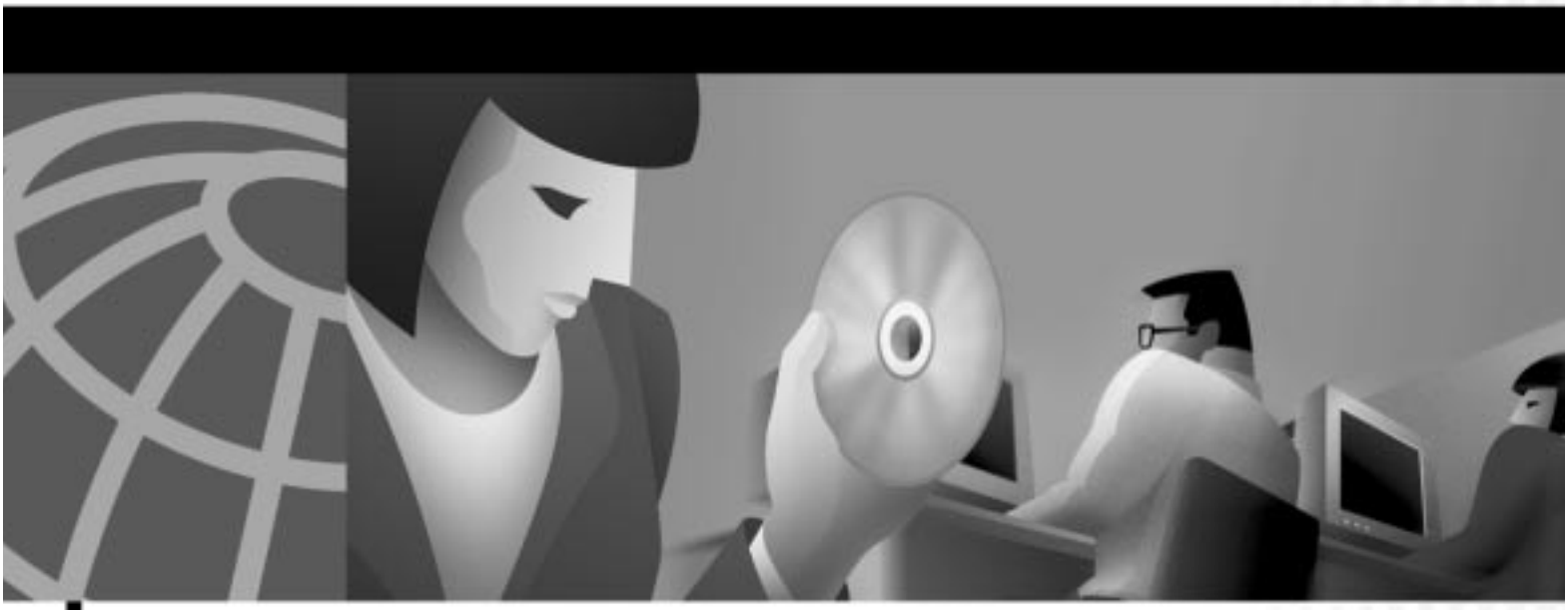


# Broadband Provisioning Registrar Developer's Guide

Software Release 2.0

Corporate Headquarters
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
http://www.cisco.com
Tel: 408 526-4000
800 553-NETS (6387)
Fax: 408 526-4100

Customer Order Number: DOC-7812374 =
Text Part Number: 78-12374-01

*Broadband Provisioning Registrar Developer's Guide*

# CONTENTS

# FIGURES

# Preface

This section describes who should read this guide, how it is organized, and the document conventions used in it.

## Who Should Read This Guide

This guide is designed for those who want to use the Broadband Provisioning Registrar (BPR) application programming interface (API) to integrate their existing operations support system (OSS) with the provisioning capabilities of BPR.

To use this guide, you should have the following knowledge and experience:

- Java program development
- Administration of service provider networks

## How This Guide Is Organized

This guide describes how to use the BPR provisioning API to integrate OSS and billing systems with BPR. This guide provides conceptual information about integration and provisioning, use cases that illustrate the use of API calls, and information about the interfaces, classes, and methods of BPR. The major sections of this guide are:

| | | |
|---|---|---|
| Chapter 1 | Getting Started | Explains provisioning clients and the provisioning process. |
| Chapter 2 | Provisioning API Use Cases | Presents the use cases. |
| Appendix A | Provisioning API Reference | Provides a reference to BPR methods. |
| Appendix B | Deprecated Methods | Provides a list of methods deprecated since CSRC Device Provisioning Registrar. |

# Document Conventions

This guide uses these conventions:

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

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

# Related Documentation

Refer to these Cisco manuals for additional information:

- *Broadband Provisioning Registrar Version 2.0 Release Notes*
- *Broadband Provisioning Registrar Installation Guide*
- *Broadband Provisioning Registrar Administrator's Guide*
- *Network Registrar User's Guide*
- *Network Registrar CLI Reference Guide*
- *Site Preparation and Safety Guide*

# Obtaining Documentation

The following sections explain how to obtain documentation from Cisco Systems.

## World Wide Web

You can access the most current Cisco documentation on the World Wide Web at the following URL:

http://www.cisco.com

Translated documentation is available at the following URL:

http://www.cisco.com/public/countries_languages.shtml

## Documentation CD-ROM

Cisco documentation and additional literature are available in a Cisco Documentation CD-ROM package, which is shipped with your product. The Documentation CD-ROM is updated monthly and may be more current than printed documentation. The CD-ROM package is available as a single unit or through an annual subscription.

## Ordering Documentation

Cisco documentation is available in the following ways:

- Registered Cisco Direct Customers can order Cisco product documentation from the Networking Products MarketPlace:

  http://www.cisco.com/cgi-bin/order/order_root.pl

- Registered Cisco.com users can order the Documentation CD-ROM through the online Subscription Store:

  http://www.cisco.com/go/subscription

- Nonregistered Cisco.com users can order documentation through a local account representative by calling Cisco corporate headquarters (California, USA) at 408 526-7208 or, elsewhere in North America, by calling 800 553-NETS (6387).

## Documentation Feedback

If you are reading Cisco product documentation on Cisco.com, you can submit technical comments electronically. Click **Feedback** at the top of the Cisco Documentation home page. After you complete the form, print it out and fax it to Cisco at 408 527-0730.

You can e-mail your comments to bug-doc@cisco.com.

To submit your comments by mail, use the response card behind the front cover of your document, or write to the following address:

Cisco Systems
Attn: Document Resource Connection
170 West Tasman Drive
San Jose, CA 95134-9883

We appreciate your comments.

# Obtaining Technical Assistance

Cisco provides Cisco.com as a starting point for all technical assistance. Customers and partners can obtain documentation, troubleshooting tips, and sample configurations from online tools by using the Cisco Technical Assistance Center (TAC) Web Site. Cisco.com registered users have complete access to the technical support resources on the Cisco TAC Web Site.

## Cisco.com

Cisco.com is the foundation of a suite of interactive, networked services that provides immediate, open access to Cisco information, networking solutions, services, programs, and resources at any time, from anywhere in the world.

Cisco.com is a highly integrated Internet application and a powerful, easy-to-use tool that provides a broad range of features and services to help you to

- Streamline business processes and improve productivity
- Resolve technical issues with online support

- Download and test software packages
- Order Cisco learning materials and merchandise
- Register for online skill assessment, training, and certification programs

You can self-register on Cisco.com to obtain customized information and service. To access Cisco.com, go to the following URL:

http://www.cisco.com

# Technical Assistance Center

The Cisco TAC is available to all customers who need technical assistance with a Cisco product, technology, or solution. Two types of support are available through the Cisco TAC: the Cisco TAC Web Site and the Cisco TAC Escalation Center.

Inquiries to Cisco TAC are categorized according to the urgency of the issue:

- Priority level 4 (P4)—You need information or assistance concerning Cisco product capabilities, product installation, or basic product configuration.
- Priority level 3 (P3)—Your network performance is degraded. Network functionality is noticeably impaired, but most business operations continue.
- Priority level 2 (P2)—Your production network is severely degraded, affecting significant aspects of business operations. No workaround is available.
- Priority level 1 (P1)—Your production network is down, and a critical impact to business operations will occur if service is not restored quickly. No workaround is available.

