener Interface Class

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

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

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

**Note** The API supports one LogoutListener at a time.

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

- Information About the logoutIndication Callback Method
- Information About the logoutBulkIndication Callback Method

## Information About the logoutIndication Callback Method

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

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

### Parameters

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

## Information About the logoutBulkIndication Callback Method

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

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

### Parameters

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

# Information About the QuotaListenerEx Interface Class

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

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

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

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

**Note** The API supports one QuotaListener at a time.

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

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

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

- Information About the quotaStatusIndication Callback Method
- Information About the quotaStatusBulkIndication Callback Method
- Information About the quotaBelowThresholdIndication Callback Method
- Information About the quotaBelowThresholdIBulkndication Callback Method
- Information About the quotaDepletedIndication Callback Method
- Information About the quotaDepletedBulkIndication Callback Method
- Information About the quotaStateRestore Callback Method
- Information About the quotaStateBulkRestore Callback Method

## Information About the quotaStatusIndication Callback Method

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

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

### Parameters

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

## Information About the quotaStatusBulkIndication Callback Method

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

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

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

#### Parameters

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

## Information About the quotaBelowThresholdIndication Callback Method

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

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

#### Parameters

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

## Information About the quotaBelowThresholdIBulkndication Callback Method

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

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

#### Parameters

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

## Information About the quotaDepletedIndication Callback Method

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

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

#### Parameters

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

## Information About the quotaDepletedBulkIndication Callback Method

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

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

#### Parameters

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

### Information About the quotaStateRestore Callback Method

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

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

#### Parameters

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

### Information About the quotaStateBulkRestore Callback Method

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

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

#### Parameters

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

# Information About Connection Monitoring

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

- ConnectionListener Interface
- Disconnect Listener: Example

## ConnectionListener Interface

The API allows setting a connection listener.

```
setConnectionListener(ConnectionListener listener)
```

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

```
public interface ConnectionListener {
/**
* called when the connection with the SCE is down.
*/
public void connectionIsDown();
/**
* called when the connection with the SCE is established.
*/
public void connectionEstablished();
}
```

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

## Disconnect Listener: Example

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

```
import com.scms.api.sce.ConnectionListener;
public class MyConnectionListener implements ConnectionListener {
public void connectionIsDown(){
System.out.println("Message: connection is down.");
return;
}
public void connectionEstablished(){
System.out.println("Message: connection is established.");
// activate thread that starts SCE synchronization
}
}
```

# Information About SCE Cascade Topology Support

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

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

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

- isRedundancyStatusActive Method
- Information About the RedundancyStateListener Interface
- Configuring the SCE to Ignore Cascade Violation Errors

## isRedundancyStatusActive Method

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

```
public boolean isRedundancyStatusActive()
```

This return value from this method has the following meaning:

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

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

# Information About the RedundancyStateListener Interface

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

```
setRedundancyStateListener(RedundancyStateListener listener)
```

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

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

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

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

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

## Parameters

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

# Configuring the SCE to Ignore Cascade Violation Errors

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

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

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

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

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

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

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

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

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

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

# Information About Result Handling

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

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

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

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

## Information About the OperationResultHandler Interface

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

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

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

}
```

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

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

The following is the data returned from the OperationResultHandler interface:

- **result** —The actual result of the operation - each entry within the array can be one of the following:
  - **NULL** —indicates success of the operation.
  - **OperationException** —indicates operation failure (see below). For non-bulk operations, the result array will have only one entry.
- For bulk operations, each entry of the result array corresponds to the relevant entry in the bulk operation.

- **handback** —The API automatically provides this object to every operation call. It contains the information about the operation that was called, including all arguments that were passed at the time of the call. The input arguments of the operation are retrieved by the argument name in the API documentation. For example, this data can be used to inspect/output the parameters after the operation failed or to repeat the operation call.

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

## Information About the OperationArguments Class

Use the following methods to retrieve the operation name:

```
public String getOperationName()
```

Use the following methods to retrieve the arguments names:

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

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

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

### Examples

Sample implementation of the OperationResultHandler interface:

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

public void handleOperationResult(Object[] result,
OperationArguments handback)
{
for (int index=0; index <result.length; index++)
{
if (result[index]==null)
{
// success
successCounter++;
}
else
{
// failure
errorCounter++;
// Extract error details
OperationException ex = (OperationException)result[index];
// Extract operation name
String operationName = handback.getOperationName();

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

```
handback.getArgument(argNames[j]));
}
}
}
}
}
}
```

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

The following example demonstrates login operation sample result handler:

