Cisco SCE Subscriber API Programmer Guide

Release 4.2.x
August 30, 2014

Cisco Systems, Inc.
www.cisco.com

Cisco has more than 200 offices worldwide. Addresses, phone numbers, and fax numbers are listed on the Cisco website at www.cisco.com/go/offices.
About this Guide

Introduction

Document Revision History

Organization

Related Documentation

Conventions

Obtaining Documentation and Submitting a Service Request

CHAPTER 1

Getting Started

Introduction

Restrictions for the Cisco SCE Subscriber API

Cisco SCE Subscriber API

Platforms

Package Content

Extracting and Installing the Package

Installing the Distribution on a UNIX Platform

Installing the Distribution on a Windows Platform

The Cisco SCE Platform Setup

Prerequisites

Configuring Cisco SCE in Pull Mode

Configuring the RDR Formatter

Configuring the RDR Formatter to Issue Quota-Related Indications

Mapping the Quota RDR Tags to a Different Category

Configuring the RDR Server

Verifying the RDR Server Configuration

Enabling the RDR Server

Changing the RDR Server Port

Configuring Cisco SCE for Quota Management Provisioning

Configuring the API Disconnection Timeout

Resetting the Disconnection Timeout to the Default Value

Viewing the Timeout Value
CHAPTER 2

Concepts and Terms 2-1

Introduction 2-1
Subscriber Characteristics 2-2
  Subscriber ID 2-2
  Anonymous Subscriber ID 2-2
Network ID 2-2
Policy Profile 2-2
Quota 2-2
Extended Attributes 2-3
Subscriber Integration Modes 2-3
  Push Mode 2-3
  Pull Mode 2-3
Nonblocking Mode 2-4
Indications Listeners 2-5
Supported Topologies 2-6
Multithreading Support 2-9
Autoreconnect Support 2-9
Reliability Support 2-9
High-Availability Support 2-9
Synchronization 2-9
Practical Tips 2-10

CHAPTER 3

Application Programming Interface Events 3-1

Introduction 3-1
API Events 3-2
  Network ID Management Events 3-2
    Login Events 3-3
    Logout Events 3-4
    Network ID Update Event 3-5
  Policy Profile Management 3-5
    Profile Update 3-5
  Quota Management 3-5
    Quota Update 3-6
    Get Quota Status 3-6
    Quota Status 3-6
    Quota Below Threshold 3-7
    Quota Depleted 3-7
    Quota State Restore 3-7
CHAPTER 4

Application Programming Interface Data Types 4-1

Introduction 4-1
Subscriber ID 4-1
Network ID Mappings 4-2
  Specifying IP Address Mapping 4-3
  Specifying IP Range Mapping 4-3
  Specifying Private IP Address or Private IP Range over VLAN Mapping 4-3
  Specifying VLAN Tag Mapping 4-4
  Specifying IPv6 Address Mapping 4-4
Network ID Mapping Examples 4-4
Cisco SCA BB Subscriber Policy Profile 4-5
  PolicyProfile Class 4-5
Subscriber Quota 4-6
  SCAS_BB_Quota 4-7
  SCAS_BB_QuotaOperation 4-8
Extended Attributes 4-9
Bulk Operations Data Types 4-10
  Bulk Iterator 4-10
    SubscriberData 4-10
  Login_BULK Class 4-10
    Constructor 4-11
    addBulkEntry Method 4-11
    Examples 4-12
  SubscriberID_BULK Class 4-13
    Constructors 4-13
    addBulkEntry Method 4-13
  NetworkAndSubscriberID_BULK Class 4-14
    Constructors 4-14
    addBulkEntry Method 4-14
  LoginPullResponse_BULK Class 4-15
    Constructors 4-15
    addBulkEntry Method 4-16
  PolicyProfile_BULK Class 4-16
    Constructors 4-17
CHAPTER 5
Programming with the Cisco SCE Subscriber API

Introduction  5-1
API Classes   5-2
  Package com.scms.api.sce.prpc  5-2
  Package com.scms.api.sce  5-2
    Indication Listeners  5-2
    Connection Monitoring  5-2
    Cisco SCE Cascade Topology Support  5-2
    Operation Results Handling  5-2
  Package com.scms.common  5-3
Programming Guidelines  5-4
  Programming with Callback Methods  5-4
PRPC_SCESubscriberApi Class  5-5
  API Construction  5-5
    Listener Setup Operations  5-6
    Advanced Setup Operations  5-6
    Connecting to Cisco SCE  5-7
    getApiVersion  5-8
    API Finalization  5-8
Virtual Link API  5-8
  Virtual Link Provisioning Operations  5-9
    Create a Virtual Link  5-9
    Create Multiple Virtual Links  5-10
    Delete a Virtual Link  5-11
<table>
<thead>
<tr>
<th>Operation</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete Multiple Virtual Links</td>
<td>5-11</td>
</tr>
<tr>
<td>Update a Virtual Link</td>
<td>5-12</td>
</tr>
<tr>
<td>Update Multiple Virtual Links</td>
<td>5-13</td>
</tr>
<tr>
<td>Check Virtual Link Status</td>
<td>5-13</td>
</tr>
<tr>
<td>Get Maximum Virtual Links</td>
<td>5-14</td>
</tr>
<tr>
<td>Enable the Virtual Link Mode</td>
<td>5-15</td>
</tr>
<tr>
<td>Indication Listeners</td>
<td>5-15</td>
</tr>
<tr>
<td>LoginPullListener Interface Class</td>
<td>5-15</td>
</tr>
<tr>
<td>loginPullRequest Callback Method</td>
<td>5-16</td>
</tr>
<tr>
<td>loginPullRequestBulk Callback Method</td>
<td>5-17</td>
</tr>
<tr>
<td>GetSubscribersBulkResponse Callback Method</td>
<td>5-17</td>
</tr>
<tr>
<td>DualLoginPullListener Interface Class</td>
<td>5-17</td>
</tr>
<tr>
<td>loginV6PullRequest Callback Method</td>
<td>5-18</td>
</tr>
<tr>
<td>LogoutListener Interface Class</td>
<td>5-19</td>
</tr>
<tr>
<td>logoutIndication Callback Method</td>
<td>5-19</td>
</tr>
<tr>
<td>logoutBulkIndication Callback Method</td>
<td>5-19</td>
</tr>
<tr>
<td>QuotaListenerEx Interface Class</td>
<td>5-19</td>
</tr>
<tr>
<td>quotaStatusIndication Callback Method</td>
<td>5-20</td>
</tr>
<tr>
<td>quotaStatusBulkIndication Callback Method</td>
<td>5-20</td>
</tr>
<tr>
<td>quotaBelowThresholdIndication Callback Method</td>
<td>5-21</td>
</tr>
<tr>
<td>quotaBelowThresholdBulkIndication Callback Method</td>
<td>5-21</td>
</tr>
<tr>
<td>quotaDepletedIndication Callback Method</td>
<td>5-21</td>
</tr>
<tr>
<td>quotaDepletedBulkIndication Callback Method</td>
<td>5-21</td>
</tr>
<tr>
<td>quotaStateRestore Callback Method</td>
<td>5-22</td>
</tr>
<tr>
<td>quotaStateBulkRestore Callback Method</td>
<td>5-22</td>
</tr>
<tr>
<td>Connection Monitoring</td>
<td>5-23</td>
</tr>
<tr>
<td>ConnectionListener Interface</td>
<td>5-23</td>
</tr>
<tr>
<td>Disconnect Listener: Example</td>
<td>5-23</td>
</tr>
<tr>
<td>Cisco SCE Cascade Topology Support</td>
<td>5-24</td>
</tr>
<tr>
<td>isRedundancyStatusActive Method</td>
<td>5-24</td>
</tr>
<tr>
<td>RedundancyStateListener Interface</td>
<td>5-24</td>
</tr>
<tr>
<td>Configuring Cisco SCE to Ignore Cascade Violation Errors</td>
<td>5-25</td>
</tr>
<tr>
<td>Configuring Cisco SCE to Define the Quota Buffer Size and Quota Rate</td>
<td>5-26</td>
</tr>
<tr>
<td>Result Handling</td>
<td>5-27</td>
</tr>
<tr>
<td>OperationResultHandler Interface</td>
<td>5-27</td>
</tr>
<tr>
<td>OperationArguments Class</td>
<td>5-28</td>
</tr>
<tr>
<td>OperationException Class</td>
<td>5-29</td>
</tr>
<tr>
<td>Subscriber Provisioning Operations</td>
<td>5-30</td>
</tr>
<tr>
<td>Login Operation</td>
<td>5-30</td>
</tr>
</tbody>
</table>
loginBulk Operation 5-32
  Syntax 5-33
  Description 5-33
  Parameters 5-33
  Error Codes 5-33
loginPullResponse Operation 5-33
  Syntax 5-33
  Description 5-34
  Parameters 5-34
  Error Codes 5-34
loginPullResponseBulk Operation 5-35
  Syntax 5-35
  Description 5-35
  Parameters 5-35
  Error Codes 5-35
Logout Operation 5-35
  Syntax 5-36
  Description 5-36
  Parameters 5-36
  Error Codes 5-36
logoutBulk Operation 5-36
  Syntax 5-36
  Description 5-37
  Parameters 5-37
  Error Codes 5-37
networkIdUpdate Operation 5-37
  Syntax 5-37
  Description 5-37
  Parameters 5-38
  Error Codes 5-38
networkIdUpdateBulk Operation 5-39
  Syntax 5-39
  Description 5-39
  Parameters 5-39
  Error Codes 5-39
profileUpdate Operation 5-39
   Syntax 5-40
   Description 5-40
   Parameters 5-40
   Error Codes 5-40
profileUpdateBulk Operation 5-40
   Syntax 5-41
   Description 5-41
   Parameters 5-41
   Error Codes 5-41
quotaUpdate Operation 5-41
   Syntax 5-41
   Description 5-42
   Parameters 5-42
   Error Codes 5-42
quotaUpdateBulk Operation 5-42
   Syntax 5-42
   Description 5-42
   Parameters 5-43
   Error Codes 5-43
getQuotaStatus Operation 5-43
   Syntax 5-43
   Description 5-43
   Parameters 5-43
   Error Codes 5-44
getQuotaStatusBulk Operation 5-44
   Syntax 5-44
   Description 5-44
   Parameters 5-44
   Error Codes 5-45
Cisco SCE-API Synchronization 5-45
   Push Mode Synchronization Procedure 5-46
      synchronizePushStart 5-47
      synchronizePushEnd 5-47
   Pull Mode Synchronization Procedure 5-47
      synchronizePullStart 5-49
      synchronizePullEnd 5-49
      getSubscribersBulk 5-49
      getSubscribersBulkIPv6 5-50
Advanced API Programming 5-52
CHAPTER 6  Troubleshooting  6-1

Introduction  6-1
Cisco SCE Logging  6-2
  Default Log Messages  6-2
  Subscriber Operation Log Messages  6-3
API Client Logging  6-6
  API Client Log Messages  6-6

Introduction  A-1
List of Error Codes  A-2
About this Guide

Published: August 30, 2014.

Introduction

This guide is for networking or computer technicians who want to integrate a policy server with a Cisco Service Control Engine (SCE) platform to provision subscribers.

Document Revision History

The following Document Revision History table records the changes made to this document.

<table>
<thead>
<tr>
<th>Cisco Service Control Release and Date</th>
<th>Change Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release 4.2.x August 30, 2014</td>
<td>First version of this document (new for the Release 4.2.x train). Updated for Release 4.2.0.</td>
</tr>
</tbody>
</table>
Organization

The guide contains the following sections.

<table>
<thead>
<tr>
<th>Table 2</th>
<th>Document Organization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>Chapter 1</td>
<td>Getting Started</td>
</tr>
<tr>
<td>Chapter 2</td>
<td>Concepts and Terms</td>
</tr>
<tr>
<td>Chapter 3</td>
<td>Application Programming Interface Events</td>
</tr>
<tr>
<td>Chapter 4</td>
<td>Application Programming Interface Data Types</td>
</tr>
<tr>
<td>Chapter 5</td>
<td>Programming with the Cisco SCE Subscriber API</td>
</tr>
<tr>
<td>Chapter 6</td>
<td>Troubleshooting</td>
</tr>
<tr>
<td>Appendix A</td>
<td>List of Error Codes</td>
</tr>
</tbody>
</table>

1. API = application programming interface

Related Documentation

Use this API guide with the following Cisco documentation:

- *Cisco Service Control Management Suite Subscriber Manager User Guide*
- *Cisco Service Control Application for Broadband User Guide*
Conventions

The document uses the following conventions.

Table 3 Conventions

<table>
<thead>
<tr>
<th>Convention</th>
<th>Indication</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bold</strong> font</td>
<td>Commands and keywords and user-entered text appear in <strong>bold</strong> font.</td>
</tr>
<tr>
<td><strong>italic</strong> font</td>
<td>Document titles, new or emphasized terms, and arguments for which you supply values are in <strong>italic</strong> font.</td>
</tr>
<tr>
<td>[   ]</td>
<td>Elements in square brackets are optional.</td>
</tr>
<tr>
<td>{x</td>
<td>y</td>
</tr>
<tr>
<td>[ x</td>
<td>y</td>
</tr>
<tr>
<td>string</td>
<td>A nonquoted set of characters. Do not use quotation marks around the string or the string will include the quotation marks.</td>
</tr>
<tr>
<td><strong>courier</strong> font</td>
<td>Terminal sessions and information the system displays appear in <strong>courier</strong> font.</td>
</tr>
<tr>
<td>&lt; &gt;</td>
<td>Nonprinting characters such as passwords are in angle brackets.</td>
</tr>
<tr>
<td>[   ]</td>
<td>Default responses to system prompts are in square brackets.</td>
</tr>
<tr>
<td>!, #</td>
<td>An exclamation point (!) or a pound sign (#) at the beginning of a line of code indicates a comment line.</td>
</tr>
</tbody>
</table>

**Note**

Means reader take note.

**Tip**

Means the following information will help you solve a problem.

**Caution**

Means reader be careful. In this situation, you might perform an action that could result in equipment damage or loss of data.

**Timesaver**

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

**Warning**

Means reader be warned. In this situation, you might perform an action that could result in bodily injury.
Obtaining Documentation and Submitting a Service Request

For information on obtaining documentation, using the Cisco Bug Search Tool (BST), submitting a service request, and gathering additional information, see What’s New in Cisco Product Documentation at: http://www.cisco.com/c/en/us/td/docs/general/whatsnew/whatsnew.html.