Which Cisco TAC resource you choose is based on the priority of the problem and the conditions of service contracts, when applicable.

## Cisco TAC Web Site

The Cisco TAC Web Site allows you to resolve P3 and P4 issues yourself, saving both cost and time. The site provides around-the-clock access to online tools, knowledge bases, and software. To access the Cisco TAC Web Site, go to the following URL:

http://www.cisco.com/tac

All customers, partners, and resellers who have a valid Cisco services contract have complete access to the technical support resources on the Cisco TAC Web Site. The Cisco TAC Web Site requires a Cisco.com login ID and password. If you have a valid service contract but do not have a login ID or password, go to the following URL to register:

http://www.cisco.com/register/

If you cannot resolve your technical issues by using the Cisco TAC Web Site, and you are a Cisco.com registered user, you can open a case online by using the TAC Case Open tool at the following URL:

http://www.cisco.com/tac/caseopen

If you have Internet access, it is recommended that you open P3 and P4 cases through the Cisco TAC Web Site.

## Cisco TAC Escalation Center

The Cisco TAC Escalation Center addresses issues that are classified as priority level 1 or priority level 2; these classifications are assigned when severe network degradation significantly impacts business operations. When you contact the TAC Escalation Center with a P1 or P2 problem, a Cisco TAC engineer will automatically open a case.

To obtain a directory of toll-free Cisco TAC telephone numbers for your country, go to the following URL:

http://www.cisco.com/warp/public/687/Directory/DirTAC.shtml

Before calling, please check with your network operations center to determine the level of Cisco support services to which your company is entitled; for example, SMARTnet, SMARTnet Onsite, or Network Supported Accounts (NSA). In addition, please have available your service agreement number and your product serial number.

CHAPTER 1

# Getting Started

The Broadband Provisioning Registrar (BPR) software includes a provisioning application programming interface (API) that you can use to automate the provisioning of cable modems, digital set-top boxes (DSTBs), computers, and IP telephony devices on service provider networks. This chapter helps you get started with the development process.

## What is Provisioning?

A provisioned device meets two conditions: it is registered, and it is activated.

- Registration means that a provisioning API call (for example, addDOCSISModem) added the device to BPR. Otherwise, the device is unregistered.
- Activation means that the device has booted and received a configuration from BPR. Otherwise, the device is not activated.

In the absence of either of these conditions, the device is not provisioned.

## Provisioning Flows

Self-provisioning and preprovisioning are the two most common provisioning flows. Each flow passes through a set of states defined by the attributes registered/unregistered and activated/not activated.

### Self-Provisioning

During self-provisioning, device activation precedes device registration. Self-provisioning passes through these states:

1. Unregistered and not activated. A subscriber buys the device.
2. Unregistered and activated. The subscriber connects the device, turns it on, and BPR gives the device a default configuration for an unregistered device.
3. Registered and activated. The subscriber uses a web user interface to register the modem.

## Preprovisioning

During preprovisioning, device registration precedes device activation. Preprovisioning passes through these states:

1. Unregistered and not activated. The device arrives at the service provider's warehouse.
2. Registered and inactivated. The service provider adds the device to BPR.
3. Activated and registered. The device boots and receives the class of service defined when the service provider added the device.

# Configuration Details

The first-time configuration of a device is the result of a collaboration between the RDU, DPE, and Network Registrar extensions. Network Registrar extensions send DHCP information to the RDU. The RDU stores this information, then generates a configuration context which contains:

- Stable configuration information
- An instruction set for generating the dynamic configuration information

The RDU also validates rules to decide when the configuration is invalid. It then passes this configuration context to the DPEs. Unless subsequent configuration requests from the device are deemed invalid, they only require the Network Registrar extensions and the DPE.

## The RDU

When a device appears on the network for the first time, the RDU builds the initial configuration information using data collected from several sources.

- The DHCP DISCOVER packet adds the MAC address of the device and an options field containing information that can be processed to identify the technology.
- The BPR Network Registrar extension point adds provisioning group information.
- A device detection extension point on the RDU determines the technology and if a computer or custom CPE, the modem in front of the computer.
- A device record, created when a provisioning client added the device to BPR, adds the MAC address, class of service, DHCP criteria, and properties for the device. If a record has not been added, the default Class of Service and DHCP criteria is used for the device's technology.

Having collected the data, the RDU runs an extension point for the technology and generates the configuration information for the device. The RDU then sends the configuration information to all DPEs in the device's provisioning group.

## The DPE

The DPE stores the configuration context that the RDU has sent it. It responds to configuration requests, being from the NR extensions for DHCP configuration for a device, or from the device requesting a TFTP file.

The DPE adds to the initial configuration information by performing realtime mixing of data such as:

- A time stamp from the Time of Day (TOD) server
- The location of the Trivial File Transfer Protocol (TFTP) server
- A shared secret for a DOCSIS device (from the DOCSIS technology defaults)

## Network Registrar Extensions

The Network Registrar extensions request a configuration from the best available DPE. This configuration contains operations to perform on the DHCP response that is sent to the device. These operations include: setting the TFTP server as the DPE that returned the configuration, and setting the host and domain name.

## Configuration Changes

Changes to configuration occur under these conditions:

- When the device appears on the network for the first time.
- Because of a provisioning API call.
- When the contents of a device's DHCP request packet change (for example, when the relay agent changes).
- When a device roams to a different provisioning group.
- When a computer or custom CPE roams to a different computer.

# Provisioning Clients

The goal of the development process is the creation of one or more provisioning clients that use the provisioning API commands. The service provider's application then calls the provisioning client, which in turn, serves as the entry point to the BPR regional distribution unit (RDU).

**Note** For simplicity, the term provisioning client encompasses any implementation of provisioning API commands, including event driven implementations. Also for simplicity, the term service provider application refers to any application that calls a provisioning client, such as an Operations Support System (OSS), workflow, or billing system.

A single provisioning client can handle all provisioning tasks, or the tasks can be divided among separate clients. High-level provisioning client tasks include:

- Initialization—Seed BPR with the default service definitions needed when a device first appears on the network.
- Modifying Defaults—Modify the default service definitions after a subscriber has signed up for specific services.
- Device Management—Report on, add, and modify devices, or provision subscribers for additional services.
- Provisioning Flows—Enable self-provisioning of broadband devices.

# What You Need and Where to Find It

Use the information in this section to help determine how best to use the resources associated with BPR.

## Provisioning Clients and the Provisioning Process

This chapter contains information on writing a provisioning client and helps you understand how your provisioning client behaves during the provisioning process.

### Use Cases

Chapter 2, "Provisioning API Use Cases," outlines the procedures for common provisioning use cases, such as:

- Changes in services
- Registration of devices
- Addition and deletion of devices
- Bulk provisioning of devices

Each procedure includes provisioning API code segments in pseudocode.

### The Demonstration Program

The demonstration program shows how to call the provisioning API to perform various tasks, through a web-based interface.

**Caution** The demonstration program is intended only to give you an understanding of how to use the provisioning API to provision devices. It is not an out-of-the box solution for your provisioning needs. The installation process automatically installs the demonstration program at this location: <BPR_HOME>/rdu/DemoUI.

### Sample Code Directory

You can find BPR code examples and sample technology templates at this location: <BPR_HOME>/rdu/samples.

## Provisioning API and JavaDoc

The provisioning API supports several technologies; each with it's own separate interface. The JavaDoc that accompanies this product fully describes the provisioning API.

## Device and Technology Support

The provisioning API supports provisioning of these devices:

- Cable modems
- Digital set-top boxes (DSTBs)
- Customer premises equipment (CPE), the computers behind modems

The provisioning API supports these technologies:

- Data-over-Cable Interface Specifications (DOCSIS) high-speed data (HSD) services for broadband cable modems and DSTBs.
- Gateway Control Protocols (xGCPs), such as the Simple Gateway Control Protocol (SGCP) for residential voice services.
- The middleware component in a DSTB.

**Note** Basic DHCP devices like computers are also supported as technologies.

## Interfaces

The provisioning API supports these interfaces:

- Configuration—Get/set server defaults or settings, handle licensing, create custom property definitions.
- Provisioning—Query devices, generate configurations, reset, add/delete a class of service.
- DOCSIS—Manage DOCSIS devices.
- Computer—Manage computers.
- DSTB—Manage DSTB devices.
- XGCP—Manage voice service for DOCSIS modems.
- DeviceSearch—Search for multiple devices by class of service, IP address, MAC address, DHCP criteria, and FQDN.