```
public class LoginOperationHandler implements OperationResultHandler
{

public void handleOperationResult(Object[] result,
OperationArguments handback)
{
for (int index=0; index <result.length; index++)
{
if (result[index]!=null)
{
// failure
// Extract error details
OperationException ex =
(OperationException)result[index];
// Print operation name and error message
System.out.println("Error for login operation "+
":" + ex.getErrorMessage());
// Print subscriber ID parameter value
System.out.println("subscriberID"+
handback.getArgument("subscriberID"));
}
}
}
}
```

#### OperationException Class

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

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

See List of Error Codes for more details about error codes and their meaning.

# Information About Subscriber Provisioning Operations

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

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

# Information About the login Operation



## Syntax

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

## Description

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

- If the subscriber ID does not exists in the SCE, a new subscriber is added with all the data supplied
- If the subscriber ID exists:
  - If the **networkIdAdditive** flag is set to TRUE, the supplied NetworkID is added to the existing networkIDs of the subscriber. Otherwise, the supplied networkID replaces the existing networkIDs.
  - **policy** —Policy is *updated* with the new policy values. Subscriber Policy entries that are not provided within the PolicyProfile remain unchanged or created with default values.

  - **quota** —The quota is *updated* according to the bucket values and the operations provided, see Information About Subscriber Quota.

- If there is a networkID congestion with another subscriber, the networkID of the other subscriber is logged out implicitly and the new subscriber is logged in.

For relevant events description, see Push Model.

## Parameters

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

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

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

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

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

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

## Error Codes

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

- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_RESOURCE_SHORTAGE
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

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

## Examples

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

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

This example adds the IP address 192.168.12.5 overriding previous mappings:

```
login(
"john",                              // subscriber name
new NetworkID(new String[]{"192.168.12.5"},
SCESubscriberApi.ALL_IP_MAPPINGS),
```