Subscribe to What’s New in Cisco Product Documentation, which lists all new and revised Cisco technical documentation, as an RSS feed and deliver content directly to your desktop using a reader application. The RSS feeds are a free service.
Getting Started

Introduction

This chapter identifies the platforms on which you can run the Cisco Service Control Engine (SCE) Subscriber application programming interface (API). The chapter also describes how to install, compile, and start running the API.

The Cisco SCE Subscriber API enables you to connect policy servers (external applications) to a Cisco SCE so that you can provision subscribers.

Subscriber provisioning entails updating the Network IDs, Policy Profile, and Quota and correlating these characteristics to a Subscriber ID. For more information about the characteristics of a subscriber in the Cisco Service Control Application for Broadband (Cisco SCA BB), see the “Subscriber Characteristics” section on page 2-2.

You can install and run the API on several policy servers concurrently. Each policy server can perform different parts of the subscriber provisioning process as shown in Figure 1-1.

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

<table>
<thead>
<tr>
<th>AAA Server</th>
<th>SCE Subscriber API</th>
<th>Subscriber ID: Network ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Policy Profile Server</td>
<td>SCE Subscriber API</td>
<td>Subscriber ID: Policy Profile</td>
</tr>
<tr>
<td>Quota Provisioning Server</td>
<td>SCE Subscriber API</td>
<td>Subscriber ID: Quota</td>
</tr>
</tbody>
</table>

The API uses the Proprietary Remote Procedure Call (PRPC) protocol to transport the connection to the Cisco SCE. The PRPC is a proprietary remote procedure call protocol designed by Cisco.

Note

The API provides a connection to one Cisco SCE platform for each API instance.
Restrictions for the Cisco SCE Subscriber API

Version 3.0.5 of the API is backward compatible with previous versions, but is not binary compatible. Recompile applications that use a previous version of the API to use the new version. Because the API is backward compatible, you do not need to change the application source code.

Note

If you upgrade Cisco SCE to Release 3.0.5, upgrade the API to Version 3.0.5 and recompile the application that uses it.

Cisco SCE Subscriber API

This section consists of the following subsections:

- Platforms, page 1-2
- Package Content, page 1-2

Platforms

The Cisco SCE Subscriber API operates on any platform that supports Java Version 5.0.

Package Content

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

The <installdir>/javadoc folder contains the Cisco 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 to operate the API.

Table 1-1 provides the layout of the installation directory.

<table>
<thead>
<tr>
<th>Path</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;installdir&gt;</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>README</td>
<td>API readme file</td>
</tr>
<tr>
<td>&lt;installdir&gt;/javadoc</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>index.html</td>
<td>Index of all API specifications</td>
</tr>
<tr>
<td>—</td>
<td>(API specification files, and so on.)</td>
<td>API specification documents</td>
</tr>
<tr>
<td>&lt;installdir&gt;/lib</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>sceapi.jar</td>
<td>Cisco SCE Subscriber API executable</td>
</tr>
<tr>
<td>—</td>
<td>asn1rt.jar</td>
<td>Utility jar used by the API</td>
</tr>
</tbody>
</table>
### Table 1-1   Layout of the Installation Directory (continued)

<table>
<thead>
<tr>
<th>Path</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>log4j.jar</td>
<td>Utility jar used by the API</td>
</tr>
<tr>
<td></td>
<td>log4j.properties</td>
<td>Property file required logging</td>
</tr>
<tr>
<td></td>
<td>jdmkrt.jar</td>
<td>Utility jar used by the API</td>
</tr>
</tbody>
</table>
Extracting and Installing the Package

The Cisco SCE Subscriber API distribution is part of the Cisco Service Control SM login event generator (LEG) distribution file and is located in the sce_api directory.

The Cisco SCE Subscriber API is packaged in a UNIX tar file. Use the UNIX tar utility to extract the Cisco SCE Subscriber API. The API is compiled with support from Log4J Version 1.2.

This section consists of the following subsections:

- Installing the Distribution on a UNIX Platform, page 1-4
- Installing the Distribution on a Windows Platform, page 1-4

Installing the Distribution on a UNIX Platform

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

Step 2  Unzip the distribution file by using the following command:

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

Step 3  Extract the Cisco SCE Subscriber API package tar file by using the following command:

```bash
#> tar -xvf sce-java-api-dist.tar
```

Installing the Distribution on a Windows Platform

To install the distribution, use a zip extractor (such as WinZip) to unzip the package.
The Cisco SCE Platform Setup

The following sections describe how to configure the Cisco SCE platform so that the API operates correctly:

- Prerequisites, page 1-5
- Configuring Cisco SCE in Pull Mode, page 1-5
- Configuring the RDR Formatter, page 1-6
- Configuring the RDR Server, page 1-6
- Configuring Cisco SCE for Quota Management Provisioning, page 1-7
- Configuring the API Disconnection Timeout, page 1-7

Prerequisites

The API connects to the PRPC server on the Cisco SCE platform. The PRPC server runs a PRPC protocol designed by Cisco.

Before using the API, ensure that the following conditions are met:

- The Cisco SCE is active, running, and reachable from the machine that hosts the API.
- The PRPC server on the Cisco SCE is started.

Configuring Cisco SCE in Pull Mode

To enable the Cisco SCE platform, running in Pull Mode, to issue a request for subscriber information, complete the following configuration steps using the Cisco SCE platform CLI. See the “Pull Mode” section on page 2-3.

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

Step 1

Configure the subscriber templates. For example,

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

For more information about the templates and the format of the comma-separated value (CSV) file, see the *Cisco Service Control Application for Broadband User Guide*.

Step 2

Configure the ranges for unmapped subscriber groups.

- Enter the `subscriber anonymous group import` CLI command to import anonymous groups from a file.

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

- Alternatively, enter `subscriber anonymous group name` the CLI command to define the anonymous group.

  ```
  (config if)#> subscriber anonymous group name NAME IP-range IP RANGE
  ```
Configuring the RDR Formatter

This section consists of the following subsections:

- Configuring the RDR Formatter to Issue Quota-Related Indications, page 1-6
- Mapping the Quota RDR Tags to a Different Category, page 1-6

Configuring the RDR Formatter to Issue Quota-Related Indications

To enable the Raw Data Record (RDR) formatter to issue quota-related indications, configure the RDR formatter on the Cisco SCE platform. Enter the RDR-formatter destination CLI command as follows:

```
#> 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, enter the RDR-formatter rdr-mapping CLI command as follows:

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

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

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

Configuring the RDR Server

To enable the API to receive quota indications, enable the RDR server to listen on the same port that is configured in the RDR formatter.

This section consists of the following subsections:

- Verifying the RDR Server Configuration, page 1-6
- Enabling the RDR Server, page 1-7
- Changing the RDR Server Port, page 1-7

Verifying the RDR Server Configuration

To verify the RDR server configuration, enter the show RDR-server CLI command as follows:

```
#> show RDR-server

RDR server is ONLINE
RDR server port is 33001
```
Enabling the RDR Server

To enable the RDR server, enter the `RDR-server` CLI command as follows:

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

Default RDR server port is 33001

Changing the RDR Server Port

To change the RDR server port, enter the `RDR-server port` CLI command as follows:

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

Configuring Cisco SCE for Quota Management Provisioning

To configure the Cisco SCE to provision the quota management, complete the following steps:

**Step 1**
Service the RDR formatter.

```
SCE8000(config)#> service RDR-formatter
```

**Step 2**
Enable, through PQB file or CLU, the Quota RDRs in service configuration file to get quota indications.