## JavaDoc

The JavaDoc packaged with BPR describes each method in the provisioning API and includes:

- Command descriptions
- Parameters, including all valid keys for property maps
- Sample code
- Events fired
- Returned command status codes
- Returned data type codes

For convenience, this guide includes a listing of the provisioning APIs. See Appendix A, “Provisioning API Reference” for more information.

# Migrating Device Provisioning Registrar (DPR) Code

The CSRC DPR provisioning API is a subset of the BPR provisioning API. However, you may need to make changes in your CSRC DPR code because:

- Some methods have been deprecated, meaning that they may be discontinued in the future. See Appendix B, "Deprecated Methods" or the JavaDoc, for a complete list of deprecated methods.

  BPR deprecates the DOCSIS and computer methods from the provisioning interface, but includes new DOCSIS and computer interfaces. In all cases, migration to the new method is recommended.
- BPR defines the class of service object separately.
- BPR validates all properties. The JavaDoc describes the valid property keys that you can use with each method.
- BPR supports additional parameters that may be useful in your previous code. For example, BPR supports authentication by means of user name and password when submitting a batch.
- BPR supports the concept of DHCP criteria.

BPR API methods are signature compatible with the corresponding CSRC DPR 2.0 API methods.

**Note** The error conditions associated with some methods, have been changed to provide a richer set of error codes for command processing. Additionally, some ambiguous behavior in DPR is more clearly defined in BPR. Both improvements have resulted in a potentially significant code update between the two products.

# Desktop Requirements

Your desktop development environment must include:

- The provisioning API
- The Java 2 Software Development Kit and the Java Runtime Environment (JRE)

**Note** BPR uses version 1.3.1 of the Java 2 Software Development Kit. You can find complete information on the Java 2 Software Development Kit and Java components at the Sun Microsystems website.

# Additional Information

In addition, you need:

- Information on accessing the technologies that you need to provision. For example, see the DOCSIS specification for information on DOCSIS options.
- Cisco Network Registrar configuration information. Your administrator initially configures Network Registrar. The commands you initiate will use the values configured for Network Registrar. For more information, see the "DHCP Criteria" section on page 1-17.

You may also need:

- Administrator user interface information. BPR includes an administrator user interface that uses the provisioning API for initial configuration. For information on the administrator user interface, see the *Broadband Provisioning Registrar Administrator's Guide*.

- Logging information. For troubleshooting, the administrator can enable logging. Logging in BPR is dynamic, that is, the administrator or the provisioning API can change the logging settings while the server is running. Each line in the log records an operation with day, date/time, and thread information.
- Information on alerts. BPR includes an alert service for system level conditions. For example, the alert service sends a message when servers are running out of disk space or memory.

# Setting Up Your Environment

This procedure outlines a general approach to setting up your environment.

1. Install the Java 2 Software Development Kit.
2. Copy the appropriate provisioning .jar file into your environment. For example:

   ```
   C:\provisioning\bpr.jar
   ```

3. Put the .jar file in the class path.
4. Include the appropriate import statements for the provisioning packages in your Java code. For example:

   ```
   import com.cisco.provisioning.cpe.*;
   ```

5. Compile your code (a check of links). For example:

   ```
   javac -cp c:\provisioning\bpr.jar myProg.java
   ```

6. Run the program. For example:

   ```
   java -cp c:\provisioning\bpr.jar myProg
   ```

# Submitting Batches

A provisioning client submits requests to the RDU in the form of batches containing single or multiple commands. The batch is submitted via a connection to the RDU (PACEConnection). The RDU in turn maintains a queue of all batches, including the status of the batch. If the RDU goes down, it can restore all reliable batches in the queue.

Batch support includes:

- Support for atomic batch processing, meaning that either all commands in the batch succeed or no commands succeed. If the batch fails, the RDU rolls back all changes.
- Support for synchronous batch submission.
- Support for asynchronous batch submission with results returns in events
- Support for batch flags that control device activation.
- Support for status flags that monitor results.
- Support for XML batch submission.
- Support for reliable batches.

When you connect to the RDU and then submit a batch, the RDU:

- Receives your request and, if reliable, the batch is also journalled to allow for recovery if the server crashes.
- Puts the batch in a queue.
- Validates and processes the batch.
- Sends back the status and results.

# Batch Submission Options

You can choose either synchronous or asynchronous batch execution. Asynchronous batch execution depends on the firing of an event. The service provider's application can register for events such as:

- IP changes
- Device modifications
- Configuration changes
- External file changes
- Batch completions
- Changes in system defaults
- Changes in server properties or defaults

To use events, define and register a listener and qualifier on the PACEConnection for a batch, and then follow the steps for creating batches. See Chapter 2, "Provisioning API Use Cases" for examples. Event notifications return through commands defined for each event. The service provider's application then needs to monitor these commands, and the service provider's application determines the nature of the notification it wishes to receive.

# Batch Flags

By default, batches submitted to the RDU include flags that can control device disruption and the progress of configuration generation. Batches that you pass to the RDU include:

- An activation mode that determines the progress of configuration file generation and whether to attempt reset of the device. The AUTOMATIC flag is the only activation mode flag that attempts device reset.
- A confirmation mode that specifies whether BPR should check that the device has taken the configuration changes and that the device has been reset. The confirmation modes are used *only* if the activation mode is AUTOMATIC.

**Note** The AUTOMATIC flag calls for device reset, but devices can reset on their own when powered off and on. Assuming that a DOCSIS device is up on a network with appropriate access permissions, the default device disruption extension in BPR uses an Object ID (OID) in an SNMP MIB to reset the DOCSIS device. But, the network or the modem could be down thereby stopping BPR from automatically resetting the device.

### Activation Modes

The ActivationMode class defines these flags.

- NO_ACTIVATION—The RDU updates the information for the device and the DPE caches its configuration. However, the RDU does not attempt to reset the device.
- AUTOMATIC—The RDU updates the information for the device, the DPE caches its configuration, and the RDU attempts to reset the device. You can use different confirmation modes with the AUTOMATIC activation mode.

Each batch returns a status code. See the ActivationMode class in the JavaDoc for more information.

### Confirmation Modes

The ConfirmationMode class defines these flags:

- NO_CONFIRMATION—Does not confirm device reboot and returns with a BatchStatusCode value of BATCH_WARNING if disruption fails. NO_CONFIRMATION is the default.
- CUSTOM_CONFIRMATION—To take advantage of this mode, you must write your own specific device disruption extension point. See the ConfirmationMode class in the JavaDoc for complete information.

## Batch Guidelines

Keep these batch guidelines in mind:

- Each batch must have a unique identifier. BPR generates a unique identifier unless you provide one.
- You can only execute a batch once.
- You cannot add commands to a batch after it has been posted to the RDU.
- A batch can contain one or more commands. If a batch contains multiple commands, these batch commands are executed in the order they are added to the batch:
  - You cannot include read (get) and write (add, change, and delete) commands in the same batch.
  - If you do not use the default Activation and Confirmation modes, you must explicitly override them.
  - If the Activation mode is set to NO_ACTIVATION, the batch can affect multiple devices.
  - If the Activation mode is AUTOMATIC, the batch can activate only one device. For example, only one modem at a time can be added using this mode.
- It is up to the provisioning client to control the rate of device provisioning.

**Note** If the client goes too fast, the RDU will return batch dropped errors.

- There are special cases for handling:
  - For Provisioning.generateConfiguration use NO_ACTIVATION. This command generates new device configurations, but does not attempt to reset the device. Consequently, the command does not support the AUTOMATIC mode.
  - For Provisioning.resetIPDevice use AUTOMATIC. This command does not generate new device configurations, but will reset the device.

# Creating a Batch

Follow these steps to create a batch:

1. Open a PACE connection to the RDU.
2. Create a new batch.
3. Get the appropriate API interfaces.
4. Reference commands to the batch.
5. Post the batch to the RDU.
6. Get the status of the batch.
7. Get the requested query data, if it is a read batch.
8. Close the connection.

The following sections discuss each of these steps in more detail.