```
false,                          // isMappingAdditive is false
null,                           // no policy
null);                          // no quota
```

For more examples, see the Login and Logout section.

# Information About the loginBulk Operation

- Syntax
- Description
- Parameters
- Error Codes

## Syntax

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

## Description

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

## Parameters

**subsBulk** —See the Login_BULK Class section.

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

## Error Codes

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

- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_RESOURCE_SHORTAGE
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

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

# Information About the loginPullResponse Operation

- Syntax
- Description
- Parameters
- Error Codes

### Syntax

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

### Description

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

For relevant events description, see Pull Model.

### Parameters

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

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

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

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

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

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

### Error Codes

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

- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_RESOURCE_SHORTAGE
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

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

## Information About the loginPullResponseBulk Operation

### Syntax

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

### Description

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

For relevant events description, see Pull Model.

### Parameters

subsBulk—See the LoginPullResponse_BULK Class section.

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

### Error Codes

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

- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_RESOURCE_SHORTAGE
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

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

## Information About the logout Operation



### Syntax

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

### Description

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

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

For relevant events description, see Logout Events.

### Parameters

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

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

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

### Error Codes

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

- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED

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

## Information About the logoutBulk Operation



### Syntax

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

### Description

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

For relevant events description, see Logout Events.

### Parameters

**subsBulk** —See the NetworkAndSubscriberID_BULK Class section.

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

### Error Codes

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

- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED

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

## Information About the networkIdUpdate Operation



### Syntax

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

### Description

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

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

For relevant events description, see Network ID Update Event.

### Parameters

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

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

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

### Error Codes

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

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

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

# Information About the networkIdUpdateBulk Operation



## Syntax

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

## Description

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

For relevant events description, see Network ID Update Event.

## Parameters

**subsBulk** —See the NetworkAndSubscriberID_BULK Class section.

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

## Error Codes

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

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

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

# Information About the profileUpdate Operation



## Syntax

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

## Description

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

For relevant events description, see Profile Update Event.

## Parameters

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

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

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

## Error Codes

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

- ERROR_CODE_SUBSCRIBER_NOT_EXIST
- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

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

# Information About the profileUpdateBulk Operation

### Syntax

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

### Description

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

For relevant events description, see Profile Update Event.

### Parameters

**subsBulk** —See the PolicyProfile_BULK Class section.

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

### Error Codes

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

- ERROR_CODE_SUBSCRIBER_NOT_EXIST
- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

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

## Information About the quotaUpdate Operation



### Syntax

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

### Description

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

For relevant event description, see Quota Update Event.

### Parameters

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

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

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

### Error Codes

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

- ERROR_CODE_SUBSCRIBER_NOT_EXIST
- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

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

## Information About the quotaUpdateBulk Operation



### Syntax

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

### Description

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

For relevant event description, see Quota Update Event.

### Parameters

**subsBulk** —See the QuotaOperation_BULK Class section.

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

### Error Codes

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

- ERROR_CODE_SUBSCRIBER_NOT_EXIST
- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

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

## Information About the getQuotaStatus Operation



### Syntax

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

### Description

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

For relevant events description, see Get Quota Status Event.

### Parameters

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

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

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

### Error Codes

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

- ERROR_CODE_SUBSCRIBER_NOT_EXIST
- ERROR_CODE_FATAL_EXCEPTION

- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

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

# Information About the getQuotaStatusBulk Operation



## Syntax

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

## Description

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

For relevant events description, see Get Quota Status Event.

## Parameters

**subsBulk** —See the Quota_BULK Class section.

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

## Error Codes

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

- ERROR_CODE_SUBSCRIBER_NOT_EXIST
- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

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

# Information About SCE-API Synchronization

In cases when the SCE and the Policy Server have a conflict in the data about a subscriber because of disconnection, loss of logon messages, or reboot, several problems can arise. These problems can cause a misclassification of one the subscriber's traffic as if it was another subscriber, enforcement of the wrong service on the subscriber's traffic, or loss of resources.

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

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

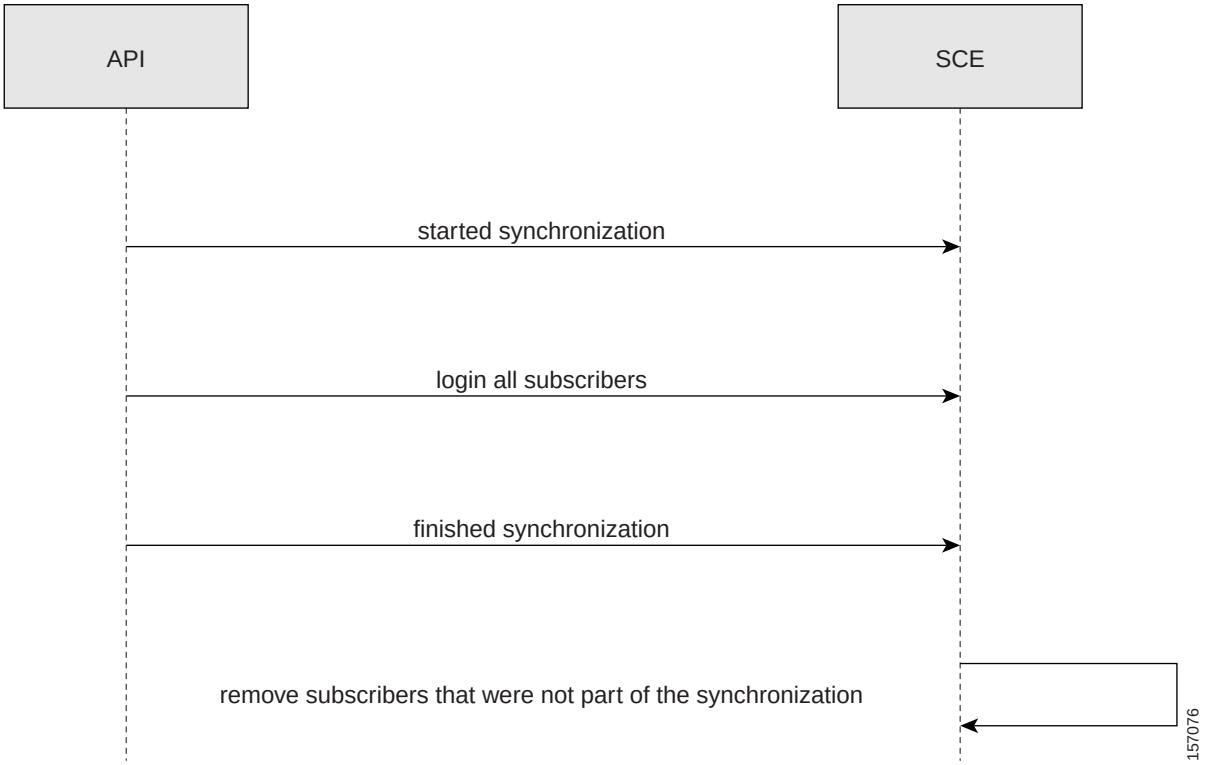
The following list describes the synchronization guidelines the Policy Server must adhere to while implementing synchronization:

- Information About the Push Model Synchronization Procedure
- Information About the Pull Model Synchronization Procedure

## Information About the Push Model Synchronization Procedure

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

*Figure 5-2 Push Model Synchronization Procedure*




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

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

- Information About synchronizePushStart
- Information About synchronizePushEnd

## Information About synchronizePushStart

- Syntax
- Description
- Parameters

### Syntax

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

### Description

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

#### Parameters

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

## Information About synchronizePushEnd



#### Syntax

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

#### Description

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

#### Parameters

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

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

# Information About the Pull Model Synchronization Procedure

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

### Algorithm

Use the following algorithm template when planning the synchronization procedure:

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

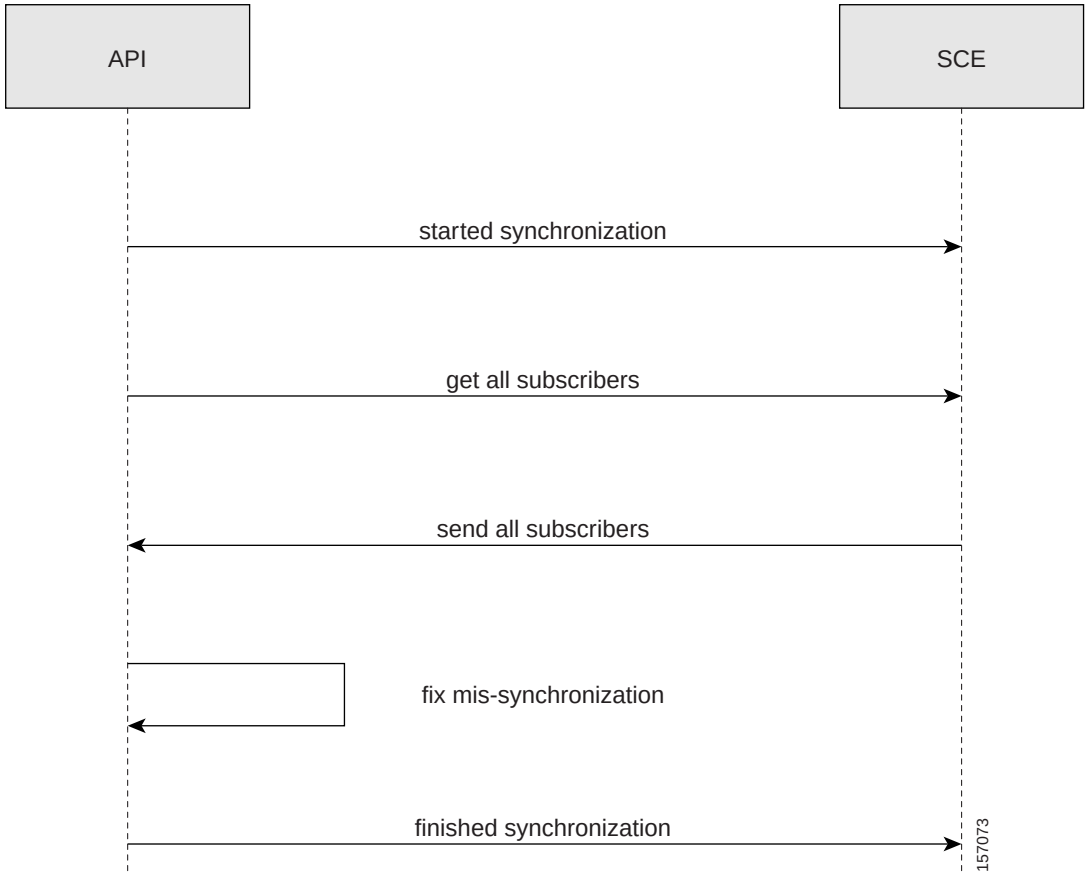
- If <SubscriberID, IP address >exists in the Policy Server database:

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

  send a logout with the Subscriber IP to the SCE

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

*Figure 5-3 Pull Model Synchonization Procedure*




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

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

- Information About synchronizePullStart
- Information About synchronizePullEnd
- Information About getSubscribersBulk

## Information About synchronizePullStart

- Syntax
- Description
- Parameters

### Syntax

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

### Description

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

### Parameters

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

## Information About synchronizePullEnd

### Syntax

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

### Description

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

### Parameters

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

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

## Information About getSubscribersBulk

### Syntax

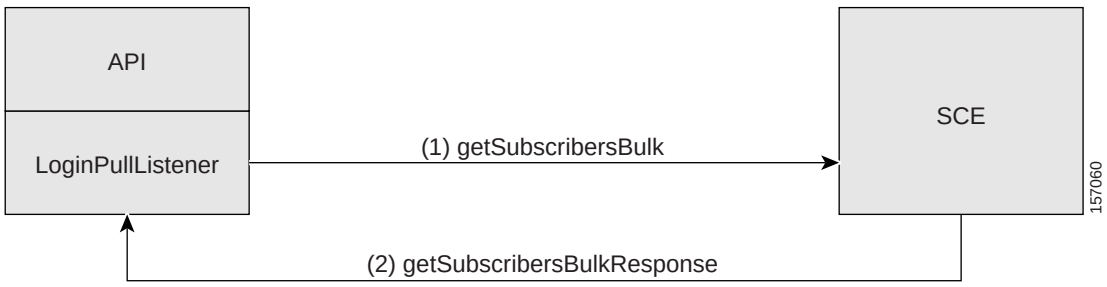
```
void getSubscribersBulk(int bulkSize,
SubscribersBulkResponseIterator iterator,
OperationResultHandler handler)
```

### Description

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

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

***Figure 5-4 Get Subscribers Bulk Description***



**Parameters**

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

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

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

# Information About Advanced API Programming

## Implementing High Availability

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

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

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

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

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

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

# API Code Examples

This section gives several code examples for the API usage:

- Login and Logout
- Login-pull Request and login-pull Response

# Login and Logout

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

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