```
SCE8000(config if)#> tunable GT_REP_QP_RemainQuota_Enable value true
SCE8000(config if)#> tunable GT_REP_QP_QuotaBreach_Enable value true
SCE8000(config if)#> tunable GT_REP_QP_ThresholdBreach_Enable value true
SCE8000(config if)#> tunable GT_REP_QP_RemainQuota_MaxFrequency value 1000
SCE8000(config if)#> tunable GT_REP_QP_RemainQuota_Rate value 60
```

**Step 3**
Set the quota management mode to External in Cisco SCA BB console.

Configuring the API Disconnection Timeout

You can set the timeout duration for the API to wait before it reconnects to the Cisco SCE platform following a disconnection. During this timeout, the Cisco SCE does not free the resources and no data is lost. After the timeout elapses and the API does not reconnect, the Cisco SCE considers the API disconnected and frees all resources. The default timeout value is 5 minutes.

To configure the API disconnection timeout, enter the `management-agent sce-api timeout` CLI command as follows:

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

This section consists of the following subsections:

- Resetting the Disconnection Timeout to the Default Value, page 1-8
- Viewing the Timeout Value, page 1-8
Resetting the Disconnection Timeout to the Default Value

To reset the API disconnection timeout to the default value, enter the
**default management-agent sce-api timeout** CLI command as follows:
```
(config)# default management-agent sce-api timeout
```

Viewing the Timeout Value

To view the timeout value, enter the **show management-agent sce-api** CLI command as follows:
```
#> show management-agent sce-api
```
**Concepts and Terms**

**Introduction**

This chapter describes the various terms and concepts related to working with the Cisco Service Control Engine (SCE) Subscriber application programming interface (API).

It consists of the following sections:

- Subscriber Characteristics, page 2-2
- Subscriber Integration Modes, page 2-3
- Nonblocking Mode, page 2-4
- Indications Listeners, page 2-5
- Supported Topologies, page 2-6
- Multithreading Support, page 2-9
- Autoreconnect Support, page 2-9
- Reliability Support, page 2-9
- High-Availability Support, page 2-9
- Synchronization, page 2-9
- Practical Tips, page 2-10
Subscriber Characteristics

A subscriber is one of the fundamental entities in the Service Control Application for Broadband (SCA BB) solution.

The configuration of services in the SCA BB solution is based on enforcing, accounting, and monitoring the individual subscriber.

For more information about formatting and configuring subscriber characteristics, see Chapter 4, “Application Programming Interface Data Types.” The following sections describe the characteristics of the subscriber in the SCA BB:

- Subscriber ID, page 2-2
- Anonymous Subscriber ID, page 2-2
- Network ID, page 2-2
- Policy Profile, page 2-2
- Quota, page 2-2
- Extended Attributes, page 2-3

Subscriber ID

A Subscriber ID is a unique identifier, such as a user name, International Mobile Subscriber Identity (IMSI), or other code that uniquely identifies a subscriber.

Anonymous Subscriber ID

For Pull Mode integration, the Cisco SCE assigns each unknown subscriber IP address a temporary, anonymous subscriber ID, until it receives the real subscriber ID from the policy server.

For more information on Pull Mode integration, see the “Subscriber Integration Modes” section on page 2-3.

Network ID

The Cisco SCE correlates a certain traffic flow to a subscriber by mapping a network identifier such as an IP address, IP range, IPv6 address, or VLAN, to the subscriber entity.

Policy Profile

A Policy Profile includes a set of parameters that the Cisco SCA BB solution uses to define the policy that is enforced on the subscriber.

Quota

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

Extended attributes represent the data structure of the extended attributes associated with the subscriber.

Subscriber Integration Modes

The Cisco SCE platform supports the following two modes of dynamic subscriber integration:

- **Push Mode**, page 2-3
- **Pull Mode**, page 2-3

Push Mode

In Push Mode integration, an external server introduces (pushes) the subscribers to the Cisco SCE platform as shown in Figure 2-1. This push occurs whenever a new subscriber logs in to the network or the external server presumes to know all the subscribers and introduces them to the Cisco SCE box when they connect.

![Figure 2-1 Push Mode Schematic](image1)

Pull Mode

In Pull Mode integration, the Cisco SCE platform requests subscriber data from the external entity when it encounters traffic of an unknown (anonymous) subscriber, as shown in Figure 2-2. The external entity retrieves the required subscriber information and sends it back to the Cisco SCE platform.

![Figure 2-2 Pull Mode Schematic](image2)
The Cisco SCE Subscriber API is implemented by using a nonblocking mode. Nonblocking methods return immediately, even before the completion of a subscriber provisioning operation. The nonblocking mode method is best suited 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 not returned at all. The API supports retrieval of operation results through an operation result handler described in the “Result Handling” section on page 5-27.

Figure 2-3 illustrates the nonblocking mode method during a subscriber provisioning operation.

Figure 2-3 Nonblocking Mode

Error logging enables you to monitor the results of an operation or to inspect the parameters that the operation uses.
Indications Listeners

The API enables you to receive an indication when certain events occur on the Cisco SCE platform as shown in Figure 2-4. The API dispatches the indications received from the Cisco SCE to the interested entities, called listeners, by activating the relevant listeners callback methods. The indications are separated into several logical groups. You can define only one listener for each group of indications.

![Figure 2-4 Indication Listeners](image-url)

To receive certain indications, 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 Cisco SCE Subscriber API provides three types of indications when separate listeners are registered to the following types of indications:

- Login-pull indications
- Logout indications
- Quota indications

For more information about listener indications, see Chapter 3, “Application Programming Interface Events.”
Supported Topologies

The Cisco SCE Subscriber API supports the following topologies:

- One policy server (or one two-node cluster) that is responsible for all aspects of the subscriber provisioning process as shown in Figure 2-5.

- Three policy servers (or three two-node clusters)—Every server is responsible for a different aspect of the subscriber provisioning process as shown in Figure 2-6.

- 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 only one aspect (any combination is allowed) as shown in Figure 2-7.
DHCP Lease Query Login Event Generator (LEG), which is responsible for mapping a network ID to a subscriber ID, with one or more policy servers as shown in Figure 2-7. Figure 2-8 shows the DHCP Lease Query LEG.

Figure 2-8  Supported Topologies - DHCP Lease Query LEG

The Cisco Service Control SM, which is responsible for mapping network ID to subscriber ID, with one or more policy servers as shown in Figure 2-9. The number of policy servers depends on whether the Subscriber Manager is used for policy profile provisioning in addition to the network ID.

Figure 2-9  Supported Topologies - Subscriber Manager
Caution

The API itself does not limit the use of any topology; however, the Cisco SCE platform does not correlate among all the entries (policy servers) that perform subscriber provisioning. Therefore, be 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 Cisco SCE platform can receive different information for the same subscriber from the two policy servers responsible for the same aspect of the subscriber provisioning. This can cause a loss of synchronization with at least one policy server. For example, if two policy servers correlate subscriber ID to network ID for the same subscriber, the Cisco SCE always synchronizes with the policy server that performed the last update for this subscriber.
Multithreading Support

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

---

Note

In a multithreaded scenario, the order of invocation is guaranteed. The API performs operations in the same order in which they were called.

---

Autoreconnect Support

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

Reliability Support

The Cisco SCE Subscriber API is implemented as a reliable API. The API ensures that no requests to the Cisco SCE are lost and no indication from the Cisco SCE is lost. The API maintains internal storage for all API requests that are sent to the Cisco SCE. Only after receiving an acknowledgment from the Cisco SCE that the request was processed does it consider the request as committed. The API can then remove the request from its internal storage. If a connection failure occurs between the API and the Cisco SCE, the API accumulates all requests in its internal storage until the connection to the Cisco SCE is re-established. On reconnection, the API resends all noncommitted requests to the Cisco SCE, ensuring that no requests are lost.

---

Note

The order of resending requests is guaranteed. The API resends the requests in the same order in which 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 in which only one server is active at any time. The other server, in standby, is not connected to the Cisco SCE. For more information, see the “Implementing High Availability” section on page 5-52.

Synchronization

The Cisco SCE and policy server must be kept synchronized regarding the subscribers for which the Cisco SCE manages internal parameters. Otherwise, the Cisco SCE can confuse the traffic of one subscriber with the traffic of another; or the service level agreement (SLA) of the subscriber might not be enforced because of a change in the policy that did not reach the Cisco SCE. For more information, see the “Cisco SCE-API Synchronization” section on page 5-45.
Practical Tips

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

- Connect once to the Cisco SCE and maintain an open API connection to the Cisco SCE at all times, using the API many times. Establishing a connection is a timely procedure, which allocates resources on the Cisco SCE side and the API client side.

- Share the API connection among your threads—it is better to have one connection per policy server. Multiple connections require more resources on the Cisco SCE and the 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, increase the internal buffer size accordingly to hold these bursts (nonblocking flavor).

- During the integration, use the logging capabilities that are described in the “Cisco SCE Logging” section on page 6-2 and in the “API Client Logging” section on page 6-6. View the API operations in the Cisco SCE client logs to troubleshoot problems during the integration.

- Use debug mode for the policy server application that logs or prints the return values of the operations.

- Use the automatic reconnect feature to improve the resiliency of the connection to the Cisco SCE.
Application Programming Interface Events

Introduction

This chapter describes the various events accessed by the Cisco Service Control Engine (SCE) Subscriber application programming interface (API).

It consists of the following sections:

- Introduction, page 3-1
- API Events, page 3-2
The API accesses a set of events, which are a predefined set of messages that are passed back and forth between the policy server and the Cisco SCE platform as shown in Figure 3-1.

Every message is assigned a type according to the purpose of the message:

- **Request**—Requests information or performance of an action. Not all requests derive a response.
- **Response**—Answers a previous request.
- **Indication**—Indicates to the other side that an event occurred.

Most of the events can be used for both push and pull modes. See the “Subscriber Integration Modes” section on page 2-3.

The events are divided into the following subscriber provisioning process groups:

- **Network ID management events**—Includes events associated with modifying subscriber network ID mapping.
- **Policy Profile management events**—Includes events associated with modifying subscriber policy profile parameters.
- **Quota management events**—Includes events relating to the management of subscriber quota.
- **Cisco SCE synchronization management events**—Includes events associated with managing the Cisco SCE synchronization process.

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

The following sections describe each type of event:

- **Network ID Management Events, page 3-2**
- **Policy Profile Management, page 3-5**
- **Quota Management, page 3-5**
- **Cisco SCE Synchronization Procedure Events, page 3-7**

### Network ID Management Events

The following events are described in this section:

- **Login Events, page 3-3**
- **Logout Events, page 3-4**
- **Network ID Update Event, page 3-5**
Login Events

Login events occur when a subscriber connects to the network. The events for pull and push modes differ.

- Push Mode, page 3-3
- Pull Mode, page 3-3

Push Mode

The integration of Push Mode assumes that the Policy Server triggers introduction of the subscriber to the Cisco SCE. For example, the server receives a subscriber login indication from an external entity such as authentication, authorization, and accounting (AAA), extracts the required subscriber attributes, and pushes the information to the Cisco SCE platform as shown in Figure 3-2.

![Figure 3-2 Login Events - Push Mode](image)

The subscriber login operation either causes the creation of a new subscriber record in the Cisco SCE or updates an existing subscriber. For example, for cable modem networks, the subscriber is a cable modem and the customer premise equipments (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 adds the CPE IP address to the subscriber network ID list.

Pull Mode

The integration of Pull Mode assumes that the Cisco 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 Cisco SCE initiates a request to the external system (a login-pull request) that either provides the subscriber login information (a login-pull reply) or is omitted when no information exists for this IP. The login information provided to the Cisco SCE replaces the anonymous subscriber with the actual subscriber and enforces the correct policy as shown in Figure 3-3.

If the external system rejects the login and the traffic continues from the anonymous subscriber, the Cisco SCE retries the pull request.

![Figure 3-3 Login Events - Pull Mode](image)
Despite being classified as network ID management event, LOGIN-REQUEST and LOGIN-PULL-RESPONSE events are optimized to allow sending all subscriber information to the Cisco SCE. Use these events for policy profile and quota updates when a single policy server performs all parts of subscriber provisioning. For topologies that include multiple policy servers, use separate events to update policy profile and quota information, as described in the following sections. For more information about topologies, see the “Supported Topologies” section on page 2-6.

Logout Events

The logout event indicates that the subscriber no longer uses a certain network ID. A logout event might not result in the removal of the subscriber record from the Cisco SCE as shown in Figure 3-4. For example, in cable modem networks, when there is more than one CPE item connected to the same modem, the logout of one CPE might not lead to the removal of a subscriber if another CPE remains connected. The subscriber is removed when all the CPEs (subscriber network IDs) are disconnected.

![Figure 3-4 Logout Request Event](image)

The logout event in Pull Mode might occur, for example, when the Cisco SCE detects that the subscriber is not active for a specific time interval. The Cisco SCE logs out the subscriber and sends a LOGOUT-INDICATION event as shown in Figure 3-5.

![Figure 3-5 Logout Indication Event](image)

The LOGOUT-INDICATION event can also follow the Logout operation. This sequence of actions occurs when a subscriber is removed, for example, when no more valid network mappings (IP) are associated with this subscriber as shown in Figure 3-6.

![Figure 3-6 Logout Request Event](image)
Network ID Update Event

The network update event is a REQUEST from the Policy Server to the Cisco SCE to update the network ID of the subscriber that exists in the Cisco SCE platform as shown in Figure 3-7. This event does not require any RESPONSE.

![Figure 3-7 Network ID Update Event](image)

Policy Profile Management

The policy profile management consists of one event, namely, profile update event.

Profile Update

The profile update event is a REQUEST from the policy server to the Cisco SCE to update the policy profile of the subscriber that exists in the Cisco SCE platform as shown in Figure 3-8. This event does not require any RESPONSE.

![Figure 3-8 Profile Update Event](image)

Note: The LOGIN-REQUEST event and LOGIN-PULL-RESPONSE event can also update the policy profile.

Quota Management

This section consists of these topics:

- Quota Update, page 3-6
- Get Quota Status, page 3-6
- Quota Status, page 3-6
- Quota Below Threshold, page 3-7
- Quota Depleted, page 3-7
- Quota State Restore, page 3-7
Quota Update

The Quota Update event is a REQUEST from the policy server to the Cisco SCE to update the quota of the subscriber that exists in the Cisco SCE platform as shown in Figure 3-9. This event does not require any RESPONSE event.

Figure 3-9  Quota Update Event

Note

The LOGIN-REQUEST event and LOGIN-PULL-RESPONSE event can also update the quota.

Get Quota Status

The Get Quota Status event is a REQUEST from the policy server to the Cisco SCE to report the quota information of the subscriber that exists in the Cisco SCE platform as shown in Figure 3-10. A QUOTA-STATUS-INDICATION event follows this event.

Figure 3-10  Get Quota Status Event

Note

The Cisco SCE can issue a QUOTA-STATUS-INDICATION event periodically without a specific request from the policy server. See the “Quota Status” section on page 3-6.

Quota Status

The Cisco SCE uses the quota status INDICATION event to notify the policy server about the remaining quota as shown in Figure 3-11. This event is invoked periodically according to a preconfigured time interval.

Figure 3-11  Quota Status Event
Quota Below Threshold

The Cisco 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 as shown in Figure 3-12. An UPDATE-QUOTA-REQUEST event from the policy server to the Cisco SCE can follow this event, but is not mandatory.

Figure 3-12 Quota Below Threshold Event

Quota Depleted

The Cisco SCE uses the quota depleted INDICATION event to notify the policy server that the quota for certain services of the specific subscriber is depleted as shown in Figure 3-13. An UPDATE-QUOTA-REQUEST event from the policy server to the Cisco SCE can follow this event.

Figure 3-13 Quota Depleted Event

Quota State Restore

The quota state restore event is an INDICATION from the Cisco SCE to the policy server to restore the quota of the subscriber that exists in the Cisco SCE platform as shown in Figure 3-14. This event is invoked immediately after a subscriber logs in to the Cisco SCE. A quota update event from the Policy Server can follow this event.

Figure 3-14 Quota State Restore Event

Cisco SCE Synchronization Procedure Events

This section consists of these topics:

- Start Synchronization, page 3-8
- End Synchronization, page 3-8
- Get Subscribers, page 3-8
Start Synchronization

The start synchronization REQUEST event notifies the Cisco SCE that the synchronization process is about to start as shown in Figure 3-15. The Cisco SCE uses this REQUEST to perform internal operations that are required to prepare for synchronization. This event has a push and a pull component.

![Figure 3-15 Start Synchronization Event](image)

End Synchronization

The End Synchronization REQUEST event notifies the Cisco SCE that the synchronization process ended as shown in Figure 3-16. This event has a push and a pull component.

![Figure 3-16 End Synchronization Event](image)

Get Subscribers

During a Cisco SCE Pull Mode synchronization process, the policy server is required to retrieve all subscribers that the Cisco SCE is currently managing. The policy server sends the GET-SUBSCRIBERS-BULK-REQUEST event to the Cisco SCE to retrieve the next bulk of subscribers that the Cisco SCE is currently managing. When the Cisco SCE receives this request, it responds with the GET-SUBSCRIBERS-BULK-RESPONSE event that supplies the subscriber names and network IDs as shown in Figure 3-17.

![Figure 3-17 Get Subscribers Event](image)

For more information, see the “Pull Mode” section on page 2-3 and the “Pull Mode Synchronization Procedure” section on page 5-47.
Application Programming Interface Data Types

Introduction

This chapter describes the API data types used in the Cisco Service Control Engine (SCE) Subscriber application programming interface (API).

- Subscriber ID, page 4-1
- Network ID Mappings, page 4-2
- Cisco SCA BB Subscriber Policy Profile, page 4-5
- Subscriber Quota, page 4-6
- Extended Attributes, page 4-9
- Bulk Operations Data Types, page 4-10
- Virtual Link Data Types, page 4-19

Subscriber ID

Most methods of the Cisco SCE Subscriber APIs require the subscriber ID as an input parameter. The Subscriber ID is a string representing a subscriber name or a content manager MAC address. This section lists the formatting rules for a subscriber ID.

The subscriber name is case-sensitive. It can contain up to 64 characters, which include all printable characters with an ASCII code from 32 through 126 (inclusive); except for 34 ("), 39 ('), and 96 (`).

The following examples present valid subscriber names:

```java
String subID1="john";
String subID2="john@yahoo.com";
```
Network ID Mappings

A Network ID is a network identifier that the Cisco SCE device associates with a specific subscriber record. A typical example of a network ID mapping is an IP address. Currently, the Cisco SCE supports mappings to an IP address, IP range, private IP address over VLAN, private IP range over VLAN, and VLAN.

The network ID class represents various types of subscriber network identification. The API supports the following subscriber mapping types:
- IP addresses or IP ranges
- Private IP addresses or private IP ranges over VLAN
- VLAN mappings
- (Starting from Cisco Service Control Subscriber Manager, Release 3.8.5) IPv6 address

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

NetworkID class constructors are defined as follows:

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

Parameters of the NetworkID constructors are:
- java.lang.String mapping identifier or array of mapping identifiers
- Short mapping type or array of mapping types

When passing arrays, the mapping types array must contain either the same number of elements as the mappings array or a single element:
- Use NetworkID.TYPE_IP, NetworkID.TYPE_IPV6, or NetworkID.TYPE_VPN constants if the array contains more than one element.

```
<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>The NetworkID.TYPE_VLAN and NetworkID.ALL_VLAN_MAPPINGS constants are deprecated. Use the NetworkID.TYPE_VPN and NetworkID.ALL_VPN_MAPPINGS constants instead.</td>
</tr>
</tbody>
</table>
```

- Use one of the following mappings when a single array element is used:
  - NetworkID.ALL_IP_MAPPINGS
  - NetworkID.ALL_IPV6_MAPPINGS
  - NetworkID.ALL_VLAN_MAPPINGS
Specifying IP Address Mapping

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

\[ \text{IP-Address} = [0-255].[0-255].[0-255].[0-255] \]

**Example**

- 216.109.118.66
  
  The mapping type of an IP address is provided in the NetworkID class:
  
  \[ \text{com.scms.common.NetworkID.TYPE_IP} \]
  
  \[ \text{com.scms.common.NetworkID.ALL_IP_MAPPINGS} \]
  
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 ones in a bit mask:

\[ \text{IP-Range} = [0-255].[0-255].[0-255].[0-255]/[0-32] \]

**Examples**

- 10.1.1.10/32 is an IP range with a complete mask, that is, a regular IP address.
- 10.1.1.0/24 is an IP range with a 24-bit mask, that is, all the addresses in the range 10.1.1.0 through 10.1.1.255.

**Note**

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

Specifying Private IP Address or Private IP Range over VLAN Mapping

The string format of an IP address and an IP range are described in the “Specifying IP Address Mapping” section on page 4-3 and the “Specifying IP Range Mapping” section on page 4-3. When the network ID mapping uses an IP address or range over VLAN, the string format includes the VLAN number.

**Examples**

- 10.1.1.10@1 is an IP address over VLAN1.
- 10.1.0.24@2 is an IP range with a 24-bit mask, that is, all the addresses in the range 10.1.1.0 through 10.1.1.255 over VLAN2.

**Note**

The mapping type of an IP address or IP range over VLAN 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 range from 2 to 2046. The com.scms.common.NetworkID class provides the VLAN mapping type:

- Mapping type of an IP address is provided in the class Network ID:
  com.scms.common.NetworkID.TYPE_VPN;
- com.scms.common.NetworkID.ALL_VLAN_MAPPINGS specifies that all the entries in the mapping identifiers array are VLAN mappings.

Specifying IPv6 Address Mapping

The string format for IPv6 address mapping is a hexadecimal string that follows the RFC 2373 standard.

Example
2005:1:2:23::/64

The mapping type of an IP address is provided in the NetworkID class:
com.scms.common.NetworkID.TYPE_IPV6:
com.scms.common.NetworkID.ALL_IPV6_MAPPINGS specifies that all the entries in the array of mapping identifiers are IPv6 mappings.

Network ID Mapping Examples

The following is a list of sample network ID mappings:

- The following example constructs a NetworkID with a single IP address:
  NetworkID nid = new NetworkID("1.1.1.1",NetworkID.TYPE_IP)

- The following example constructs a NetworkID with a range of IP addresses:
  NetworkID nid = new NetworkID("1.1.1.1/24",NetworkID.TYPE_IP)

- The following example constructs a 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)

- The following example constructs a NetworkID with a single IPv6 address:
  NetworkID nid = new NetworkID("4faf:f0ff:ddde:1234::/64", NetworkID.TYPE_IPV6 )

- The following example constructs a NetworkID with a range of IPv6 addresses:
  NetworkID nid = new NetworkID("4faf:f0ff:ddde:1234::/48", NetworkID.TYPE_IPV6)

- The following example constructs a NetworkID with multiple IPv6 addresses:
  NetworkID nid = new NetworkID(new
  String[]{"4faf:f0ff:ddde:1234::/64","2005:1:2:23::/64"}, NetworkID.ALL_IPV6_MAPPINGS)

- The following example constructs a NetworkID with a single VLAN address:
  NetworkID nid = new NetworkID("23",NetworkID.TYPE_VPN)
Cisco SCA BB Subscriber Policy Profile

The Policy Profile presents the subscriber policy information. A policy profile has two main parts. The policy package identifies a statically defined policy and a set of subscriber policy properties that can be dynamic. The package ID identifies the policy package. Most of the rules enforced on the subscriber traffic are derived from the package ID.

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

The Cisco SCA BB supports the subscriber properties packageId, monitor, upVlinkId, and downVlinkId. For a description of the subscriber properties, see the Cisco Service Control Application for Broadband User Guide.

PolicyProfile Class

The API provides a PolicyProfile class to format subscriber policy profiles that the API requires to operate.

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

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

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