## Step 1—Open a PACE Connection to the RDU

Before you can begin to build batches and send commands to the RDU, you must first open a connection. Open the connection by calling a getInstance() command on the PACEConnectionFactory class. Example 1-1 shows the code that calls a getInstance() command.

***Example 1-1 Get a Connection Using an RDU Host Name***

```
PACEConnection connection = null;
try
{
     connection =
           PACEConnectionFactory.getInstance(
          "rdu.mycompany.com",             // RDU server host name
          49187,                           // RDU port number
          "admin",                         // username
          "changeme");                     // password
}
catch (PACEConnectionException e)
catch (BPRAuthenticationException e)
{
 // connection failed -- handle exception here
}
```

**Note** To verify that the connection is established after a call to getInstance(), use the isAlive() call on the PACEConnection to see if the connection is alive.

Alternatively, you can use the IP address of the RDU server. Example 1-2 shows code that calls a getInstance() command using the IP address of the RDU server:

***Example 1-2 Get a Connection Using an RDU Server IP Address***

```
PACEConnection connection = null;
try
{
     connection =
           PACEConnectionFactory.getInstance(
          "191.14.236.129",               // RDU server IP address
          49187,                          // RDU port number
          "admin",                        // username
          "changeme");                    // password

}
catch (PACEConnectionException e)
catch (BPRAuthenticationException e)
{
 // connection failed -- handle exception here
}
```

**Note** BPRAuthenticationException is a run-time exception and, therefore, your compiler will not catch it.

# Step 2—Get a New Batch

After you open a PACE connection to the RDU, you must get a new batch by calling a newBatch() command on the PACEConnection interface. Each batch must have a unique identifier, an Activation mode, and a Confirmation mode. See the JavaDoc for complete descriptions of the flags.

- If you do not specify a batch identifier, BPR generates a globally unique identifier.
- If you do not specify the Activation mode, the default is NO_ACTIVATION.
- If you do not specify the Confirmation mode, the default is NO_CONFIRMATION.

Example 1-3 shows typical flags on the newBatch() command:

***Example 1-3 Typical Batch Flags***

```
// get a batch with a BPR-generated identifier, using defaults
Batch myBatch = connection.newBatch();    // NO_ACTIVATION and NO_CONFIRMATION assumed


// get a batch with BPR-generated identifier, AUTOMATIC mode, NO_CONFIRMATION assumed
Batch myBatch2 = connection.newBatch(ActivationMode.AUTOMATIC);
```

# Step 3—Reference the Appropriate API Interfaces

After you get a new batch, and before you can add commands to the batch, you must reference the command interface or interfaces that you need. For each new batch, use the API names class to specify the target interface. Example 1-4 shows typical referencing of interfaces.

***Example 1-4 Referencing API Interfaces***

```
Configuration configurationCommands =
(Configuration)myBatch.getProvAPI(APINames.Configuration);


Provisioning provisioningCommands =
(Provisioning)myBatch.getProvAPI(APINames.Provisioning);


DeviceSearch deviceSearchCommands =
(DeviceSearch)myBatch.getProvAPI(APINames.DeviceSearch);


DOCSIS docsisCommands =
(DOCSIS)myBatch.getProvAPI(APINames.DOCSIS);
```

# Step 4—Add Commands to the Batch

Once you have referenced the command interface or interfaces you need for the current batch, you are ready to add commands to the batch. Example 1-6 shows how to queue commands that preprovision a DSTB to myBatch.

***Example 1-5 Queue Commands to Preprovision a DSTB***

```
docsisCommands.addDOCSISModem(
     "1,6,aa:00:bb:99:cc:55",           // macAddress for DSTB DOCSIS modem component
     null,                              // FQDN: not used in this example
     "012-34-5678",                     // ownerID
     "Gold",                            // classOfService
     "provisionedCM",                   // modem DHCP criteria
     null,                              // cpe DHCP criteria
     null);                             // properties

dstbCommands.addDSTB(
     "1,6,99:cc:22:77:dd:ff",           // macAddress for DSTB middleware component
     null,                              // FQDN: not used in this example
     "012-34-5678",                     // ownerID
     "provisionedDSTB",                 // DHCP criteria
     "1,6,aa:00:bb:99:cc:55",           // relatedDOCSISModemID: macAddress for DOCSIS
     null),                             // DVB modem component of DSTB
     null),                             // properties
```

Each of these commands is implicitly added to myBatch because the command accessors (in the example, provisioningCommands and dstbCommands) are implicitly tied to the batch from which they were retrieved (See the "Step 3—Reference the Appropriate API Interfaces" section on page 1-11). You cannot use these command accessors to add commands to another batch (for example, myBatch2). If you wish to perform additional commands, after myBatch has been posted, you must first create a new batch and then re-retrieve the command accessors for the new batch.

Example 1-6 adds a query command to a separate batch. The query gets the device details for a particular device identified by its MAC address:

***Example 1-6 Add Command to Get Device Details***

```
// queue command to myBatch2
provisioningCommands2.getDetailsForIPDevice(
"1,6,99:cc:22:77:dd:ee");   // macAddress for requested device
```

# Step 5—Post the Batch to the RDU

The commands are now queued to the batch, but have not yet executed. To execute these commands, you must post the batch to the RDU. Example 1-7 shows the code that posts a batch.

***Example 1-7 Post a Batch***

```
BatchStatus bStatus = null;
try
       {
       bStatus = myBatch.post();
       }
catch (ProvisioningException e)
       {
       // batch posting failed -- handle exception here
       }
```

The post() command instructs the RDU to execute the commands that have been queued to the batch. The call waits while the RDU processes the batch which, depending on the number (and complexity) of batches queued at the RDU, could take several minutes to complete. The RDU either returns a BatchStatus or it generates an exception. A BatchStatus details either errors encountered with particular commands or the success status for the posted batch. An exception indicates that the RDU could not process the batch.

**Note** If getInstance() is unable to contact the RDU, the post method will return a PACEConnectionException or a BPRAuthenticationException to indicate if something is wrong with the connection.

# Step 6—Get the Status of the Batch

If the post() command in the previous step did not throw an exception, the batch has been processed and you have the BatchStatus return value.The BatchStatus contains a status code indicating either that the batch executed successfully, or that there were errors. In addition, the BatchStatus provides access to status for the individual commands in the batch. If the batch failed, you can get additional information as to which command caused the batch to fail. And, if the batch succeeded, and the batch consisted of query commands, then the CommandStatus provides access to the requested data (see the "Step 7—Get the Requested Query Data (If Required)" section on page 1-14).

Example 1-8 examines the BatchStatus return value from the previous step:

***Example 1-8 Examine the Returned BatchStatus***

```
CommandStatus cStatus = null;
String errorText = null;
if (bStatus.isError())
{
     cStatus = bStatus.getFailedCommandStatus();

     if (cStatus != null && cStatus.getErrorMessage() != null)
     {
           errorText =
          + "\tcommand error message = " + cStatus.getErrorMessage();
     }
     else
     {
          errorText = ("BatchID = " + bStatus.getBatchID() + "\n"
          + "\tbatch error message = " + bStatus.getErrorMessage() )+ "\n"
     }
     // do error processing
}
     else (bstatus.isWarning())
     {
          // Process Warning
     }
```

# Step 7—Get the Requested Query Data (If Required)

If the posted batch consisted of query commands, use the CommandStatus command to extract the requested data from the BatchStatus returned by Batch.post(). Example 1-9 extracts the returned data from a batch that contained a single query (see Example 1-6):

***Example 1-9 Extract the Query Data from the CommandStatus***

```
CommandStatus cStatus = null;
String errorText = null;
if (!bStatus.isError())
{
       // get command status for the 0-th command in the batch
       cStatus = bStatus.getCommandStatus(0);

       if (cStatus == null)
       {
              errorText = “Command status is null.”;
       }
       else if (cStatus.getDataTypeCode() == CommandStatus.DATA_MAP)
       {
              Map deviceDetails = (Map)cStatus.getData();
              String ipAddress =
                     (String)deviceDetails.get(DeviceDetailsKeys.IP_ADDRESS));
       }
```

```
}
       else (bstatus.isWarning())
       {
              // Process Warning
       }
```

## Step 8—Close the Connection

Repeat Steps 2 through 7 as many times as you want for the given connection. When you are done posting batches, you should close the connection and release system resources. However, if the client is continually connecting and disconnecting from the RDU, it may be more efficient to leave the connection open. Example 1-10 shows how to close the connection.