```
// Class responsible for operations result handling
import com.scms.api.sce.OperationArguments;
import com.scms.api.sce.OperationException;
import com.scms.api.sce.OperationResultHandler;
public class MyOperationResultHandler implements OperationResultHandler
{
long count = 0;

public void handleOperationResult(Object[] result,
OperationArguments handback)
{
for (int index=0; index <result.length; index++)
{
count++;
if (result[index]==null)
{
//print success every 100 operations
//if (++count%100 == 0)
{
System.out.println("\tsuccess "+count);
}
}
else // error - print every error
{
// failure
count++;
// Extract error details
OperationException ex =
(OperationException)result[index];
// Extract operation name
String operationName = handback.getOperationName();
// Print operation name and error message
System.out.println("Error for operation "+
operationName+": "+
ex.getMessage());
}
}
}
public synchronized void waitForLastResult(int lastResult)
{
while (count<lastResult)
{
try
{
wait(100);
}
catch (InterruptedException ie)
{
ie.printStackTrace();
}
}
}
}
```

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

```
import com.scms.api.sce.LogoutListener;
import com.scms.common.NetworkAndSubscriberID_BULK;
import com.scms.common.SubscriberID_BULK;
class MyLogoutListener implements LogoutListener
{
long count = 0;

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

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

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

public void logoutBulkIndication(SubscriberID_BULK subs)
{
increaseCounter(subs.getSize());
}
}
```

Class that contains the **main** method:

```
import com.scms.api.sce.prpc.PRPC_SCESubscriberApi;
import com.scms.common.*;
public class LogonPolicyServer {
public static void main (String args[]) throws Exception
{
int numSubscribersToLogin = 500;
//instantiate an API with reconnect interval of 5 seconds
PRPC_SCESubscriberApi api = new PRPC_SCESubscriberApi(
"myAPI",
args[0], // IP of the SCE
5000);
try {
// instantiate operation result handler
// we will use one handler for all operations
MyOperationResultHandler resultHandler =
new MyOperationResultHandler();
// instantiate logout listener
MyLogoutListener listener = new MyLogoutListener();
// register to logout indications
api.registerLogoutListener(listener);
```

```
// connect to the SCE
api.connect();
//login
System.out.println("login of "+numSubscribersToLogin+
" subscribers");
PolicyProfile pp = new PolicyProfile(
new String[]{"packageId=1",
"monitor=1"});
for (int i=0; i<numSubscribersToLogin; i++)
{
api.login("sub"+i,
new NetworkID(getMappings(i), // generate ip
NetworkID.ALL_IP_MAPPINGS),
true,                          // additive flag
pp,                            // policy
null,                          // no quota
resultHandler);
}
// wait for subscribers to log in
resultHandler.waitForLastResult(numSubscribersToLogin);
// logout all subscribers
System.out.println("logout of "+numSubscribersToLogin+
" subscribers");
for (int i=0; i<numSubscribersToLogin; i++)
{
NetworkID nid = new NetworkID(getMappings(i),
NetworkID.ALL_IP_MAPPINGS);
api.logout("sub"+i,nid,resultHandler);
}
// wait for all subscribers to be logged out -
// but this time use
// logout listener to count the results
listener.waitForLastResult(numSubscribersToLogin);
}
finally
{
api.unregisterLogoutListener
api.disconnect();
}
}
//'automatic' mapping generator for the sample program
private static String[] getMappings(int i) {
return new String[]{ "10." +((int)i/65536)%256 + "." +
((int)(i/256))%256 + "." + (i%256)};
}
}
```

# Login-pull Request and login-pull Response

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

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

```
import java.util.Iterator;
// result handler from the previous example
import MyOperationResultHandler;
import com.scms.api.sce.*;
import com.scms.common.*;
class MyListener implements LoginPullListener, LogoutListener
{
// indications counters
long logoutCount = 0;
```

```
long pullCount=0;
// api instance – used to send login-pull responses to the SCE
PRPC_SCESubscriberApi api = null;

// construct operation handler -
// from previous (Login and Logout) example
MyOperationResultHandler h = new MyOperationResultHandler();
public MyListener(PRPC_SCESubscriberApi api)
{
this.api = api;
}
// Increase logout counter
public void logoutIndication(String subscriberID)
{
increaseLogoutCounter(1);
System.out.println("Got logout notification " +
getLogoutCounter());
}
// Increase logout counter
public void logoutBulkIndication(SubscriberID BULK subs)
{
System.out.println("Got logout notification");
increaseLogoutCounter(subs.getSize());
}
public void loginPullRequest (String anonymousSubscriberID,
NetworkID networkID)
{
try
{
increasePullCounter(1);
System.out.println("Got pull request" + getPullCounter());

// prepare policy
PolicyProfile pp = new PolicyProfile(
new String[]{"packageId=1",
"monitor=1"});
// Answer with pull response
// retrieve subscriber name – for example from your
// policy server database
// In this example we use fixed names based on the
// subscribers counter
api.loginPullResponse(anonymousSubscriberID,
"sub"+getPullCounter(),
networkID,
pp,     // policy
null,   // no quota
h); // handler from previous example
}
catch (Exception ex)
{
System.out.println(ex.getMessage());
}
}
public void loginPullRequestBulk(NetworkAndSubscriberID BULK subs)
{
try
{
increasePullCounter(subs.getSize());
System.out.println("Got pull request" + getPullCounter());
// Answer with pull response in bulk form
PolicyProfile pp = new PolicyProfile(
new String[]{"packageId=1",
"monitor=1"});
LoginPullResponse_BULK responseBulk =
```

```
new LoginPullResponse_BULK();
Iterator subsIterator = subs.getIterator();
// iterate of the received bulk (IPs and anonymous IDs)
// and build a response bulk
int count=0;
while(subsIterator.hasNext())
{
// retrieve subscriber name – for example from your
// policy server database
// In this example we use fixed names based on the
// subscribers counter
String subName = "sub_"+count;
SubscriberData sub = (SubscriberData)subsIterator.next();
// Extract subscriber mappings from the bulk and
// constract a new NetworkID based on those mappings
NetworkID subNetId = new NetworkID(sub.getMappings(),
NetworkID.ALL_IP_MAPPINGS);
responseBulk.addEntry(sub.getAnonymousSubscriberID(),
subName,
subNetId,
true,
pp,
null);
count++;
}
//use the bulk constructed above in the bulk response
//use handler from the previous example
api.loginPullBulkResponse(responseBulk,h);
}
catch (Exception ex)
{
System.out.println(ex.getMessage());
}
}
public void getSubscribersBulkResponse(
NetworkAndSubscriberID BULK subs,
SubscruberBulkResponseIterator iterator)
{
//  not implemented in this example
}

synchronized void increaseLogoutCounter(long value)
{
logoutCount = logoutCount + value;
}
synchronized void increasePullCounter(long value)
{
pullCount = pullCount + value;
}
synchronized long getPullCounter()
{
return pullCount;
}
synchronized long getLogoutCounter()
{
return logoutCount;
}
//waits for result number 'last result' to arrive
public synchronized void waitForPullResult(int lastResult) {
while (pullCount<lastResult) {
try {
wait(100);
} catch (InterruptedException ie) {
ie.printStackTrace();
```

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

Class that contains the **main** method:

```
import java.util.Iterator;
import com.scms.api.sce.*;
import com.scms.common.*;
public class LogonPolicyServer {
static PRPC_SCESubscriberApi api = null;

// This sample program waits for pull requests from the SCE
// and answers to them with pull response
// The program exists after all 500 were logged in
public static void main (String args[]) throws Exception
{
int numSubscribersToLogin = 500;
//instantiate an API with reconnect interval of 5 seconds
api = new PRPC_SCESubscriberApi("myAPI","1.1.1.1",5000);
// construct an operation result handler (from the
// previous example
MyOperationResultHandler handler =
new MyOperationResultHandler();

// instantiate logout and login-pull listener
MyListener listener = new MyListener(api);
try
{
// register to logout indications
api.registerLogoutListener(listener);
api.registerLoginPullListener(listener);
// connect to the SCE
api.connect();

// wait for login-pull requests from the SCE
// they will be issued if you have traffic for unknown
// subscribers at the SCE
System.out.println("Waiting for pull requests for "+
numSubscribersToLogin+
" subscribers");
// wait for all subscribers to be logged in
listener.waitForPullResult(numSubscribersToLogin);
//logout all subscribers
System.out.println("logout of "+numSubscribersToLogin+
" subscribers");
for (int i=0; i<numSubscribersToLogin; i++)
{
api.logout("sub"+i,null,handler);
}
// wait for all subscribers to be logged out
listener.waitForLogoutResult(numSubscribersToLogin);
}
finally
{
```

```
api.unregisterLoginPullListener();
api.unregisterLogoutListener();
api.disconnect();
}
}
}
```

CHAPTER 6

# Troubleshooting

This module describes the usage of the API logging abilities for troubleshooting the integration with the API. API logging enables the user to monitor the operations being called including the received parameters both at the API client and at the SCE side.



## SCE Logging

The SCE platform provides the ability to log all of the operations called by the Policy Server into the SCE user-log file.



### Default Log Messages

The SCE issues the following messages by default without any further configuration:

- For connect operation:

  ```
  <client-name>- connect operation was called, registered listeners: <type of the listeners that were registered>
  ```

- For disconnect operation:

  ```
  <client-name>- disconnected
  ```

- For registerLoginPullListener operation:

  ```
  <client-name>- registered a Login Pull Listener
  ```

- For unregisterPullListener operation:

  ```
  <client-name>- unregistered a Pull Listener
  ```

- For registerLogoutListener operation:

  ```
  <client-name>- registered a Logout Listener
  ```

- For unregisterLogoutListener operation:

  ```
  <client-name>- unregistered a Logout Listener
  ```

- For registerQuotaListener operation:

  ```
  <client-name>- registered Quota Listener
  ```

- For unregisterQuotaListener operation:

```
<client-name>- unregister Quota Listener
```

- For synchronizePushStart operation:

```
<client-name>- synchronize Push Start
```

- For synchronizePushEnd operation:

```
<client-name>- synchronize Push End
```

- For synchronizePullStart operation:

```
<client-name>- synchronize Pull Start
```

- For synchronizePullEnd operation:

```
<client-name>- synchronize Pull End
```

# Subscriber Operations Log Messages

A special flag activates subscriber operation log messages. To receive these messages, enable the flag.

To enable logging, use the following CLI at the SCE platform:

**(config)# management-agent sce-api logging**

To view the USERLOG file, use the following CLI at the SCE platform:

#>**logger get user-log** *FILE NAME*

**Note** Enabling logging causes performance degradation. Therefore, it is advisable to use logging only for troubleshooting purposes.

To disable logging, use the following CLI at the SCE platform:

**(config)#>no management-agent sce-api logging**

To view whether the logging is enabled, use the following CLI at the SCE platform:

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

When the logging flag is enabled, the message below is issued for the following operations:

- login operation
- networkIDUpdate operation
- logout operation
- quotaUpdate operation
- loginPullResponse operation
- profileUpdate operation
- getQuotaStatus operation

```
<operation name>operation was called
with parameters:
subscriberID - <subscriber ID>
anonymousSubscriberID - <anonymousSubscriberID >
mappings - <mappings list>
mappings types - <mapping types list>
policy - <policy properties list>
quota - <quota operation/quota buckets list>
```

For the following bulk operations:

- loginBulk operation
- networkIDUpdateBulk operation
- logoutBulk operation

- quotaUpdateBulk operation
- loginPullBulkResponse operation
- profileUpdateBulk operation
- getQuotaStatuBulkRequest operation
- getSubscribersBulk

The following message is issued:

```
<operation name>operation was called with parameters:
bulk size - <bulk size>
```

The following messages are issued for the LoginPullListener:

- For loginPullRequest:

```
loginPullRequest operation was called with parameters:
anonymousSubscriberID - <anonymous subscriber ID>
mappings - <mappings list>
mapping types - <mapping types>
```

- For loginPullRequestBulk:

```
loginPullRequestBulk operation was called with parameters:
bulk size - <bulk size>
```

- getSubscribersBulkResponse

```
getSubscribersBulkResponse operation was called with parameters:
bulk size - <bulk size>
```

The following messages are issued for the LogoutListener:

- For logoutIndication:

```
logoutIndication operation was called with parameters:
subscriberID - <anonymous subscriber ID>
```

- For logoutBulkIndication:

```
logoutBulkIndication operation was called with parameters:
bulk size - <bulk size>
```

The following messages are issued for the QuotaListenerEx:

- For quotaStatusIndication:

```
quotaStatusIndication operation was called with parameters:
subscriberID - <Subscriber ID>
quota - <subscriber quota>
```

- For quotaBelowThresholdIndication:

```
quotaBelowThresholdIndication operation was called with parameters:
subscriberID - <Subscriber ID>
quota - <subscriber quota>
```

- For quotaDepletedIndication:

```
quotaDepletedIndication operation was called with parameters:
subscriberID - <Subscriber ID>
quota - <subscriber quota>
```

- For quotaStateRestore:

```
quotaStateRestore operation was called with parameters:
subscriberID - <Subscriber ID>
quota - <subscriber quota>
```

- For quotaStatusBulkIndication:

```
quotaStatusBulkIndication operation was called with parameters:
subs - <bulk size>
```

- For quotaBelowThresholdBulkIndication:

```
quotaBelowThresholdBulkIndication operation was called with parameters:
subs - <bulk size>
```

- For quotaDepletedBulkIndication:

```
quotaDepletedBulkIndication operation was called with parameters:
subs - <bulk size>
```

- For quotaStateBulkRestore:

```
quotaStateBulkRestore operation was called with parameters:
subs - <bulk size>
```

# API Client Logging

The API provides the ability to log every activated operation into the apilog file located under **${user.home}** directory. The logging parameters are configured using the Log4J properties files. To enable the logging make sure this file is in the application's CLASSPATH. This file is read at startup of the application so after changing it you must restart the application.

The following is the content of the **log4.properies** file:

```
# default Log4j configuration for SCE Subscriber API
log4j.rootCategory=INFO, apiStdout
# In order to enable the logging to the file Replace the above
# line with the following:
# log4j.rootCategory=INFO, files
# stdout is set to be a ConsoleAppender.
log4j.appender.apiStdout=org.apache.log4j.ConsoleAppender
log4j.appender.apiStdout.layout=org.apache.log4j.PatternLayout
log4j.appender.apiStdout.layout.ConversionPattern=+ %d{dd-MMM HH:mm:ss.SSS} [%t] %-5p
%c%n%m%n
# files is set to be a RollingFileAppender.
#log4j.appender.files=org.apache.log4j.RollingFileAppender
#log4j.appender.files.layout=org.apache.log4j.PatternLayout
#log4j.appender.files.layout.ConversionPattern=+ %d{dd-MMM yyyy HH:mm:ss.SSS} [%t] %-5p %c
%x\n%m\n
#log4j.appender.files.File=${user.home}/apilog
#log4j.appender.files.Threshold=INFO
#log4j.appender.files.ImmediateFlush=true
#log4j.appender.files.MaxFileSize=1MB
#log4j.appender.files.MaxBackupIndex=4
# In order to enable debug logging uncomment the following line
#log4j.category.com.scms.api.sce.prpc=DEBUG
```

To enable the debug logging, uncomment the last line in the file. By default, the logging is performed to the standard output. To direct the logging to the file, uncomment the # log4j.rootCategory=INFO, files line as explained in the file.

# API Client Log Messages

The API client issues the following messages after properly configuring the **log4j.properties** file:

- For API constructor:

- ```
  PRPC_SCESubscriberApi constructor was called with the following parameters:
  apiName - <apiName>
  host - <sceHost>
  port - <scePort>
  auto-reconnect - <autoReconnectInterval>
  ```

- For init operation:

```
init operation was called with parameters <properties>
```
```

- For setConnectionListener:

```
setConnectionListener operation was called
```

- For setRedundancyStateListener:

```
setRedundancyStateListener operation was called
```

- For isConnected:

```
isConnected operation was called
```

- For getAPIVersion:

```
getAPIVersion operation was called
```

For the following operations:

- login operation
- networkIDUpdate operation
- logout operation
- quotaUpdate operation
- loginPullResponse operation
- profileUpdate operation
- getQuotaStatus operation

The following message is issued:

```
<operation name>operation was called with parameters:
subscriberID - <subscriber ID>
anonymousSubscriberID - <anonymousSubscriberID>
mappings - <mappings list>
mappings types - <mapping types list>
policy - <policy properties list>
quota - <quota operation/quota buckets list>
```

For the following bulk operations:

- loginBulk operation
- networkIDUpdateBulk operation
- logoutBulk operation
- quotaUpdateBulk operation
- loginPullBulkResponse operation
- profileUpdateBulk operation
- getQuotaStatuBulkRequest operation
- getSubscribersBulk operation

The following message is issued:

```
operation name>operation was called with parameters:
bulk size - <bulk size>
```