```text
property_name=property_value
```

The following method enables you to add a policy property to the profile:

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

**Note**

This method is not optimized for performance. To optimize performance, use the PolicyProfile constructor.

**Example**

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

To provision subscriber quotas in Cisco SCA BB, you define quota buckets. Each subscriber has 16 buckets. You define each bucket for volume or sessions. When a subscriber uses a particular service, the volume or number of sessions consumed is subtracted from one of the buckets.

Use the SCA BB Console to create a general policy definition, which specifies which bucket to use for each service. Volume consumption is measured in ubits of Layer 3 kilobytes. The number of sessions determines the consumption of session buckets. For example, it is possible to specify that the browsing and e-mail services consume quota from Bucket 1, P2P service consumes quota from Bucket 2, and that all other services are not bound to any particular bucket.

A quota bucket has the following components:

- Bucket ID—Unique identifier of the bucket (string) as specified by the predefined policy. Valid values are numbers in the range 1 to 16
- Bucket value—Quota bucket value (long)

Quota operation dynamically modifies the quota buckets for a subscriber. There are two types of quota operations:

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

Examples

Current subscriber quota values at the Cisco SCE are as shown in Figure 4-1.

Figure 4-1  Subscriber Quota - Current Values

FTP  80
P2P  1
Browsing  60
Figure 4-2 shows the application of actions to the existing quota.

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

- **SET**
  - `85`
  - FTP
- **ADD**
  - `5`
  - P2P
- **ADD**
  - `10`
  - Browsing

After performing the quota actions, the result is as shown in Figure 4-3.

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

- **85**
  - FTP
- **6**
  - P2P
- **70**
  - Browsing

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**, page 4-7
- **SCAS_BB_QuotaOperation**, page 4-8

---

**SCAS_BB_Quota**

The SCAS_BB_Quota class implements the Quota interface, which the QuotaListenerEx interface uses in all callback functions. See the “QuotaListenerEx Interface Class” section on page 5-19.

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

```java
public SCAS_BB_Quota (String[] bucketIDs,
                       long[] bucketValues)
```
Chapter 4 Application Programming Interface Data Types

SCAS_BB_QuotaOperation

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

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

The following method retrieves quota bucket IDs:

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

The following method retrieves quota buckets values:

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

The quotaProfileId parameter identifies the quota profile, which is the package ID. The following method retrieves the quota profile ID:

```java
public int getQuotaProfileId()
```

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

- 0—Configured time was reached, for example, every 2 minutes
- 1—A subscriber logout triggered a quota status event
- 2—A package change triggered a quota status event

The following method retrieves the reason:

```java
public int getReason()
```

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

The following method retrieves the time stamp:

```java
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 operations (see the “Login Operation” section on page 5-30) and update quota operations (see the “quotaUpdate Operation” section on page 5-41).

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

```java
public SCAS_BB_QuotaOperation (String[] IDs,
                                long[] values,
                                short[] actions)
```
The following method retrieves the quota bucket IDs:
```java
public String[] getBucketIDs()
```

The following method retrieves the quota bucket values:
```java
public long[] getBucketValues()
```

The following method retrieves the quota bucket actions:
```java
public short[] getBucketActions()
```

## Extended Attributes

The Extended Attributes class represents the data structure of extended attributes associated with a subscriber. Extended attributes can be Vendor Specific Attributes (VSA) or other attributes in RADIUS or other IP packets. The Cisco SCE Subscriber API sends extracted extended, attributes during the subscriber login and bulk login operations, both in Push and Pull Mode.

The following constructor is used for the esa object:
```java
ExtendedAttributes(long[] esaCodeArr, 
                   long[] vendorIdArr, 
                   byte[] esaDataTypeArr, 
                   java.lang.String[] esaDataArr)
```

### Parameters

The extendend attributes parameters are as follows:

- `esaCodeArr`—Code or attribute ID of the extended attributes.
- `vendorIdArr`—Vendor ID of the extended attributes. If non-VSA, this value is zero (0).
- `esaDataTypeArr`—Data type of the extended attributes, which can be one of the following types: INTEGER, IP_ADDRESS, STRING, or OCTET_STRING.
- `esaDataArr`—Actual data for the specific extended attributes.

**Note**

Only the Cisco SCE 8000 platform supports the Extended Attributes class.
Bulk Operations Data Types

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

Consider the following details when using the bulk operations:

- All bulk classes are inherited from the common BulkBase class.
- Because of the memory constraints of the Cisco 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()
```

You can use this iterator for iterations over the bulks received from the Cisco SCE in various indications (for example, logoutBulkIndication and loginPullBulkResponseIndication) or for inspecting the data you provided to various operations when an operation fails.

The iterator provides the following methods for data retrieval:

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

The next() method returns a SubscriberData object.

The SubscriberData class retrieves the information of a single subscriber contained within the bulk.

SubscriberData

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

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

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

Login_BULK Class

The Login_BULK class represents the bulk of subscribers and it includes all the data required for the loginBulk operation:

- Constructor, page 4-11
- addBulkEntry Method, page 4-11
- Examples, page 4-12
## Constructor

To construct the Login_BULK class filled with the data, use the following constructor:

```java
class Login_BULK {
    public Login_BULK(String[] subscriberIDs,
                      NetworkID[] networkIDs,
                      boolean[] additive,
                      PolicyProfile[] policy,
                      QuotaOperation[] quota)
}
```

To support extended attributes in the Login_BULK class, a new parameter was added to the constructor starting in Cisco SCE Subscriber, Release 3.6.5:

```java
class Login_BULK {
    public Login_BULK(String[] subscriberIDs,
                      NetworkID[] networkIDs,
                      boolean[] additive,
                      PolicyProfile[] policy,
                      QuotaOperation[] quota,
                      ExtendedAttributes[] esa)
}
```

To construct an empty Login_BULK class, use the following method:

```java
class Login_BULK {
    public Login_BULK()
}
```

### Parameters

The Login_BULK class parameters are as follows:

- **subscriberIDs**—Unique ID of the subscriber. For a description of the subscriber ID format, see the “Subscriber ID” section on page 4-1.
- **networkIDs**—Network identifier of the subscriber. For more information, see the “Network ID Mappings” section on page 4-2.
- **additive**—If this parameter is set to TRUE, the specified networkID is added to the existing networkIDs of the subscriber. Otherwise, the specified networkID replaces the existing networkIDs.
- **policy**—Policy profile of the subscriber. For more information, see the “Cisco SCAB Subscriber Policy Profile” section on page 4-5.
- **quota**—Quota of the subscriber. For more information, see the “Subscriber Quota” section on page 4-6.
- **esa**—Extended attributes associated with the subscriber and passed in the login operation.

## addBulkEntry Method

To add entries to the bulk, use the following method:

```java
class Login_BULK {
    public void addBulkEntry(String subscriberID,
                            NetworkID networkID,
                            boolean networkIdAdditive,
                            PolicyProfile policy,
                            QuotaOperation quota)
}
```

To support extended attributes in the addBulkEntry class, a new parameter was added to the constructor starting in Cisco SCE Subscriber, Release 3.6.5:

```java
class Login_BULK {
    public void addBulkEntry(String subscriberID,
                            NetworkID networkID,
                            boolean networkIdAdditive,
                            PolicyProfile policy,
                            ExtendedAttributes[] esa)
}
```
Login_BULK Class

QuotaOperation quota,
ExtendedAttributes esa)

Parameters
The addBulkEntry method parameters are as follows:

- subscriberID—Unique ID of the subscriber. For a description of the subscriber ID format, see the “Subscriber ID” section on page 4-1.
- networkID—Network identifier of the subscriber. See the “Network ID Mappings” section on page 4-2.
- networkIdAdditive—If you set this parameter 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 the “Cisco SCA BB Subscriber Policy Profile” section on page 4-5.
- quota—Quota of the subscriber. See the “Subscriber Quota” section on page 4-6.
- esa—Extended attributes associated with the subscriber and passed in the login operation.

Examples

- Login_BULK Object Usage, page 4-12
- Manipulating Login_BULK, page 4-13

Login_BULK Object Usage

// 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

// 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, which requires you to enter only subscriber IDs, uses the SubscriberID_BULK class. See the “logoutBulkIndication Callback Method” section on page 5-19.

- Constructors, page 4-13
- addBulkEntry Method, page 4-13

Constructors

To construct 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()

Parameter

subscriberID—Unique ID of the subscriber. For a description of the subscriber ID format, see the “Subscriber ID” section on page 4-1.

addBulkEntry Method

To add entries to the SubscriberID bulk, use the following method:

addBulkEntry(String subscriberID)

Parameter

subscriberID—Unique ID of the subscriber. For a description of the subscriber ID format, see the “Subscriber ID” section on page 4-1.
NetworkAndSubscriberID_BULK Class

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

- `getSubscribersBulkResponse` callback (see the “LoginPullListener Interface Class” section on page 5-15)
- `logoutBulk` operation (see the “logoutBulk Operation” section on page 5-36)
- `networkIDUpdateBulk` operation (see the “networkIdUpdateBulk Operation” section on page 5-39)

Constructors

To construct `NetworkAndSubscriberID_BULK` with the subscriberID and networkID data, use the following constructor:

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

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

```java
public NetworkAndSubscriberID_BULK()
```

Parameters

The `NetworkAndSubscriberID_BULK` class parameters are as follows:

- `subscriberID`—Unique ID of the subscriber. For a description of the subscriber ID format, see the “Subscriber ID” section on page 4-1.
- `networkID`—Network identifier of the subscriber. See the “Network ID Mappings” section on page 4-2.
- `networkIDAdditive`—If this parameter is set to TRUE, the supplied NetworkID is added to the existing networkIDs of the subscriber. Otherwise, the supplied networkID replaces the existing networkIDs.

addBulkEntry Method

To add entries to the bulk, use the following method:

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

Parameters

The `addBulkEntry` method parameters are as follows:

- `subscriberID`—Unique ID of the subscriber. For a description of the subscriber ID format, see the “Subscriber ID” section on page 4-1.
- `networkID`—Network identifier of the subscriber. See the “Network ID Mappings” section on page 4-2.
- `networkIDAdditive`—If this parameter is set to TRUE, the supplied 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, page 4-15
- addBulkEntry Method, page 4-16

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

```
public LoginPullResponse_BULK()
```

Constructors

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

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

To support extended attributes in the LoginPullResponse_BULK class, a new parameter was added to the constructor starting in Cisco SCE Subscriber, Release 3.6.5:

```
public LoginPullResponse_BULK(String[] anonymousIDs,
                               String[] subscriberIDs,
                               NetworkID[] networkIDs,
                               boolean[] additive,
                               PolicyProfile[] policy,
                               QuotaOperation[] quota,
                               ExtendedAttributes[] esa)
```

Parameters

The LoginPullResponse_BULK class parameters are as follows:

- anonymousIDs—Identifier of the anonymous subscriber. The Cisco SCE sends this parameter within the loginPullRequest/loginPullBulkRequest indication (see the “loginPullRequest Callback Method” section on page 5-16 and the “loginPullRequestBulk Callback Method” section on page 5-17). See the “Subscriber Integration Modes” section on page 2-3.
- subscriberIDs—Unique ID of the subscriber. For a description of the subscriber ID format, see the “Subscriber ID” section on page 4-1.
- networkIDs—Network identifier of the subscriber. See the “Network ID Mappings” section on page 4-2.
- additive—if this parameter 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 the “Cisco SCA BB Subscriber Policy Profile” section on page 4-5.
- quota—Quota of the subscriber. See the “Subscriber Quota” section on page 4-6.
- esa—Extended attributes associated with the subscriber and passed in the login operation.
Chapter 4 Application Programming Interface Data Types

PolicyProfile_BULK Class

addBulkEntry Method

To add entries to the bulk, use the following method:

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

To support extended attributes in the addBulkEntry class, a new parameter was added to the constructor starting in Cisco SCE Subscriber, Release 3.6.5:

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

Parameters

The addBulkEntry method parameters are as follows:

- **anonymousSubscriberID**—Identifier of the anonymous subscriber. The Cisco SCE sends this parameter within the loginPullRequest/loginPullBulkRequest indication (see the “loginPullRequest Callback Method” section on page 5-16 and the “loginPullRequestBulk Callback Method” section on page 5-17). See the “Subscriber Integration Modes” section on page 2-3.

- **subscriberID**—Unique ID of the subscriber. For a description of the subscriber ID format, see the “Subscriber ID” section on page 4-1.

- **networkID**—Network identifier of the subscriber. See the “Network ID Mappings” section on page 4-2.

- **networkIdAdditive**—If this parameter is set to TRUE, the supplied 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 the “Cisco SCA BB Subscriber Policy Profile” section on page 4-5.

- **quota**—Quota of the subscriber. See the “Subscriber Quota” section on page 4-6.

- **esa**—Extended attributes associated with the subscriber and passed in the login operation.

PolicyProfile_BULK Class

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

- **Constructors, page 4-17**

- **addBulkEntry Method, page 4-17**
Constructors

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

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

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