***Example 1-10 Close the Connection***

```
connection.releaseConnection();
```

# Key Parameters When Adding a Device

When adding a device, a provisioning client passes a number of important parameter values to the RDU.

## Device and Owner Identification Information

Device identification information can include:

- MAC address
- FQDN
- Owner ID

### MAC Address

The MAC address is a combination of the hardware type, hardware length, and the hardware address of a device. It is the address of the Ethernet interface of the device. BPR uses the MAC address as a key identifier of a device and ensures that the address is unique for each device. For example, if the hardware type is “1”, the hardware address length is “6”, and the hardware address is “00:01:02:nn:nn:nn”, then the MAC address of the device is 1:6:00.01.02:nn:nn:nn, where n is a digit.

MAC addresses are found in commands that:

- Add a device (for example, addDOCSISModem)
- Change the MAC address for a device (for example, changeDOCSISModemMACaddress)
- Get by MAC addresses (for example, getDetailsFor..., getIPDevicesbyMACaddress)

### FQDN

The fully qualified domain Name (FQDN) is the full name of a system, that is a host name and domain name in dot notation. For example, if the host name is “foo” and the domain name is “bar.com”or “bar.com.”, then the FQDN is “foo.bar.com” or “foo.bar.com.”

BPR ensures that the FQDN is unique for each device. Your system must enable dynamic DNS (DDNS) on the Network Registrar DHCP server and DNS servers to populate DNS with device FQDNs.

FQDNs are found in commands that:

- Add a device (for example, addDOCSISModem)
- Change device FQDNs (for example, changeComputerFQDN)
- Get by FQDNs (for example, getIPDevicesByFQDN, or getDetailsFor...)

### Owner Identifier

The owner identifier (ownerID) correlates all devices belonging to one owner. It can, for example, associate a device with an account number used by a billing system. The owner identifier is a string which can be anything including numbers.

The ownerID is found in commands that:

- Add a device (for example, addComputer)
- Change ownerIDs (for example, changeDOCSISModemOwnerID)
- Get by ownerIDs (for example, getIPDevicesByOwnerID or getDetailsFor...)

## Class of Service

A class of service is the template for the device configuration. The service model defines classes of service, such as Gold with a list of options that can include:

- Connection speed
- Upstream an/or downstream bandwidth setting
- Ports open or blocked
- Web server hosting

### Initial Configuration

Initial configuration for the classes of service takes place in the administrator application, where classes of service are defined for each technology. For example, both DOCSIS devices and computers will have a default class of service for devices that have not been previously registered with BPR by a command, such as addDOCSISModem or addComputer.

A class of service specification is found in commands that:

- Add or delete a class of service (for example, addClassOfService, deleteClassOfService)
- Get classes of service (for example, getAllClassesOfService, getClassOfServiceProperties)
- Add a device (for example, addDOCSISModem, addComputer)
- Change a devices class of service (for example, changeDOCSISModemClassOfService)
- Change a class of service (for example, changeClassOfServiceProperties)

When specifying classes of service, keep these points in mind:

- If BPR does not already have an existing definition for the class of service you pass to the method, an error is generated.

### DOCSIS Class of Service Templates

BPR can store class of service configurations as templates (for example, gold.tmpl or unprov.tmpl). When templates are used, the templates can contain hard-coded specifications for each configuration option or macros that read configuration options from properties passed in by a provisioning client. To understand your current class of service definitions and their implementations, view the individual files. For information on building templates, see the "Building Class of Service Templates" section on page 1-20.

## DHCP Criteria

The DHCPCriteria object bundles a set of parameters that targets a certain criteria for IP address selection and may contain additional DHCP options. A provisioning client references a DHCPCriteria object by identifier. Each device has one associated DHCPCriteria object. A DHCP criteria may be a client-class and/or selection-tag and determines which scope a device will be put in when it issues a DHCP discover.

**Caution** BPR and Network Registrar have configuration items for some of the same options. In particular, DHCP option properties can be configured in both systems. If there is contention between the values configured in Network Registrar and values specified in a provisioning client, the values in BPR will override the values configured in Network Registrar.

All client classes and selection tags must first be set up in Network Registrar before they can be used by a provisioning client. An incorrect association of class of service, client class, and selection tags in your client may mean that the batch cannot succeed or can result in an inappropriately provisioned device.

### A Client Class Example

An administrator has created a client class called ccProvComputer in Network Registrar. A scope selection tag (tagProvComputer) has also been created and associated to the client class ccProvComputer and a scope. A provisioning client has added a device to BPR with a DHCPCriteria object that uses the client class ccProvComputer.

**Note** The scope selection tag tagProvComputer is not added to the DHCP criteria in this case.

The device boots and BPR uses the client class ccProvComputer to a device. Network Registrar now gives the device an IP address from a scope that has tagProvComputer assigned and that has the correct gateway address. Network Registrar also updates the DHCP reply with any policy options that have been configured on the client class ccProvComputer.

## A Selection Tag Example

A scope selection tag has been created in Network Registrar and associated tagComputer with a scope. A device has been added to BPR with a DHCPCriteria object that uses tagComputer. The device boots and BPR uses this device the scope selection tag tagComputer. Network Registrar now gives this device an IP address from a scope that has the tagComputer assigned and has the correct gateway address.

## Important Programming Points

When specifying DHCP criteria, keep these points in mind:

- DHCP criteria specifications may include selection tags. If the selection tags have been specified in Network Registrar, it is not necessary to specify them in BPR. If a selection tag is specified in both places, then the BPR selection tag overrides the Network Registrar selection tag.
- DHCP criteria specifications are case sensitive and must be identical with the specifications in Network Registrar. The command will fail if you specify an arbitrary string.
- If the Network Registrar configuration definitions do not include the client class or selection tags you use, the device cannot get an IP address.

# Properties

Properties are used throughout the provisioning API to provide optional arguments to the commands. They are either predefined in BPR or added through the provisioning API. BPR stores properties as maps containing key/value pairs, and validates each property. See the com.cisco.provisioning.cpe.constants package in Appendix A, “Provisioning API Reference” or the JavaDoc for a listing of all keys.

## Property Validation

When BPR is processing an API operation, each command accepting properties validates each predefined or custom property. Use the addCustomPropertyDefinition method to register new properties with BPR. BPR returns an error if the property is not registered or if the specification of the property does not exactly match the registered name of the property.

## Order of Precedence for Properties

BPR supports two types of properties, each with an underlying order of precedence. The two types are:

- Device properties
- System properties

**Note** The two property types do not interact.

Table 1-1 shows the precedence hierarchy for device properties and the methods that return the device properties on each level.

*Table 1-1 Device Property Precedence Hierarchy*

| Precedence Hierarchy | Methods that Return Properties on a Given Level |
|---|---|
| System (global for device properties) | getSystemDefaults (for example, SNMP community strings) |
| Technology | getDOCSISDefaults, getXGCPDefaults, getComputerDefaults |
| DHCP criteria | getDHCPCriteriaDetails |
| Class of service | getClassOfServiceProperties |
| Device (lowest level) | getDetailsFor... |

Table 1-2 shows the precedence hierarchy for system properties and the methods that return the system properties on each level.

*Table 1-2 System Property Precedence Hierarchy*

| Precedence Hierarchy | Methods that Return Properties on a Given Level |
|---|---|
| System (global for system properties) | getSystemDefaults (for example, SERVER_TRACE_ENABLE) |
| Server (lowest level) | getRDUDefaults, getDPEDefaults, getCNRDefaults |

For properties that exist on more than one level, the property on the lowest level overrides the property on a higher level. For example, a property set on the IP device level overrides the same property set on the class of service level. See the JavaDoc for specific information on property levels.

All properties supported on a lower level will also be supported on the higher level. But the reverse may not be true. For example, all properties supported on a DOCSIS modem (at the technology level) are supported on the DOCSIS class of service. But all the properties supported on the DOCSIS class of service may not be supported on a DOCSIS modem.

# Registration Modes

Registration modes implement the business rules by which the service provider determines the number of interactions with the subscriber. For any registered device, the service provider must be prepared to process any change to the device. So there is a significant quantitative difference between registering 100 cable modems with unregistered computers behind them and registering 100 cable modems each of which has a potentially large number of registered computers behind it.