- For connect operation:

```
connect operation was called, registered listeners:
<type of the listeners that were registered>
```

- For disconnect operation:

```
disconnect operation was called
```

- For registerLoginPullListener operation:

```
registerLoginPullListener operation was called
```

- For unregisterPullListener operation:

```
unregisterPullListener operation was called
```

- For registerLogoutListener operation:

```
registerLogoutListener operation was called
```

- For unregisterLogoutListener operation:

```
unregisterLogoutListener operation was called
```

- For registerQuotaListener operation:

```
registerQuotaListener operation was called
```

- For unregisterQuotaListener operation:

```
unregisterQuotaListener operation was called
```

- For synchronizePushStart operation:

```
synchronizePushStart operation was called
```

- For synchronizePushEnd operation:

```
synchronizePushEnd operation was called
```

- For synchronizePullStart operation:

```
synchronizePullStart operation was called
```

- For synchronizePullEnd operation:

```
synchronizePullEnd operation was called
```

The following messages are issued for the LoginPullListener listener callback methods:

- For loginPullRequest:

```
loginPullRequest operation was called with parameters:
anonymousSubscriberID - <anonymous subscriber ID>
mappings - <mappings list>
mapping types - <mapping types>
```

- For loginPullRequestBulk:

```
loginPullRequestBulk operation was called with parameters:
bulk size - <bulk size>
```

- For getSubscribersBulkResponse:

```
getSubscribersBulkResponse operation was called with parameters:
bulk size - <bulk size>
```

The foillowing messages are issued for the LogoutListener listener callback methods:

- For logoutIndication:

```
logoutIndication operation was called with parameters:
subscriberID - <anonymous subscriber ID>
```

- For logoutBulkIndication:

```
logoutBulkIndication operation was called with parameters:
bulk size - <bulk size>
```

The following messages are issued for the QuotaListenerEx listener callback methods:

- For quotaStatusIndication:

```
quotaStatusIndication operation was called with parameters:
subscriberID - <Subscriber ID>
quota - <subscriber quota>
```

- For quotaBelowThresholdIndication:

```
quotaBelowThresholdIndication operation was called with parameters:
subscriberID - <Subscriber ID>
quota - <subscriber quota>
```

- For quotaDepletedIndication:

```
quotaDepletedIndication operation was called with parameters:
subscriberID - <Subscriber ID>
quota - <subscriber quota>
```

- For quotaStateRestore:

```
quotaStateRestore operation was called with parameters:
subscriberID - <Subscriber ID>
quota - <subscriber quota>
```

- For quotaStatusBulkIndication:

```
quotaStatusBulkIndication operation was called with parameters:
subs - <bulk size>
```

- For quotaBelowThresholdBulkIndication:

```
quotaBelowThresholdBulkIndication operation was called with parameters:
subs - <bulk size>
```

- For quotaDepletedBulkIndication:

```
quotaDepletedBulkIndication operation was called with parameters:
subs - <bulk size>
```

- For quotaStateBulkRestore:

```
quotaStateBulkRestore operation was called with parameters:
subs - <bulk size>
```

APPENDIX A

# List of Error Codes

This module lists the error codes that are returned by the API.

## List of Error Codes

Error codes are used for interpreting the actual error for which an **OperationException** was returned. The error code is extracted using the **getErrorCode** method.

A list of the error codes and their description are given in the following table.

***Table A-1 List of Error Codes***

| Error Code | Description |
|---|---|
| ERROR_CODE_NO_APPLICATION_INSTALLED | Application required for the operation execution is not installed. |
| ERROR_CODE_INVALID_PARAMETER | One of the arguments provided to the method is illegal. |
| ERROR_CODE_SUBSCRIBER_ALREADY_EXISTS | The subscriber on which the operation was performed already exists in the SCE. |
| ERROR_CODE_SUBSCRIBER_DOES_NOT_EXIST | The subscriber on which the operation is performed does not exist in the SCE. |
| ERROR_CODE_FATAL_EXCEPTION | Too many errors occurred at the SCE when trying to perform the operation. |
| ERROR_CODE_RESOURCE_SHORTAGE | Internal error. |
| ERROR_CODE_OPERATION_ABORTED | Internal error. |
| ERROR_CODE_ARRAY_ACCESS | Internal error. |
| ERROR_CODE_ATTRIBUTE_NOT_FOUND | Internal error. |
| ERROR_CODE_CLASS_CAST | Internal error. |
| ERROR_CODE_CLASS_NOT_FOUND | Internal error. |
| ERROR_CODE_CLIENT_INTERNAL_ERROR | Internal error. |
| ERROR_CODE_CLIENT_OUT_OF_THREADS | Internal error. |
| ERROR_CODE_ILLEGAL_STATE | Internal error. |
| ERROR_CODE_OBJECT_NOT_FOUND | Internal error. |
| ERROR_CODE_OPERATION_NOT_FOUND | Internal error. |

*Table A-1 List of Error Codes (continued)*

| Error Code | Description |
|---|---|
| ERROR_CODE_OUT_OF_MEMORY | Internal error. |
| ERROR_CODE_RUNTIME | Internal error. |
| ERROR_CODE_NULL_POINTER | Internal error. |
| ERROR_CODE_UNKNOWN | Internal error. |