```java
public PolicyProfile_BULK()
```

Parameters

The PolicyProfile_BULK class parameters are as follows:

- `subscriberID`—Unique ID of the subscriber. For a description of the subscriber ID format, see the “Subscriber ID” section on page 4-1.
- `policy`—Policy profile of the subscriber. See the “Cisco SCA BB Subscriber Policy Profile” section on page 4-5.

addBulkEntry Method

To add entries to the bulk, use the following method:

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

Parameters

The addBulkEntry Method parameters are as follows:

- `subscriberID`—The unique ID of the subscriber. See the “Subscriber ID” section on page 4-1 for the subscriber ID format description.
- `policy`—Policy profile of the subscriber. See the “Cisco SCA BB Subscriber Policy Profile” section on page 4-5.

Quota_BULK Class

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

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

Constructors

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

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

To construct an empty Quota_BULK table, use the following method:

```java
public Quota_BULK()
```
Parameters
The Quota_BULK class parameters are as follows:

- subscriberID—Unique ID of the subscriber. For a description of the subscriber ID format, see the “Subscriber ID” section on page 4-1.
- quota—Quota of the subscriber. See the “Subscriber Quota” section on page 4-6.

addBulkEntry Method
To add entries to the bulk, use the following method:

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

Parameters
The addBulkEntry method parameters are as follows:

- subscriberID—Unique ID of the subscriber. For a description of the subscriber ID format, see the “Subscriber ID” section on page 4-1.
- quota—Quota of the subscriber. See the “Subscriber Quota” section on page 4-6.

QuotaOperation_BULK Class
The QuotaUpdateBulk operation and the login operation use this class, which represents a bulk of subscriber IDs and subscriber quota operations.

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

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

To construct an empty QuotaOperation_BULK table, use the following method:

```java
public QuotaOperation_BULK()
```

Parameters
The QuotaOperation_BULK class parameters are as follows:

- subscriberID—Unique ID of the subscriber. For a description of the subscriber ID format, see the “Subscriber ID” section on page 4-1.
- quotaOperation—Quota operation to perform on the quota of the subscriber. See the “Subscriber Quota” section on page 4-6.

addBulkEntry Method
To add entries to the bulk, use the following method:

```java
addBulkEntry(String subscriberID, QuotaOperation quotaOperation)
```
Chapter 4 Application Programming Interface Data Types

Virtual Link Data Types

This section describes the various data types related to the virtual link API.

VLink Class

The VLink class represents the data structure of the virtual link associated with a subscriber. When a virtual link is created, a channel too gets created by default for this virtual link.

The following method adds channels to a virtual link by passing a channel object:

```
addChannel(Channel channel)
```

The following method adds channels to a virtual link by passing an array of channel objects:

```
addChannel(Channel[] channels)
```

Constructors

To construct a VLink class based on vlinkId, vlinkName, direction, agcOffset, vlinkPIR, vlinkCIR, and vlinkA1Value, use the following constructor:

```
public VLink(int vlinkId, String vlinkName, int direction, int agcOffset, int vlinkPIR,
        int vlinkCIR, int vlinkA1Value)
```

To construct a VLink class based on vlinkId, vlinkName, direction, agcOffset, vlinkPIR, and vlinkCIR, use the following constructor:

```
public VLink(int vlinkId, String vlinkName, int direction, int agcOffset, int vlinkPIR,
        int vlinkCIR)
```

Parameters

The VLink class parameters are as follows:

- vlinkId—Describes the unique ID of a virtual link.
- vlinkName—Describes the name of a virtual link.
- direction—Describes the upstream and downstream directions that are represented by the integer constants in Cisco SCE. The value is 0 for upstream direction and 1 for downstream direction.
- agcOffset—Describes the offset value of Aggregate Global Control (AGC) that is configured.
- vlinkPIR—Describes the peak information rate (PIR) of a virtual link.
- vlinkCIR—Describes the committed information rate (CIR) of a virtual link.
- vlinkA1Value—Describes the Assurance Level (AL) value of a virtual link.
Channel Class

A channel class represents the data structure of the channel associated with a subscriber. It also contains the direction and vlinkId parameters of the virtual link to which the corresponding channel is associated.

Constructors

To construct a channel class based on a channelId, channelName, channelPIR, channelCIR, and channelAlValue, use the following constructor:

```java
public Channel(int channelId, String channelName, int channelPIR, int channelCIR, int channelAlValue)
```

To construct a channel class based on channelId, channelName, channelPIR, and channelCIR use the following constructor:

```java
public Channel(int channelId, String channelName, int channelPIR, int channelCIR)
```

Parameters

The channel class parameters are as follows:

- `channelId`—Describes the unique ID of a channel class.
- `channelName`—Describes the name of a channel class.
- `channelPIR`—Describes the PIR of a channel class.
- `channelCIR`—Describes the CIR of a channel class.
- `channelAlValue`—Describes the Assurance Level value of a channel class.

VLinkDisabledException Class

VLinkDisabledException is a user-defined exception in com.scms.api.sce.prpc. This exception is displayed when a user tries to use the virtual link APIs with the virtual link being disabled in the Cisco SCE.

VLinkModeException Class

VLinkModeException is a user-defined exception that is displayed when virtual link APIs are not used in the user mode. To enable the user mode, use the following method:

```java
enableVlinkMode()
```

Note

We recommend that in order to enable the user mode, you use the `enableVlinkMode()` command before the Virtual Link API gets connected to the Cisco SCE platform.
Programming with the Cisco SCE Subscriber API

Published: August 30, 2014

Introduction

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

- API Classes, page 5-2
- Programming Guidelines, page 5-4
- PRPC_SCESubscriberApi Class, page 5-5
- Virtual Link API, page 5-8
- Connection Monitoring, page 5-23
- Cisco SCE Cascade Topology Support, page 5-24
- Result Handling, page 5-27
- Subscriber Provisioning Operations, page 5-30
- Cisco SCE-API Synchronization, page 5-45
- Advanced API Programming, page 5-52
- API Code Examples, page 5-53
API Classes

The following list presents the classes provided by the API:

- Package com.scms.api.sce.prpc, page 5-2
- Package com.scms.api.sce, page 5-2
- Package com.scms.common, page 5-3

Package com.scms.api.sce.prpc

- PRPC_SCESubscriberApi Class, page 5-5—Main API class.
- Package com.scms.api.sce.prpc, page 5-2

Package com.scms.api.sce

- Indication Listeners, page 5-2
- Connection Monitoring, page 5-2
- Cisco SCE Cascade Topology Support, page 5-2
- Operation Results Handling, page 5-2

Indication Listeners

- LoginPullListener Interface Class, page 5-15 (interface)
- DualLoginPullListener Interface Class, page 5-17 (interface)
- LogoutListener Interface Class, page 5-19 (interface)
- QuotaListenerEx Interface Class, page 5-19 (interface)

Connection Monitoring

- ConnectionListener Interface, page 5-23 (interface)

Cisco SCE Cascade Topology Support

- RedundancyStateListener Interface, page 5-24 (interface)

Operation Results Handling

- OperationException Class, page 5-29 (class)
- SCESubscriberApi (interface)—Contains error codes constants that can be received inside OperationException
- OperationArguments Class, page 5-28 (class)
- OperationResultHandler Interface, page 5-27 (interface)
Package com.scms.common

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

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

This section provides the programming guidelines for the API methods.

Programming with Callback Methods

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

⚠️ Caution

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

This caution notice applies to the following methods:

- LoginPullListener callback methods
- DualLoginPullListener callback methods
- LogoutListener callback methods
- QuotaListenerEx callback methods
- ConnectionListener callback methods
PRPC_SCESubscriberApi Class

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

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

API Construction

The PRPC_SCESubscriber API provides the following constructors:

Syntax

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

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

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

Parameters

The following list presents parameters for the API constructors:

- apiName—Specifies an API name.

Note

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

- sceHost—Can be either an IP address or a reachable hostname.
- scePort—PRPC protocol TCP port by which to connect to the Cisco SCE (default value is 14374).
- autoReconnectInterval—Defines the interval (in milliseconds) after which reconnection is attempted, as follows:
  - If the value is 0 or lesser, the reconnection task is not activated (no auto-reconnect is attempted).
  - If the value is greater than 0 and a connection failure occurs, the reconnection task runs every <autoReconnectInterval>milliseconds.
  - Default value is 1 (no auto-reconnect is attempted).
Chapter 5 Programming with the Cisco SCE Subscriber API

API Construction

Note
To support the auto-reconnect action, call the `connect` method of the API at least once.

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

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

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

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

Listener Setup Operations

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

The following list presents the types of listener setup operations:

- Setting a connection listener. See the “Connection Monitoring” section on page 5-23:
  ```java
  public void setConnectionListener(ConnectionListener listener)
  ```

- Setting a login-pull listener. See the “LoginPullListener Interface Class” section on page 5-15:
  ```java
  public void registerLoginPullListener(LoginPullListener listener)
  ```

- Setting a dual-login-pull listener. See the “DualLoginPullListener Interface Class” section on page 5-17:
  ```java
  public void registerLoginPullListener(DualLoginPullListener listener)
  ```

- Setting a logout listener. See the “LogoutListener Interface Class” section on page 5-19:
  ```java
  public void registerLogoutListener(LogoutListener listener)
  ```

- Setting a quota listener. See the “QuotaListenerEx Interface Class” section on page 5-19:
  ```java
  public void registerQuotaListener(QuotaListener listener)
  ```

- Setting a redundancy state listener. See the “RedundancyStateListener Interface” section on page 5-24:
  ```java
  public void setRedundancyStateListener(RedundancyStateListener listener)
  ```

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

Advanced Setup Operations

To customize the API, you can initialize certain internal properties. Perform the initialization by using the API `init` method.
For settings to take effect, the API must call the init method before the connect method.

You can set the following properties:

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

**Syntax**

The following example presents the syntax for the init method:

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

**Parameters**

- `properties` (java.util.Properties)—Enables setting the properties described in the “Advanced Setup Operations” section on page 5-6:
  - To set the output queue size, use `prpc.client.output.machinemode.recordnum` as a property key.
  - To set the operation timeout, use `com.scms.api.sce.prpc.regularInvocationTimeout` or `com.scms.api.sce.prpc.listenerInvocationTimeout` as a property key.

**Note**

`com.scms.api.sce.prpc.listenerInvocationTimeout` is used for operations that can be invoked from listener callback. To avoid deadlocks, set this timeout to be shorter than `com.scms.api.sce.prpc.regularInvocationTimeout`.

**Customize Properties—Example**

This example shows how to customize properties during initialization:

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

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

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

**Note**

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

**Connecting to Cisco SCE**

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

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

```java
public void connect() throws Exception
```
At any time during the API operation, you can check if the API is connected to the Cisco SCE by using the isConnected() method:

```java
public boolean isConnected()
```

**Note**

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

### getApiVersion

- **Syntax, page 5-8**
- **Description, page 5-8**

**Syntax**

```java
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:

```java
public void disconnect()
```

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

Use a “finally” statement in your main class. For example:

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

### Virtual Link API

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

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

- Create a Virtual Link, page 5-9
- Create Multiple Virtual Links, page 5-10
- Delete a Virtual Link, page 5-11
- Delete Multiple Virtual Links, page 5-11
- Update a Virtual Link, page 5-12
- Update Multiple Virtual Links, page 5-13
- Check Virtual Link Status, page 5-13
- Get Maximum Virtual Links, page 5-14
- Enable the Virtual Link Mode, page 5-15

Create a Virtual Link

- Syntax, page 5-9
- Description, page 5-9
- Parameters, page 5-9
- Return value, page 5-9
- Exception, page 5-10

Syntax

The virtual link creation syntax is as follows:

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

Description

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

Parameters

The create virtual link operation parameter is as follows:

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

Return value

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

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

Create Multiple Virtual Links

- Syntax, page 5-10
- Description, page 5-10
- Parameters, page 5-10
- Return Value, page 5-10
- Exception, page 5-10

Syntax
The multiple virtual links creation syntax is as follows:

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

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

Parameters
The create multiple virtual links operation parameter is as follows:

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

Return Value
Null.

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

- VLinkDisabledException. For more details on the VLinkdisabledException class, see the “VLinkDisabledException Class” section on page 4-20.
- VLinkModeException. For more details on the VLinkModeException class, see the “VLinkModeException Class” section on page 4-20.
Delete a Virtual Link

- Syntax, page 5-11
- Description, page 5-11
- Parameters, page 5-11
- Return Value, page 5-11
- Exception, page 5-11

Syntax
The delete virtual link syntax is as follows:

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

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

Parameters
The delete virtual link operation parameter is as follows:

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

Return Value
Null.

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

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

Delete Multiple Virtual Links

- Syntax, page 5-11
- Description, page 5-11
- Parameters, page 5-12
- Return Value, page 5-12
- Exception, page 5-12

Syntax
The delete multiple virtual links syntax is as follows:

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

Description
This operation is used to delete multiple virtual links in Cisco SCE.
Parameters
The delete multiple virtual link operation parameter is as follows:

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

Return Value
Null.

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

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

Update a Virtual Link

- Syntax, page 5-12
- Description, page 5-12
- Parameters, page 5-12
- Return Value, page 5-12
- Exception, page 5-12

Syntax
The update virtual link syntax is as follows:

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

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

Parameters
The update virtual link operation parameter is as follows:

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

Return Value
Null.

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

- VLinkDisabledException. For more details on the VLinkdisabledException class, see the “VLinkDisabledException Class” section on page 4-20.
- VLinkModeException. For more details on the VLinkModeException class, see the “VLinkModeException Class” section on page 4-20.
Update Multiple Virtual Links

- Syntax, page 5-13
- Description, page 5-13
- Parameters, page 5-13
- Return Value, page 5-13
- Exception, page 5-13

Syntax
The update multiple virtual links syntax is as follows:

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

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

Parameters
The update multiple virtual links operation parameter is as follows:

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

Return Value
Null.