In order to control device changes, BPR includes these registration modes:

- Standard mode—The modem and all computers behind it are registered. Standard mode has two options.
  - Roaming (the default)—A registered computer can receive its class of service behind any registered modem.
  - Fixed—A registered computer can receive its class of service only when the computer boots behind a specific modem. To invoke fixed standard mode, set the MUST_BE_BEHIND_DEVICE property in the IPDeviceKeys interface to the MAC address of the modem. If the computer reboots behind a different modem, the computer becomes unprovisioned. See the JavaDoc for complete information.

- Fixed Prov Group mode— A registered device can receive its class of service only when it is part of the specified provisioning group. To invoke the Fixed Prov Group mode, set the MUST_BE_IN_PROV_GROUP property, in the IPDeviceKeys interface, to the required provisioning group name. If the device reboots in a different provisioning group, it becomes unprovisioned. Refer to the JavaDoc for complete information.
- Promiscuous mode—Only the modem is registered, and the DHCP server maintains lease information about the computers behind the modem. All computers behind a registered modem receive network access, although the total number of computers given access at any given moment can be controlled by the maxCPEs DOCSIS parameter. To invoke promiscuous mode, enable the PROMISCUOUS_MODE_ENABLED property in the ModemKeys interface and cpeDHCPCriteria. Refer to the JavaDoc for complete information.
- Network Address Translation (NAT) mode—Only the modem is registered and no information is maintained about the computers behind the modem, but all computers receive network access. NAT mode is invoked at the hardware level.

# Building Class of Service Templates

BPR includes sample templates that can define classes of service as well as other information specific to a technology implementation. You can use the sample templates, modify them, or write your own.

When working with templates, keep these important points in mind:

- The template must be registered with BPR using the addExternalFile method and the template file must end with a *.tmpl file extension.
- You must add each class of service using the addClassOfService method and the COS_DOCSIS_FILE key to specify the target template.
- If the template specifies a property, the property must be registered with BPR. See the "Properties" section on page 1-18 for more information.
- Macro substitution is possible.

Example 1-11 shows a simple template for a DOCSIS modem.

***Example 1-11 Simple DOCSIS Template***

```
# silver.tmpl:
#
# Silver class of service
option 3 1
option 4.1 1 [alternatively: option 4.1 ${macro1}]
option 4.2 512000 [alternatively: option 4.2 ${macro11,512000}]
option 4.3 64000
option 4.4 3
option 4.5 0
option 4.5 0
option 4.7 0

# Allow SNMP modem reset
include "modem_reset.tmpl"
# Max CPE's behind modem
option 18 2
```

# Provisioning In Detail

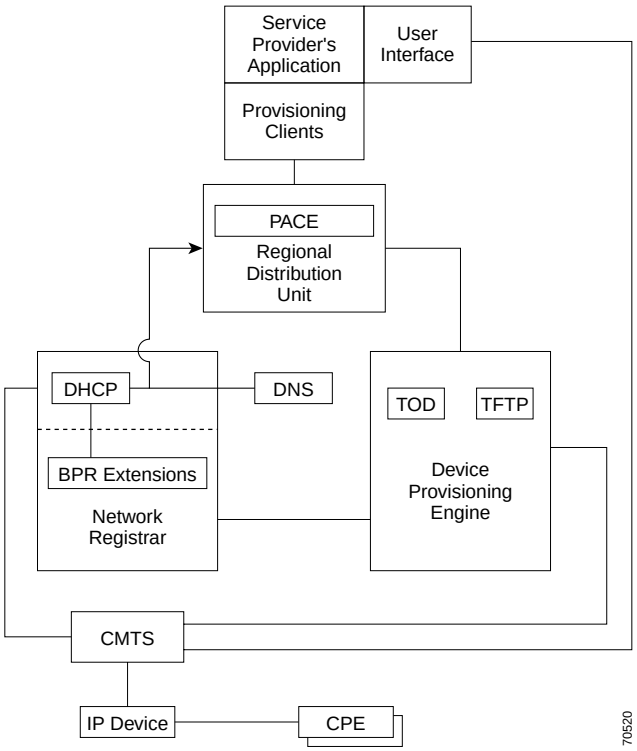
This section describes initial and subsequent activation and shows the process which a DOCSIS modem follows to receive a configuration from BPR. In this context, activation is the process of a device receiving a configuration (DHCP, TFTP, TOD) from BPR.

## A Provisioning Reference Model

Figure 1-1 is a reference model for the provisioning process.

**Note** Some provisioning flows, such as the voice flow, may require additional components not shown here.

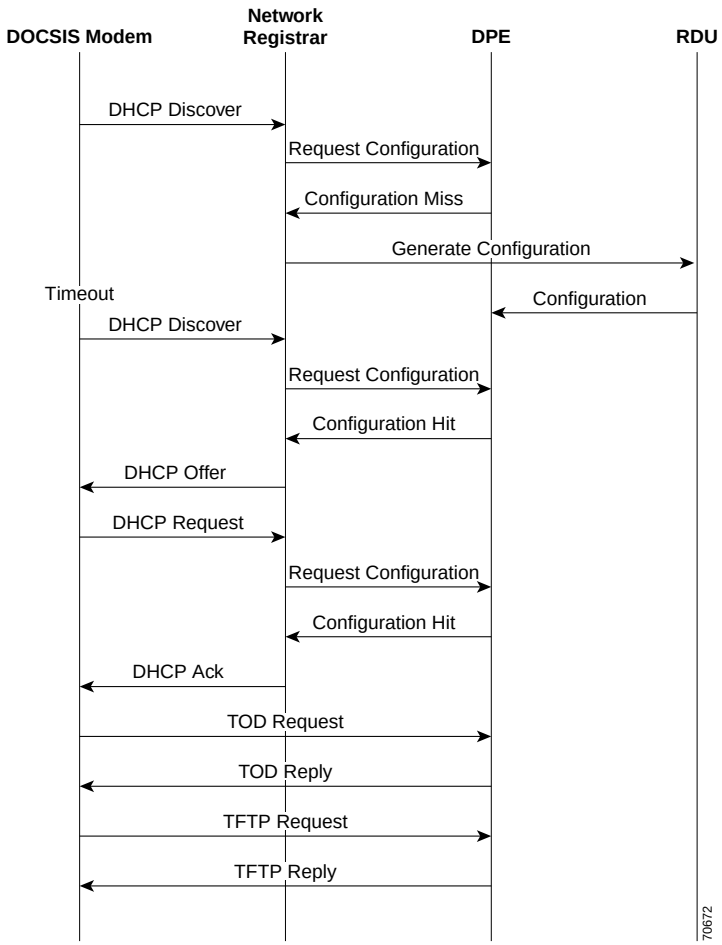
***Figure 1-1 Provisioning Reference Model***




## First Time Activation

First time activation refers to the first time BPR sees network traffic from the device. As shown in Figure 1-2, this initial traffic is the DHCP DISCOVER. The configuration given to the device depends on its registered state. If the device has been registered, in a process known as pre-provisioning, the device will automatically receive its registered level of service. However, if the device is unregistered, it only receives the default level of service. This is useful in self-provisioning where the desired outcome is to provide just enough service to the device to allow the subscriber to complete the provisioning process.

*Figure 1-2 Provisioning Flow, First Time On The Network*




This is the process by which a DOCSIS modem is activated for the first time.

## Step 1—Cable Modem Activity

**a.** The cable modem boots and sends out a DHCP DISCOVER packet. This packet includes the modem's MAC address which is used as a device identifier.

## Step 2—CMTS Interaction

**a.** The CMTS receives the packet and forwards it to Network Registrar.

**Note** For simplicity, assume that the traffic involved in all of the remaining steps passes through the CMTS.

## Step 3—Network Registrar Interaction

**a.** Network Registrar invokes a BPR extension point, which contacts a DPE (REQUEST CONFIGURATION).

## Step 4—DPE Interaction

The DPE:

a. Checks its cache and determines that it does not have a configuration for the modem based on the modem's MAC address.

b. Returns a negative response (CONFIGURATION MISS) to Network Registrar.

## Step 5—Network Registrar Interaction

a. Network Registrar drops the packet and requests the configuration (GENERATE CONFIGURATION) from the RDU. The information included in this requests identifies the devices provisioning group.

b. The cable modem times out and, after a few seconds, the modem sends a new DHCP DISCOVER packet.

## Step 6—RDU Interaction

If the device is registered, the RDU:

a. Runs device detection to determine the technology. The information in the request is used to determine if the device has roamed between provisioning groups.

b. Creates a configuration from the information specified when the device was registered. This includes the DHCP criteria and the class of service of the device.

c. Sends the configuration to every DPE in the provisioning group for this device.

If the device is unregistered, the RDU:

a. Runs device detection to determine the technology, using the information in the request to determine what kind of configuration to create (for example, DOCSIS).

b. Creates a configuration from the information generated by device detection, the information passed in by the Network Registrar extensions, and defaults for this technology.

c. Sends the device configuration to every DPE in the provisioning group to which the device is assigned.

## Step 7—DPE Interaction

a. The DPE caches the configuration information.

## Step 8—Cable Modem Interaction

a. After the modem times out, it resends a DHCP DISCOVER packet. This packet includes the modem's MAC address which is used as a device identifier.

## Step 9—Network Registrar Interaction

a. The Network Registrar invokes a BPR extension point, which contacts a DPE.

## Step 10—DPE Interaction

The DPE:

a. Checks its cache and finds a configuration for the modem based on the modem's MAC address.

b. Passes the configuration back to the Network Registrar. The configuration includes those DHCP options that must be in the DHCP reply.

## Step 11—Network Registrar Interaction

The Network Registrar:

a. Receives the configuration, and executes the commands specified in that configuration.

b. Returns a DHCP OFFER to the DOCSIS modem. This OFFER includes an IP address for the device.

## Step 12—Cable Modem Interaction

a. The cable modem sends a DHCP REQUEST packet to the Network Registrar. This packet includes the modem's MAC address, which is used as a device identifier.

## Step 13—Network Registrar Interaction

a. The Network Registrar invokes a BPR extension point, which contacts the DPE (REQUEST CONFIGURATION).

## Step 14—DPE Interaction

The DPE:

a. Checks its cache and uses the modem's MAC address to locate the proper device configuration.

b. Passes the configuration back to Network Registrar. The configuration includes those DHCP options that must be in the DHCP reply.

## Step 15—Network Registrar Interaction

The Network Registrar:

a. Receives the configuration and executes the commands specified in that configuration.

b. Returns a DHCP ACK.

## Step 16—Cable Modem Interaction

a. The cable modem sends a TOD REQUEST packet to the DPE.

## Step 17—DPE Interaction

a. The DPE sends a TOD REPLY packet to the modem.

### Step 18—Cable Modem Interaction

a. The cable modem sends a TFTP REQUEST packet to the DPE.

### Step 19—DPE Interaction

a. Mixes in any necessary information.

b. Returns TFTP response containing the device configuration.

### Step 20—Cable Modem

a. Validates the configuration file.

**Note** If this is a DOCSIS cable modem, the cable modem and CMTS may be configured for additional validation.

b. Enters an operational state.

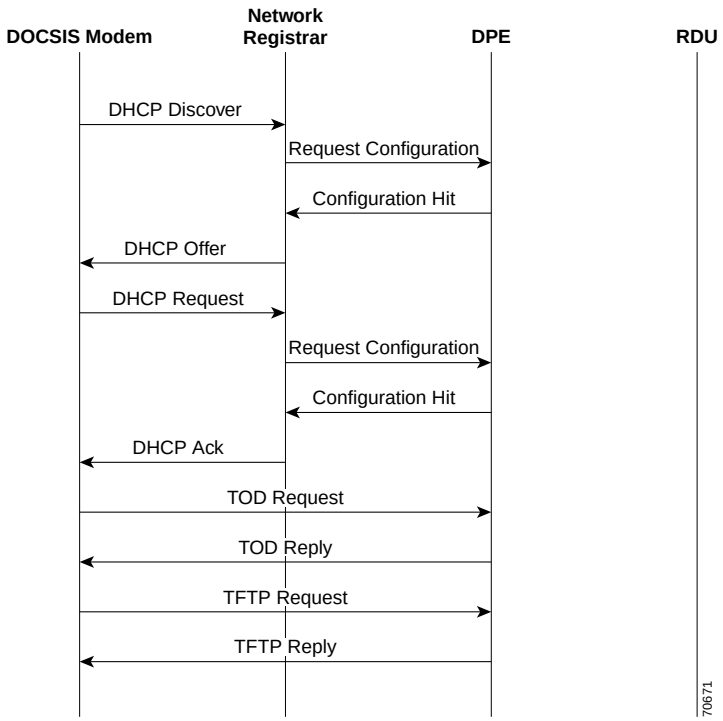
## Provisioning Computers

The process by which you provision a computer is identical to steps 1 through 15 provided for a DOCSIS modem. This is because the computer does not have to request ToD or a TFTP file to become provisioned.

## Subsequent Activation

After a device is initially activated, as shown in Figure 1-3, subsequent activation does not require communication with the RDU to request a configuration be generated. The RDU generates a new configuration if service selection on the device has changed.

*Figure 1-3 Provisioning Flow, Subsequent Activation On The Network*




These are the steps a DOCSIS modem takes on subsequent activations.

## Step 1—Cable Modem

**a.** Boots and sends a DHCP DISCOVER packet to the Network Registrar. The packet includes those DHCP options that must be in the DHCP reply.

## Step 2—CMTS Interaction

**a.** Receives and forwards the packet to the Network Registrar.

**Note** For simplicity, the remaining steps assume that all traffic passes through the CMTS.

## Step 3—Network Registrar Interaction

**a.** Network Registrar invokes a BPR extension point, which contacts the DPE.

## Step 4—DPE Interaction

The DPE:

**a.** Checks its cache and finds a configuration for the modem based on the modem's MAC address.

**a.** Passes the configuration back to the Network Registrar. The configuration includes those DHCP options that must be in the DHCP reply.

## Step 5—Network Registrar Interaction

The Network Registrar:

a. Receives the configuration, and executes the commands in the configuration.

b. Returns a DHCP OFFER, which includes an IP address for the device.

## Step 6—Cable Modem Interaction

a. The cable modem sends a DHCP REQUEST packet back to the Network Registrar. The packet includes the modem's MAC address which is used as the device identifier.

## Step 7—Network Registrar Interaction

a. The Network Registrar invokes a BPR extension point, which contacts a DPE.

## Step 8—DPE Interaction

The DPE:

a. Checks its cache and finds a configuration for the modem based on the modem's MAC address.

b. Passes the configuration back, which includes the DHCP options to be put into the DHCP reply.

## Step 9—Network Registrar Interaction

The Network Registrar:

a. Receives the configuration, and executes the commands in the configuration.

b. Returns a DHCP ACK.

## Step 10—Cable Modem Interaction

a. The cable modem sends a TOD REQUEST packet to the DPE.

## Step 11—DPE Interaction

a. The DPE sends a TOD REPLY packet back to the modem.

## Step 12—Cable Modem Interaction

a. The cable modem sends a TFTP REQUEST packet to the DPE.

## Step 13—DPE Interaction

The DPE:

a. Mixes in any necessary information.

b. Returns a TFTP REPLY containing the device configuration, to the cable modem.

## Step 14—Cable Modem

The cable modem:

**a.** Validates the configuration file.

**Note** If this is a DOCSIS cable modem, the cable modem and CMTS may be configured for additional validation.

**b.** Enters an operational state.

CHAPTER 2

# Provisioning API Use Cases

To help you understand how and where to use the provisioning API, this chapter presents a series of the most common provisioning API use cases, including code segments that you can use as models for your code. The code segments are embedded in typical service provider workflows.

**Note** This chapter uses pseudocode to illustrate the code segments. This pseudocode resembles Java code, but is *not* intended for compilation.

## A Quick Reference to the Use Cases

This section identifies all uses cases found in this guide.