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

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

Check Virtual Link Status

- Syntax, page 5-13
- Description, page 5-14
- Parameters, page 5-14
- Return Value, page 5-14
- Exception, page 5-14

Syntax
The check virtual link status syntax is as follows:

```java
public boolean isVlinkEnabled() throws VLinkModeException;
```
Description
This operation is used to check whether virtual links are enabled or disabled in the Cisco SCE platform.

Parameters
Null.

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

Exception
The following is the exception thrown by the check virtual link status method: VLinkModeException. For more details on the VLinkModeException class, see the “VLinkModeException Class” section on page 4-20.

Get Maximum Virtual Links

- Syntax, page 5-14
- Description, page 5-14
- Parameters, page 5-14
- Return Value, page 5-14
- Exception, page 5-14

Syntax
The get maximum virtual links syntax is as follows:

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

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

Parameters
Null.

Return Value
Maximum number of virtual links.

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

- VLinkDisabledException. For more details on the VLinkdisabledException class, see the “VLinkDisabledException Class” section on page 4-20.
- VLinkModeException. For more details on the VLinkModeException class, see the “VLinkModeException Class” section on page 4-20.
Enable the Virtual Link Mode

- Syntax, page 5-15
- Description, page 5-15
- Parameters, page 5-15
- Return Value, page 5-15
- Exception, page 5-15

Syntax
The enable virtual link mode syntax is as follows:

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

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

Parameters
Null.

Return Value
Null.

Exception
Null.

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

- Login-pull indications
- Logout indications
- Quota indications

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

LoginPullListener Interface Class
The LoginPullListener interface defines a set of callback functions that are used only in Pull Mode. Policy servers that manage the Network ID part of the subscriber provisioning process, and that are intended to work in Pull Mode, register a LoginPullListener. This registration enables the policy server to respond to the login-pull requests from the Cisco SCE and to synchronize the Cisco SCE platform.
Chapter 5 Programming with the Cisco SCE Subscriber API

LoginPullListener Interface Class

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

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

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

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

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

    public void loginPullRequestBulk(NetworkAndSubscriberID_BULK subs)

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

**loginPullRequest Callback Method**

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

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

Figure 5-1 illustrates the loginPullRequest callback method.
Parameters

The loginPullRequest Callback method parameters are as follows:

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

**loginPullRequestBulk Callback Method**

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

**Parameter**

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

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

**GetSubscribersBulkResponse Callback Method**

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

**DualLoginPullListener Interface Class**

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

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

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

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

**Note**

The Cisco SCE Subscriber API supports one DualLoginPullListener at a time. Do not have more than one API that has registered a DualLoginPullListener. If more than one Cisco SCE responds to the same login-pull request, the Cisco SCEs can lose synchronization.
The DualLoginPullListener is an interface that registers a login-pull indication listener. It is defined as follows:

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

**Note**

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

### loginV6PullRequest Callback Method

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

**Figure 5-2 loginV6PullRequest Callback Method**

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

**Parameter**

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

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

The API enables setting a LogoutListener to receive logout indications:

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

The API supports one LogoutListener at a time.

The following sections describe callback functions of the LogoutListener interface:

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

logoutIndication Callback Method

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

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

Parameter

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

logoutBulkIndication Callback Method

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

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

Parameter

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

QuotaListenerEx Interface Class

Note: From Version 3.0.5, the QuotaListener interface is deprecated. Replace it with QuotaListenerEx. For backwards compatibility, the QuotaListener interface still exists, but use the QuotaListenerEx interface when integrating with Version 3.0.5 of the API.

Policy servers that are responsible for the quota management operations in the subscriber provisioning process are able to receive quota-related indications issued by the Cisco SCE platform.
Chapter 5 Programming with the Cisco SCE Subscriber API

QuotaListenerEx Interface Class

The API enables setting the QuotaListener to receive quota indications.

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

Note: The API supports one QuotaListener at a time.

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

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

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

**quotaStatusIndication Callback Method**

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

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

**Parameters**
The `quotaStatusIndication` Callback method parameters are as follows:

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

**quotaStatusBulkIndication Callback Method**

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

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

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

**Parameter**

- `subs` — Contains quota data of the bulk of the subscribers. See the “Quota_BULK Class” section on page 4-17.
quotaBelowThresholdIndication Callback Method

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

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

**Parameters**
The `quotaBelowThresholdIndication` Callback method parameters are as follows:
- `subscriberID`—The unique ID of the subscriber. See the “Subscriber ID” section on page 2-2.
- `quota`—Quota of the subscriber. See the “Subscriber Quota” section on page 4-6.

quotaBelowThresholdBulkIndication Callback Method

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

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

**Parameter**
- `subs`—Contains quota data of the bulk of the subscribers. See the “Quota_BULK Class” section on page 4-17.

quotaDepletedIndication Callback Method

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

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

**Parameters**
The `quotaDepletedIndication` Callback method parameters are as follows:
- `subscriberID`—The unique ID of the subscriber. See the “Subscriber ID” section on page 2-2.
- `quota`—Quota of the subscriber. See the “Subscriber Quota” section on page 4-6.

quotaDepletedBulkIndication Callback Method

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

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

**Parameter**
- `subs`—Contains the names of the subscribers whose quota was depleted. See the “SubscriberID_BULK Class” section on page 4-13.
QuotaListenerEx Interface Class

quotaStateRestore Callback Method

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

\[
\text{public void quotaStateRestore(String subscriberID, Quota quota)}
\]

Parameters

The quotaStateRestore Callback method parameters are as follows:

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

quotaStateBulkRestore Callback Method

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

\[
\text{public void quotaStateBulkRestore(SubscriberID_BULK subs)}
\]

Parameter

- \text{subs}—Contains the names of the subscribers whose quota was depleted. See the “SubscriberID_BULK Class” section on page 4-13.
Connection Monitoring

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

- ConnectionListener Interface, page 5-23
- Disconnect Listener: Example, page 5-23

ConnectionListener Interface

The API enables setting a connection listener.

```
setConnectionListener(ConnectionListener listener)
```

The ConnectionListener interface is defined as follows:

```
public interface ConnectionListener {

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

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

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

Disconnect Listener: Example

This example presents a basic implementation of a disconnect listener that prints a message to stdout and returns.

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

public class MyConnectionListener implements ConnectionListener {

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

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

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

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

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

This section consists of the following topics:
- isRedundancyStatusActive Method, page 5-24
- RedundancyStateListener Interface, page 5-24
- Configuring Cisco SCE to Ignore Cascade Violation Errors, page 5-25

isRedundancyStatusActive Method

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

```java
public boolean isRedundancyStatusActive()
```

The return values from this method indicate the following:
- TRUE—If the Cisco SCE status is active.
- FALSE—in all other cases.

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

RedundancyStateListener Interface

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

```java
setRedundancyStateListener(RedundancyStateListener listener)
```

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

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

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

Note

The Policy Server performs a synchronization procedure on the Cisco SCE that becomes active. This synchronization is similar to the procedure that the Policy Server performs when a connection is established to the Cisco SCE.
The API provides a connection to one Cisco SCE platform for each API instance. Therefore, for cascade setups, two Cisco SCE Subscriber API instances are required.

### Parameters

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

### Configuring Cisco SCE to Ignore Cascade Violation Errors

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

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

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

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

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

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

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

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

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

**Note**

Configure Cisco SCE to ignore cascade violation only if you require backward compatibility with the existing Cisco SCE API. To use the cascade feature fully, monitor and use the Cisco SCE redundancy status.
Configuring Cisco SCE to Define the Quota Buffer Size and Quota Rate

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

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

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

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

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

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

---

**Note**

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

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

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

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

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

OperationResultHandler Interface

This OperationResultHandler interface receives results of operations performed through the API.

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

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

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

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

The following data is returned from the OperationResultHandler interface:

- **result**—Actual result of the operation. Each entry within the array can be one of the following:
  - **NULL**—Indicates success of the operation.
  - **OperationException**—Indicates operation failure. For non-bulk operations, the result array has only one entry. For bulk operations, each entry of the result array corresponds to the relevant entry in the bulk operation.

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

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

**OperationArguments Class**

Use the following method to retrieve the operation name:

```java
public String getOperationName()
```

Use the following method to retrieve the argument names:

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

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

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

**Examples**

The following example implements the OperationResultHandler interface:

```java
public class MyOperationHandler implements OperationResultHandler {
    long successCounter = 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.getMessage());

                // 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]));
                    }
                }
            }
        }
    }
}
```
The following example implements the login operation result handler:

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

**OperationException Class**

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

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

See Appendix A, “List of Error Codes” for details about error codes and their meanings.
Subscriber Provisioning Operations

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

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

This sections consists of these subscriber provisioning operations:

- Login Operation, page 5-30
- loginBulk Operation, page 5-32
- loginPullResponse Operation, page 5-33
- loginPullResponseBulk Operation, page 5-35
- Logout Operation, page 5-35
- logoutBulk Operation, page 5-36
- networkIdUpdate Operation, page 5-37
- networkIdUpdateBulk Operation, page 5-39
- profileUpdate Operation, page 5-39
- profileUpdateBulk Operation, page 5-40
- quotaUpdate Operation, page 5-41
- quotaUpdateBulk Operation, page 5-42
- getQuotaStatus Operation, page 5-43
- getQuotaStatusBulk Operation, page 5-44

Login Operation

- Syntax, page 5-30
- Description, page 5-31
- Parameters, page 5-31
- Error Codes, page 5-32
- Examples, page 5-32

Syntax

The login operation syntax is as follows:

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

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

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

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

Parameters

The login operation parameters are as follows:

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

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

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

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

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

- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_RESOURCE_SHORTAGE
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

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

Examples

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

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

This example adds the IP address 192.168.12.5 overriding previous mappings:

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

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

```java
login(
    "alpha", // subscriber name
    NetworkID.ALL_IPV6_MAPPINGS),
    true, // isMappingAdditive is true
    null, // no policy
    null, // no quota
    null, // no quota
    handler); // operation result handler
```

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

loginBulk Operation

- Syntax, page 5-33
- Description, page 5-33
- Parameters, page 5-33
Syntax

The loginBulk operation syntax is as follows:

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

Description

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

Parameters

The loginBulk Operation parameters are as follows:

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

Error Codes

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

- `ERROR_CODE_FATAL_EXCEPTION`
- `ERROR_CODE_RESOURCE_SHORTAGE`
- `ERROR_CODE_OPERATION_ABORTED`
- `ERROR_CODE_INVALID_PARAMETER`
- `ERROR_CODE_NO_APPLICATION_INSTALLED`

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

loginPullResponse Operation

- Syntax, page 5-33
- Description, page 5-34
- Parameters, page 5-34
- Error Codes, page 5-34

Syntax

The loginPullResponse operation syntax is as follows:

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

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

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

Parameters

The loginPullResponse Operation parameters are as follows:

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

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

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

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

Error Codes

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

- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_RESOURCE_SHORTAGE
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

For a description of error codes, see Appendix A, “List of Error Codes.”
loginPullResponseBulk Operation

- Syntax, page 5-35
- Description, page 5-35
- Parameters, page 5-35
- Error Codes, page 5-35

Syntax

The loginPullResponseBulk operation syntax is as follows:

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

Description

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

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

Parameters

The loginPullResponseBulk operation parameters are as follows:

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

Error Codes

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

- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_RESOURCE_SHORTAGE
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

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

Logout Operation

- Syntax, page 5-36
- Description, page 5-36
- Parameters, page 5-36
- Error Codes, page 5-36
The Logout operation is as follows:

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

**Description**

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

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

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

**Parameters**

The logout operation parameters are as follows:

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

**Error Codes**

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

- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED

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

### logoutBulk Operation

- Syntax, page 5-36
- Description, page 5-37
- Parameters, page 5-37
- Error Codes, page 5-37

**Syntax**

The logoutBulk operation syntax is as follows:

```java
void logoutBulk(NetworkAndSubscriberID_BULK subsBulk,
```
networkIdUpdate Operation

Description

This operation applies the logic described in the logout operation for each subscriber in the bulk. For a description of the relevant events, see the “Logout Events” section on page 3-4.

Parameters

The logoutBulk operation parameters are as follows:
- subsBulk—See the “NetworkAndSubscriberID_BULK Class” section on page 4-14.
- handler—Result handler for this operation. See the “Result Handling” section on page 5-27 for a description of the OperationResultHandler interface.

Error Codes

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

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

Syntax

The networkIdUpdate operation syntax is as follows:

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

Description

This operation adds or replaces an existing subscriber network ID.

Note

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

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

The networkIdUpdate operation parameters are as follows:

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

Error Codes

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

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

For a description of error codes, see Appendix A, “List of Error Codes”. 
networkIdUpdateBulk Operation

- Syntax, page 5-39
- Description, page 5-39
- Parameters, page 5-39
- Error Codes, page 5-39

Syntax

The networkIdUpdateBulk operation syntax is as follows:

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

Description

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

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

Parameters

The networkIdUpdateBulk operation parameters are as follows:

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

Error Codes

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

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

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

profileUpdate Operation

- Syntax, page 5-40
- Description, page 5-40
- Parameters, page 5-40
ProfileUpdateBulk Operation

Syntax

The profileUpdate operation syntax is as follows:

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

Description

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

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

Parameters

The profileUpdate operation parameters are as follows:

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

Error Codes

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

- ERROR_CODE_SUBSCRIBER_NOT_EXIST
- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

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

profileUpdateBulk Operation

- Syntax, page 5-41
- Description, page 5-41
- Parameters, page 5-41
- Error Codes, page 5-41
Syntax

The profileUpdateBulk operation syntax is as follows:

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

Description

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

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

Parameters

The profileUpdateBulk operation parameters are as follows:

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

Error Codes

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

- `ERROR_CODE_SUBSCRIBER_NOT_EXIST`
- `ERROR_CODE_FATAL_EXCEPTION`
- `ERROR_CODE_OPERATION_ABORTED`
- `ERROR_CODE_INVALID_PARAMETER`
- `ERROR_CODE_NO_APPLICATION_INSTALLED`

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

quotaUpdate Operation

- Syntax, page 5-41
- Description, page 5-42
- Parameters, page 5-42
- Error Codes, page 5-42

Syntax

The quotaUpdate operation syntax is as follows:

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

This operation updates the subscriber quota.
For a description of the relevant events, see the “Quota Update” section on page 3-6.

Parameters

The quotaUpdate operation parameters are as follows:

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

Error Codes

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

- ERROR_CODE_SUBSCRIBER_NOT_EXIST
- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

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

quotaUpdateBulk Operation

- Syntax, page 5-42
- Description, page 5-42
- Parameters, page 5-43
- Error Codes, page 5-43

Syntax

The quotaUpdateBulk operation syntax is as follows:

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

Description

This operation applies the logic of the quotaUpdate operation on each subscriber in the bulk.
For a description of the relevant events, see the “Quota Update” section on page 3-6.
Parameters

The quotaUpdateBulk operation parameters are as follows:

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

Error Codes

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

- `ERROR_CODE_SUBSCRIBER_NOT_EXIST`
- `ERROR_CODE_FATAL_EXCEPTION`
- `ERROR_CODE_OPERATION_ABORTED`
- `ERROR_CODE_INVALID_PARAMETER`
- `ERROR_CODE_NO_APPLICATION_INSTALLED`

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

goingQuotaStatus Operation

- Syntax, page 5-43
- Description, page 5-43
- Parameters, page 5-43
- Error Codes, page 5-44

Syntax

The goingQuotaStatus operation syntax is as follows:

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

Description

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

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

Parameters

The goingQuotaStatus operation parameters are as follows:

- `subscriberID`—Unique ID of the subscriber. For a description of the subscriber ID format, see the “Subscriber ID” section on page 2-2.
getQuotaStatusBulk Operation

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

Error Codes

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

- ERROR_CODE_SUBSCRIBER_NOT_EXIST
- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

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

getQuotaStatusBulk Operation

- Syntax, page 5-44
- Description, page 5-44
- Parameters, page 5-44
- Error Codes, page 5-45

Syntax

The getQuotaStatusBulk operation syntax is as follows:

```java
void getQuotaStatusBulk(Quota_BULK subsBulk,
                           OperationResultHandler handler) throws Exception
```

Description

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

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

Parameters

The getQuotaStatusBulk operation parameters are as follows:

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

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

- ERROR_CODE_SUBSCRIBER_NOT_EXIST
- ERROR_CODE_FATAL_EXCEPTION
- ERROR_CODE_OPERATION_ABORTED
- ERROR_CODE_INVALID_PARAMETER
- ERROR_CODE_NO_APPLICATION_INSTALLED

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

Cisco SCE-API Synchronization

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

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

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

Caution

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

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

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

Figure 5-3 illustrates the following procedure:

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

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

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

- Syntax, page 5-47
- Description, page 5-47
- Parameter, page 5-47

Syntax

void synchronizePushStart(OperationResultHandler handler)

Description

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

Parameter

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

synchronizePushEnd

- Syntax, page 5-47
- Description, page 5-47
- Parameters, page 5-47

Syntax

void synchronizePushEnd(boolean success, OperationResultHandler handler)

Description

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

Parameters

The synchronizePushEnd parameters are as follows:

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

Pull Mode Synchronization Procedure

Figure 5-4 illustrates the following procedure:

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

Use the following algorithm template when planning the synchronization procedure:

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

- If <SubscriberID, IP address> exists in the Policy Server database, send a policy profile and networkID update to the Cisco SCE.
- Otherwise, send a logout with the Subscriber IP to the Cisco SCE.

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

**Figure 5-4 Pull Mode Synchronization**

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

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

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

- Syntax, page 5-49
- Description, page 5-49
- Parameter, page 5-49

**Syntax**
void synchronizePullStart(OperationResultHandler handler)

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

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

synchronizePullEnd

- Syntax, page 5-49
- Description, page 5-49
- Parameters, page 5-49

**Syntax**
void synchronizePullEnd(boolean success, OperationResultHandler handler)

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

**Parameters**
The synchronizePullEnd parameters are as follows:
- handler—Result handler for this operation. See the “Result Handling” section on page 5-27 for a description of the OperationResultHandler interface.
- success—Indicates to the Cisco SCE that the synchronization was successful.

getSubscribersBulk

- Syntax, page 5-49
- Description, page 5-50
- Parameters, page 5-50

**Syntax**
void getSubscribersBulk(int bulkSize,
                        SubscribersBulkResponseIterator iterator,
                        OperationResultHandler handler)
Description
Use this operation in the Pull mode synchronization process to retrieve a bulk of subscribers that the Cisco SCE is currently managing (see the “Pull Mode Synchronization Procedure” section on page 5-47).

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

Figure 5-5 getSubscribersBulk Description

Parameters
The getSubscribersBulk parameters are as follows:

- **bulkSize**—Size of the bulk to retrieve. Maximum bulk size is limited to 100 entries.
- **iterator**—Iterator of the subscribers at the Cisco SCE side. This iterator is received in the getSubscribersBulkResponse indication and it is passed to the next call to the getSubscribersBulk method. When calling the getSubscribersBulk method for the first time, use null as an iterator (using null indicates that you want to start from the beginning).
- **handler**—Result handler for this operation. See the “Result Handling” section on page 5-27 for a description of the OperationResultHandler interface.

getSubscribersBulkIPv6

- Syntax, page 5-50
- Description, page 5-50
- Parameters, page 5-51

Syntax

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

Description
Use this operation in the pull mode synchronization process to retrieve the bulk of IPv6 subscribers that Cisco SCE is currently managing (see the “Pull Mode Synchronization Procedure” section on page 5-47).
On receiving a getSubscribersBulkIPv6 request, Cisco SCE asynchronously issues the getSubscribersBulkResponse indication containing the subscriber IDs and the corresponding Network IDs (see the “DualLoginPullListener Interface Class” section on page 5-17). One of the indication parameters is the isLastBulk flag that is checked before moving to the next call to the getSubscribersBulkIPv6. The isLastBulk flag is specified by getSubscribersBulkIPv6 in the interface SCESubscriberApi.

### Parameters

- **bulkSize**—Number of entries in the bulk. The maximum number of entries is 100.
- **iterator**—Iterator of the subscribers on the Cisco SCE side. This iterator is received in getSubscribersBulkIndication and is passed to the next call to getSubscribersBulkIPv6 method. When calling the getSubscribersBulkIPv6 method for the first time, use null as an iterator. Using null indicates that you want to start from the beginning.
- **handler**—Result handler for getSubscribersBulkIPv6 operation. See the “Result Handling” section on page 5-27 for a description of the OperationResultHandler interface.

### Note

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

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

Implementing High Availability

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

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

Note

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

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

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

Note

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

This section presents several API code examples:

- Login and Logout, page 5-53
- login-pull Request and login-pull Response, page 5-59
- iVirtual Link Operations, page 5-70

Login and Logout

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

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

```java
// 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("success "+count);
                }
            }else // error - print every error
            {
                // failure
                count++;
                // Extract error details
                OperationException ex = (OperationException)result[index];

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

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

The following class contains a basic LogoutListener implementation that counts the number of received logout indications:

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

class MyLogoutListener implements LogoutListener {
    long count = 0;

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

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

    synchronized long getCounter() {
        return count;
    }

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

    public void logoutBulkIndication(SubscriberID_BULK subs) {
        increaseCounter(subs.getSize());
    }
}
```
The following class contains the main method:

```java
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
```
Chapter 5 Programming with the Cisco SCE Subscriber API

Login and Logout

```java
api.unregisterLogoutListener
api.disconnect();
```

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

The following example logs in a predefined number of subscribers to Cisco SCE, and then logs them out. Because this example uses autoreconnect support, it does not define a connection listener. The following example also contains a sample implementation of a result handler that counts success and failure results:

```java
// 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) {
            }
        }
    }
}
```
The following class contains a basic LogoutListener implementation that counts the number of received logout indications:

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

class MyLogoutListener implements LogoutListener {
    long count = 0;

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

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

    synchronized long getCounter() {
        return count;
    }

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

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

The following class contains the main method:

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

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

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

        try {
            // instantiate operation result handler
            // we will use one handler for all operations
            MyOperationResultHandler resultHandler = new MyOperationResultHandler();
```
// instantiate logout listener
MyLogoutListener listener = new MyLogoutListener();
// register to logout indications
api.registerLogoutListener(listener);
// connect to the SCE
api.connect();
// login
System.out.println("login of "+numSubscribersToLogin+" subscribers");
PolicyProfile pp = new PolicyProfile(new String[]{"packageId=1","monitor=1"});
for (int i=0; i<numSubscribersToLogin; i++)
{
    api.login("sub"+i,
    new NetworkID(getIPv6Mappings(i), // generate ip
    NetworkID.ALL_IPV6_MAPPINGS),
    true, // additive flag
    pp, // policy
    null, // no quota
    null, no esa
    resultHandler);
}
// wait for subscribers to log in
resultHandler.waitForLastResult(numSubscribersToLogin);
// logout all subscribers
System.out.println("logout of "+numSubscribersToLogin+" subscribers");
for (int i=0; i<numSubscribersToLogin; i++)
{
    NetworkID nid = new NetworkID(getIPv6Mappings(i),
    NetworkID.ALL_IPV6_MAPPINGS);
    api.logout("sub"+i,nid,resultHandler);
}
// wait for all subscribers to be logged out -
// but this time use
// logout listener to count the results
listener.waitForLastResult(numSubscribersToLogin);
}
finally
{
    api.unregisterLogoutListener
    api.disconnect();
}

// 'automatic' mapping generator for the sample program
private static String[] getIPv6Mappings(int i)
{
    return new String[]{"abcd:" +((int)i/65536)%256 + ":": +
    ((int)(i/256))%256 + ":": + (i%256) + ":/64" };
login-pull Request and login-pull Response

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

```java
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_SCESSubscriberApi api = null;

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

    public MyListener(PRPC_SCESSubscriberApi 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(),
```
public void loginPullRequestBulk(NetworkAndSubscriberID BULK subs)
{
    try
    {
        increasePullCounter(subs.getSize());
        System.out.println("Got pull request" + getPullCounter());
        // Answer with pull response in bulk form
        PolicyProfile pp = new PolicyProfile(new String[] { "packageId=1", "monitor=1" });
        LoginPullResponse_BULK responseBulk = new LoginPullResponse_BULK();
        Iterator subsIterator = subs.getIterator();
        // iterate of the received bulk (IPs and anonymous IDs)
        // and build a response bulk
        int count = 0;
        while (subsIterator.hasNext())
        {
            // retrieve subscriber name – for example from your
            // policy server database
            // In this example we use fixed names based on the
            // subscribers counter
            String subName = "sub_" + count;
            SubscriberData sub = (SubscriberData) subsIterator.next();
            // Extract subscriber mappings from the bulk and
            // construct a new NetworkID based on those mappings
            NetworkID subNetId = new NetworkID(sub.getMappings(),
                NetworkID.ALL_IP_MAPPINGS);
            responseBulk.addEntry(sub.getAnonymousSubscriberID(),
                subName,
                subNetId,
                true,
                pp,
                null);
            count++;
        }
        // use the bulk constructed above in the bulk response
        // use handler from the previous example
        api.loginPullBulkResponse(responseBulk, h);
    }
    catch (Exception ex)
    {
        System.out.println(ex.getMessage());
    }
}

public void getSubscribersBulkResponse(
    NetworkAndSubscriberID BULK subs,
    SubscruberBulkResponseIterator iterator)
synchronized void increaseLogoutCounter(long value) {
    logoutCount = logoutCount + value;
}

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

synchronized long getPullCounter() {
    return pullCount;
}

synchronized long getLogoutCounter() {
    return logoutCount;
}

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

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

The following class contains the main method:

```java
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;
```
import java.util.Iterator;
import com.scms.api.sce.*;
import com.scms.common.*;

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

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

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

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

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

            // wait for login-pull requests from the SCE
            // they will be issued if you have traffic for unknown
            // subscribers at the SCE
            System.out.println("Waiting for pull requests for "+
                              numSubscribersToLogin+
                              " subscribers");

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

            // logout all subscribers
            System.out.println("logout of "+numSubscribersToLogin+
                               " subscribers");
            for (int i=0; i< numSubscribersToLogin; i++) {
                api.logout("sub"+i,null,handler);
            }

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

The following code fragment demonstrates a login-pull request and login-pull response manipulation for IPv6. This class is a sample implementation of the listener for the logout and login pull indications.
api = new PRPC_SCESubscriberApi("myAPI","1.1.1.1",5000);
// construct an operation result handler (from the
// previous example
MyOperationResultHandler handler = new MyOperationResultHandler();
// instantiate logout and login-pull listener
MyListener listener = new MyListener(api);
try
{
    // register to logout indications
    api.registerLogoutListener(listener);
    api.registerLoginPullListener(listener);
    // connect to the SCE
    api.connect();
    // wait for login-pull requests from the SCE
    // they will be issued if you have traffic for unknown
    // subscribers at the SCE
    System.out.println("Waiting for pull requests for "+
    numSubscribersToLogin+" subscribers");
    // wait for all subscribers to be logged in
    listener.waitForPullResult(numSubscribersToLogin);
    //logout all subscribers
    System.out.println("logout of "+numSubscribersToLogin+" subscribers");
    for (int i=0; i<numSubscribersToLogin; i++)
    {
        api.logout("sub"+i,null,handler);
    }
    // wait for all subscribers to be logged out
    listener.waitForLogoutResult(numSubscribersToLogin);
}
finally
{
    api.unregisterLoginPullListener();
    api.unregisterLogoutListener();
    api.disconnect();
}
}

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

import java.util.Iterator;
// result handler from the previous example
import MyOperationResultHandler;
import com.scms.api.sce.*;
import com.scms.common.*;
class MyListener implements DualLoginPullListener, LogoutListener
{
    // indications counters
    long logoutCount = 0;
    long pullCount=0;
    // api instance - used to send login-pull responses to the SCE
    PRPC_SCESubscriberApi api = null;
    // construct operation handler -
    // from previous (Login and Logout) example
    MyOperationResultHandler h = new MyOperationResultHandler();
    public MyListener(PRPC_SCESubscriberApi api)
    {
        this.api = api;
    }
    // Increase logout counter
    public void logoutIndication(String subscriberID)
    {
// Increase logout counter
public void logoutBulkIndication(SubscriberID BULK subs) {
    System.out.println("Got logout notification");
    increaseLogoutCounter(subs.getSize());
}

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

public void loginV6PullRequest(String anonymousSubscriberID, long arg1, long arg2) {
    try {
        increasePullCounter(1);
        System.out.println("MyListener - Got IPv6 pull request " + getPullCounter());
        // prepare policy
        PolicyProfile pp = new PolicyProfile(new String[] {"packageId=1","monitor=1"});
        // Answer with pull response
        // retrieve subscriber name - for example from your
        // policy server database
        // In this example we use fixed names based on the
        // subscribers counter
        String ipv6Address = IPV6Utilities.getIPv6As64BitsString(IPV6Utilities.getBigIntIPV6(arg1, arg2));
        NetworkID networkID = new NetworkID(new String[] {ipv6Address + "/64"}, new short[]{NetworkID.TYPE_IPV6});
        api.loginPullResponse("sub" + getPullCounter(), anonymousSubscriberID, networkID, pp, // policy
                              null, // no quota
                              h); // handler from previous example
    } catch (Exception ex) {
        System.out.println(ex.getMessage());
    }
}

public void loginPullRequestBulk(NetworkAndSubscriberID BULK subs)
try {
    increasePullCounter(subs.getSize());
    System.out.println("Got pull request" + getPullCounter());
    // Answer with pull response in bulk form
    PolicyProfile pp = new PolicyProfile(new String[]{"packageId=1","monitor=1"});
    LoginPullResponse_BULK responseBulk = new LoginPullResponse_BULK();
    Iterator subsIterator = subs.getIterator();
    // Iterate of the received bulk (IPs and anonymous IDs)
    // and build a response bulk
    int count=0;
    while(subsIterator.hasNext()) {
        // Retrieve subscriber name – for example from your
        // policy server database
        // In this example we use fixed names based on the
        // subscribers counter
        String subName = "sub_"+count;
        SubscriberData sub = (SubscriberData)subsIterator.next();
        // Extract subscriber mappings from the bulk and
        // construct a new NetworkID based on those mappings
        NetworkID subNetId = new NetworkID(sub.getMappings(),
                                             NetworkID.ALL_IP_MAPPINGS);
        responseBulk.addEntry(sub.getAnonymousSubscriberID(),
                              subName,
                              subNetId,
                              true,
                              pp,
                              null);
        count++;
    }
    // Use the bulk constructed above in the bulk response
    // Use handler from the previous example
    api.loginPullBulkResponse(responseBulk,h);
} catch (Exception ex) {
    System.out.println(ex.getMessage());
}

public void getSubscribersBulkResponse(
    NetworkAndSubscriberID_BULK subs,
    SubscriberBulkResponseIterator 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 (InterruptedException ie) {
            ie.printStackTrace();
        }
    }
}

The following class demonstrates the pull synchronization for IPv6:

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

public class myDualLoginPullListener implements DualLoginPullListener, LogoutListener {
    // indications counters
    public long logoutCount = 0;
    public long pullCount = 0;
    public long myNumOfBulkInPullSynchProcess = 0;
    public boolean synchEnded = false;
    // api instance - used to send login-pull responses to the SCE
    PRPC_SCESubscriberApi api = null;
    // construct operation handler -
    // from previous (Login and Logout) example
    MyOperationResultHandler myResultHandler = new MyOperationResultHandler();
    public myDualLoginPullListener(PRPC_SCESubscriberApi api) {
        this.api = api;
    }
    // Increase logout counter
    public void logoutIndication(String subscriberID) {
        increaseLogoutCounter(1);
        System.out.println("Got logout notification " + getLogoutCounter());
    }
    // Increase logout counter
    public void logoutBulkIndication(SubscriberID_BULK subs) {
        System.out.println("Got logout notification");
        increaseLogoutCounter(subs.getSize());
    }
    public void loginPullRequest (String anonymousSubscriberID, NetworkID networkID) {
        try {
            increasePullCounter(1);
            System.out.println("Got pull request" + getPullCounter());
        // prepare policy
    }
```
login-pull Request and login-pull Response

PolicyProfile pp = new PolicyProfile(new String[] {"packageId=1","monitor=1"}); // Answer with pull response
// retrieve subscriber name – for example from your
// policy server database
// In this example we use fixed names based on the
// subscribers counter
api.loginPullResponse(anonymousSubscriberID, "sub"+getPullCounter(),
networkID, pp, // policy
null, // no quota
myResultHandler); // handler from previous example
} catch (Exception ex) {
    System.out.println(ex.getMessage());
}

public void loginV6PullRequest(String anonymousSubscriberID, long arg1, long arg2) {
  try {
    increasePullCounter(1);
    System.out.println("MyListener - Got IPv6 pull request " + getPullCounter());
    // prepare policy
    PolicyProfile pp = new PolicyProfile(new String[] {"packageId=1","monitor=1"}); // Answer with pull response
    // retrieve subscriber name – for example from your
    // policy server database
    // In this example we use fixed names based on the
    // subscribers counter
    String ipv6Address = IPV6Utilities.getIPv6As64BitsString(IPV6Utilities.getBigIntIPV6(arg1, arg2));
    NetworkID networkID = new NetworkID(new String[] {ipv6Address+"/64"}, new
    short[] {NetworkID.TYPE_IPV6});
    api.loginPullResponse("sub"+getPullCounter(),
                        anonymousSubscriberID,
                        networkID,
                        pp, // policy
                        null, // no quota
                        myResultHandler); // handler from previous example
  } catch (Exception ex) {
    System.out.println(ex.getMessage());
  }
}

public void loginPullRequestBulk(NetworkAndSubscriberID_BULK subs) {
  // not implemented in this example
}

public void getSubscribersBulkResponse(NetworkAndSubscriberID_BULK subs,
                                        SubscribersBulkResponseIterator iterator) {
  Iterator it = subs.iterator();
  PolicyProfile pp = new PolicyProfile(new String[] {"packageId=1","monitor=1"});
  while(it.hasNext()) {
    SubscriberData sub = (SubscriberData)it.next();
  }
}
String subName = sub.getSubscriberID();
if(!subName.equals("N/A"))
{
    try
    {
        api.policyProfileUpdate(sub.getSubscriberID(), pp,
        myResultHandler);
        System.out.println("Policy Updated for " + sub.getSubscriberID());
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
    else
    {
        System.out.println("Party N/A in the Bulk !!!!!");
    }
}
if (iterator != null)
{
    try
    {
        myNumOfBulkInPullSynchProcess++;
        api.getSubscribersBulkIPv6(subs.getSize(), iterator,
        myResultHandler);
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}
else // null indicates all parties were sent to the api
{
    synchEnded = true;
}
}synchronized void increaseLogoutCounter(long value)
{
    logoutCount = logoutCount + value;
}synchronized void increasePullCounter(long value)
{
    pullCount = pullCount + value;
}synchronized long getPullCounter()
{
    return pullCount;
}synchronized long getLogoutCounter()
{
    return logoutCount;
}//waits for result number 'last result' to arrive
public synchronized void waitForPullResult(int lastResult)
{
    while (pullCount<lastResult)
The following class contains the main method of the pull synchronization:

```java
import java.util.Iterator;
import com.scms.api.sce.*;
import com.scms.common.
public class pullSynch {
    static PRPC_SCESubscriberApi api = null;
    // This sample program waits for pull requests from the SCE
    // and answers to them with pull response
    // The program exists after all 500 were logged in
    public static void main(String args[]) throws Exception {
        int numSubscribersToLogin = 5;
        int bulkSize = 5;
        SubscribersBulkResponseIterator myIt = null;
        // instantiate an API with reconnect interval of 5 seconds
        api = new PRPC_SCESubscriberApi("myAPI","10.78.241.51",5000);
        // construct an operation result handler (from the
        // previous example
        MyOperationResultHandler handler = new MyOperationResultHandler();
        // instantiate logout and login-pull listener
        myDualLoginPullListener listener = new myDualLoginPullListener(api);
        try {
            // register to logout indications
            api.registerLogoutListener(listener);
            api.registerLoginPullListener(listener);
            // connect to the SCE
```
### iVirtual Link Operations

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

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

public class VLinkAPI extends PRPC_SCESubscriberApi {
    static PRPC_SCESubscriberApi Api = null;

    public static void main(String args[]) throws Exception {
        try {
            VLink v1 = new VLink(100, "vlink1", VLink.UPSTREAM, 0, 2432, 20, -1);
            VLink v2 = new VLink(102, "vlink2", VLink.DOWNSTREAM, 0, 2432, 16, -1);
            VLink v3 = new VLink(105, "vlink3", VLink.UPSTREAM, 0, 2432, 16, -1);
        }
    }
}
```
VLink v4 = new VLink(105, "vlink3", VLink.UPSTREAM, 0, 2432, 18);

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

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

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

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

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

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

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

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

//connect to SCE
Api.connect();

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

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

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

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

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

v1.setVlinkCIR(20);

v2.setVlinkPIR(30);

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

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

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

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

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

catch (Exception e) {
    e.printStackTrace();
iVirtual Link Operations

```java
}

finally {
    Api.disconnect();
}

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

Published: August 30, 2014,

Introduction

This chapter describes application programming interface (API) logging for troubleshooting integration with the API. API logging enables you to monitor the operations being called, including the received parameters, at the API client side and at the Cisco Service Control Engine (SCE) side.

- Cisco SCE Logging, page 6-2
- API Client Logging, page 6-6
The Cisco SCE platform enables you to log all operations called by the policy server to a Cisco SCE user-log file.

This section consists of these subsections:

- **Default Log Messages, page 6-2**
- **Subscriber Operation Log Messages, page 6-3**

### Default Log Messages

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

For connect operation:

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

For disconnect operation:

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

For registerLoginPullListener operation:

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

For unregisterLoginPullListener operation:

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

For registerLogoutListener operation:

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

For unregisterLogoutListener operation:

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

For registerQuotaListener operation:

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

For unregisterQuotaListener operation:

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

For synchronizePushStart operation:

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

For synchronizePushEnd operation:

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

For synchronizePullStart operation:

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

For synchronizePullEnd operation:

```plaintext
<client-name>- synchronize Pull End
```
Subscriber Operation Log Messages

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

To enable logging, issue the following CLI command on the Cisco SCE platform:

```
(config)#> management-agent sce-api logging
```

To view the user-log file, issue the following CLI command on the Cisco SCE platform:

```
#> logger get user-log file-name <FILE NAME>
```

**Note**

Enabling logging degrades performance. Therefore, use logging only for troubleshooting.

To disable logging, issue the following CLI command on the Cisco SCE platform:

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

To detect whether logging is enabled, issue the following CLI command on the Cisco SCE platform:

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

When the logging parameter is enabled, for the following subsequent 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>

The following messages are issued for 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>

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

The following messages are issued for 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 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 enables the logging of every activated operation in the apilog file located under `${user.home}` directory. Configure the logging parameters by using the Log4J properties files. To enable logging, ensure that this file is in the applications CLASSPATH. This file is read at startup of the application; therefore, after changing it, restart the application.

The following is the content of the log4.properties file:

```properties
# 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
%i%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 debug logging, uncomment the last line in the file. By default, logging is performed to the standard output. To direct 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/quotas buckets list>

For the following bulk operations:
  • loginBulk operation
  • networkIDUpdateBulk operation
  • logoutBulk operation
  • quotaUpdateBulk operation
  • loginPullBulkResponse operation
  • profileUpdateBulk operation
  • getQuotaStatusBulkRequest operation
  • getSubscribersBulk operation
  • getSubscribersBulkIPv6 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(LoginPullListener) operation:
  registerLoginPullListener operation was called
• For registerLoginPullListener(DualLoginPullListener) operation:
  registerDualLoginPullListener operation was called

• For unregisterLoginPullListener operation:
  unregisterLoginPullListener operation was called
  or
  unregisterDualLoginPullListener operation was called

• For registerLogoutListener operation:
  registerLogoutListener operation was called

\[\textbf{Note}\]
Only one listener can be registered at a time. If you try to register a second listener, the following message is displayed:

Already another login pull listener <type of the login pull listener that was registered> has been registered. Please unregister it.

• 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 following message is issued for the DualLoginPullListener listener callback methods:

• For loginV6PullRequest:
  loginV6PullRequest operation was called with parameters:
  anonymous subscriber ID - <anonymous subscriber ID>
  mappings: higher octet value=<higher octet value> and lower octet value=<lower octet value>
  mapping types - TYPE_IPV6

The following 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>
List of Error Codes

Published: August 30, 2014,

Introduction

This appendix lists the error codes that the application programming interface (API) returns.
List of Error Codes

Use error codes to interpret the actual error for which an OperationException is returned. The error code is extracted by using the getErrorCode method.

Table A-1 lists and describes the error codes.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR_CODE_ARRAY_ACCESS</td>
<td>Internal error.</td>
</tr>
<tr>
<td>ERROR_CODE_ATTRIBUTE_NOT_FOUND</td>
<td>Internal error.</td>
</tr>
<tr>
<td>ERROR_CODE_CLASS_CAST</td>
<td>Internal error.</td>
</tr>
<tr>
<td>ERROR_CODE_CLASS_NOT_FOUND</td>
<td>Internal error.</td>
</tr>
<tr>
<td>ERROR_CODE_CLIENT_INTERNAL_ERROR</td>
<td>Internal error.</td>
</tr>
<tr>
<td>ERROR_CODE_CLIENT_OUT_OF_THREADS</td>
<td>Internal error.</td>
</tr>
<tr>
<td>ERROR_CODE_FATAL_EXCEPTION</td>
<td>Too many errors occurred at the Cisco SCE when trying to perform the operation.</td>
</tr>
<tr>
<td>ERROR_CODE_ILLEGAL_STATE</td>
<td>Internal error.</td>
</tr>
<tr>
<td>ERROR_CODE_INVALID_PARAMETER</td>
<td>One of the arguments provided to the method is illegal.</td>
</tr>
<tr>
<td>ERROR_CODE_NO_APPLICATION_INSTALLED</td>
<td>Application required to execute the operation is not installed.</td>
</tr>
<tr>
<td>ERROR_CODE_NULL_POINTER</td>
<td>Internal error.</td>
</tr>
<tr>
<td>ERROR_CODE_OBJECT_NOT_FOUND</td>
<td>Internal error.</td>
</tr>
<tr>
<td>ERROR_CODE_OPERATION_ABORTED</td>
<td>Internal error.</td>
</tr>
<tr>
<td>ERROR_CODE_OPERATION_NOT_FOUND</td>
<td>Internal error.</td>
</tr>
<tr>
<td>ERROR_CODE_OUT_OF_MEMORY</td>
<td>Internal error.</td>
</tr>
<tr>
<td>ERROR_CODE_RESOURCE_SHORTAGE</td>
<td>Internal error.</td>
</tr>
<tr>
<td>ERROR_CODE_RUNTIME</td>
<td>Internal error.</td>
</tr>
<tr>
<td>ERROR_CODE_SUBSCRIBER_ALREADY_EXISTS</td>
<td>The subscriber on which the operation was performed exists in the Cisco SCE already.</td>
</tr>
<tr>
<td>ERROR_CODE_SUBSCRIBER_DOES_NOT_EXIST</td>
<td>The subscriber on which the operation is performed does not exist in the Cisco SCE.</td>
</tr>
<tr>
<td>ERROR_CODE_UNKNOWN</td>
<td>Internal error.</td>
</tr>
</tbody>
</table>