**Fixed Standard Mode (new or existing modems and computers)**

- Self-Provisioned Modem and Computer in Fixed Standard Mode, page 2-2
- Add a New Computer in Fixed Standard Mode, page 2-5
- Disable a Subscriber in Fixed Standard Mode, page 2-6

**Roaming Standard Mode (new or existing modems and computers)**

- Preprovisioned Modem/Self-Provisioned Computer in Roaming Standard Mode, page 2-7
- Modify an Existing Modem in Roaming Standard Mode, page 2-8
- Delete a Subscriber's Devices in Roaming Standard Mode, page 2-8

**Promiscuous Mode (new or existing modems and computers)**

- Self-Provisioned First Time Activation in Promiscuous Mode, page 2-9
- Bulk Provision 100 Modems in Promiscuous Mode, page 2-11
- Preprovisioned First Time Activation in Promiscuous Mode, page 2-13
- Replace an Existing Modem in Promiscuous Mode, page 2-14
- Add a Second Computer in Promiscuous Mode, page 2-15

**NAT Mode (new modems and computers)**

- Self-Provisioning First Time Activation with NAT, page 2-15
- Add a New Computer Behind a Modem with NAT, page 2-17

**DSTBs (new or existing DSTBs)**

- Preprovisioned First Time Activation for DSTB, page 2-17
- Modify an Existing DSTB, page 2-18
- Replace an Existing DSTB, page 2-19
- Remove an Existing DSTB, page 2-20

**Voice**

- Preprovisioned xGCP Capable DOCSIS Modem, page 2-20
- Activating an Additional Telephone Line on an xGCP Capable DOCSIS Modem, page 2-22
- Disable an Existing Telephone Line on an xGCP Capable DOCSIS Modem, page 2-23

**Configuration Generation**

- Generate a Configuration, page 2-23

**DHCP Criteria**

- Add DHCP Criteria, page 2-24

**System Defaults**

- Change System Defaults, page 2-25

**Using Events**

- Log Device Deletions, page 2-25
- Monitor a Connection to the RDU, page 2-26
- Log Batch Completions, page 2-27

**Device Searches**

- Get Detailed Information for a Device, page 2-27
- Retrieve Devices Matching a Vendor Prefix, page 2-28

The remainder of this chapter contains detailed examples of these use cases.

# Self-Provisioned Modem and Computer in Fixed Standard Mode

**Case:** A subscriber has a computer (with a browser application) installed in a single dwelling unit and has purchased a DOCSIS cable modem.

**Desired Outcome:** Use this workflow model to bring a new unprovisioned DOCSIS cable modem and computer online so that they have the appropriate level of service.

**Step 1** The subscriber purchases a DOCSIS cable modem, installs it in the home, and connects the computer to the cable modem.

**Step 2** The subscriber turns on the modem and BPR gives it restricted access.

**Step 3** The subscriber turns on the computer, BPR gives it restricted access.

**Step 4** The subscriber starts a browser application on the computer and a spoofing DNS server points the browser to the service provider's registration server (for example, an OSS user interface or a mediator).

**Step 5** The subscriber uses the service provider's user interface to complete the steps required for registration, including selecting class of service.

**Step 6** The service provider's user interface passes the subscriber's information, such as the selected class of service and computer IP address, to BPR. The subscriber's modem and computer are then registered with BPR.

```
// First we query the computer's information to find the modem's MAC address
// We use the computers IP address (the web browser received this when the
// subscriber opened the service providers web interface)

get-new-batch(NO_ACTIVATION, NO_CONFIRMATION)

// NO_ACTIVATION is the activation mode because this is a query

// NO_CONFIRMATION is the confirmation mode because we are not attempting to
// reset the device

Map computerLease = getDetailsforIPDevicebyIPAddress(
      "10.0.14.38")             // ipAddress: restricted access computer lease, from
                                // Network Registrar DHCP
// Derive the modem MAC address from the computer's network information
// The 1,6, is a standard prefix for an Ethernet device
// The fully qualified MAC address is required by BPR
String modemMACAddress = "1,6," + computerLease.RELAY_AGENT_REMOTE_ID;

// Now let's provision the modem and the computer in the same batch.
// This can be done because the activation mode of this batch is NO_ACTIVATION.
// More than one device can be operated on in a batch if the activation mode
// does not lead to more than one device being reset.

// NO_ACTIVATION will generate a configuration for the devices. However it will
// not attempt to reset the devices.
// The configuration can be generated because the devices have booted.

// NO_CONFIRMATION is the confirmation mode because we are not attempting to
// reset the modem

// We do not want to reset the modem here because we want to notify the user to
// reset their computer. If we reset the modem in this batch, we will not be
// able to notify the user of anything until the modem has come back online.

// To add a DOCSIS modem:
addDOCSISModem(
modemMACaddress,                // macAddress: derived from computer lease
      null,                     // FQDN: not used in this example
      "0123-45-6789",           // ownerID: here, account number from billing system
      "Silver",                 // ClassOfService
      "provisionedCM",          // DHCP Criteria: Network Registrar uses this to
                                // select a modem lease granting provisioned IP address
      null,                     // CPE DHCP Criteria: not used in this example
      null);                    // properties: not used
```

```
// Create a Map for the computer's properties

Map properties;

// Setting the property IPDeviceKeys.MUST_BE_BEHIND_DEVICE on the computer
// ensures that when the computer boots it will only receive
// its provisioned access when it is behind the given device. If it is not
// behind the given device, it will receive default access (unprovisioned)

properties.put(IPDeviceKeys.MUST_BE_BEHIND_DEVICE, modemMACaddress);


addComputerByIPAddress(
      "10.0.14.37",             // ipAddress: restricted access lease, from Network
                                // Registrar DHCP
      null,                     // FQDN: not used in this example
      "0123-45-6789",           // ownerID: here, account number from billing system
null,                           // ClassOfService: not used in this example
      "provisionedCPE",         // DHCP Criteria: Network Registrar uses this to select
                                // a CPE lease granting a provisioned IP address
      properties);              // properties: as previously set
```

**Step 7** The user interface instructs the subscriber to reboot the computer.

**Step 8** The provisioning client calls resetIPDevice to reboot the modem and gives the modem provisioned access.

```
get-new-batch(AUTOMATIC, NO_CONFIRMATION)

// AUTOMATIC is the activation mode because we are attempting to reset the modem
// so that it receives its new class of service.

// NO_CONFIRMATION is the confirmation mode because we don't want the batch to fail
// if we can't reset the modem. The user may have power cycled the modem when
// they rebooted their computer.

// Send a batch to reset the modem now that the user has been notified to reboot
// their computer.

resetIPDevice(modemMACaddress);
```

**Step 9** When the computer is rebooted, it receives a new IP address, and the cable modem and computer are now both provisioned devices. The computer has access to the Internet through the service provider's network.

# Add a New Computer in Fixed Standard Mode

**Case:** The landlord of an apartment building has four tenants sharing a modem and accessing the service provider's network. The landlord wants to provide Internet access to a new tenant, sharing the building's modem. The new tenant has a computer connected to the modem.

**Desired Outcome:** Use this workflow model to bring a new unprovisioned computer online with a previously provisioned cable modem so that the new computer has the appropriate level of service.

**Step 1** The subscriber turns on the computer, BPR gives it restricted access.

**Step 2** The subscriber starts a browser application on the computer and a spoofing DNS server points the browser to the service provider's registration server (for example, an OSS user interface or a mediator).

**Step 3** The subscriber uses the service provider's user interface to complete the steps required for adding a new computer.

```
get-new-batch(NO_ACTIVATION, NO_CONFIRMATION)

// NO_ACTIVATION will generate a configuration for the computer. However it will
// not attempt to reset the computer (this cannot be done)
// The configuration can be generated because the device has booted.

// NO_CONFIRMATION is the confirmation mode because we are not attempting to
// reset the modem.

Map properties;                  // Map containing the properties for the computer

// Setting the property IPDeviceKeys.MUST_BE_BEHIND_DEVICE on the computer
// ensures that when the computer boots, it will only receive
// its provisioned access when it is behind the given device. If it is not
// behind the given device, it will receive default access (unprovisioned).

properties.put(IPDeviceKeys.MUST_BE_BEHIND_DEVICE, modemMACaddress);

addComputerByIPAddress(
      "10.0.14.37",              // ipAddress: restricted access lease, from Network
                                 // Registrar DHCP
      null,                      // FQDN: not used in this example
      "0123-45-6789",            // ownerID: here, account number from billing system
      null,                      // ClassOfService: not used in this example
      "provisionedCPE",          // DHCP Criteria: Network Registrar uses this to select
                                 // a CPE lease granting a provisioned IP address
      properties);               // properties: as previously set
```

**Step 4** The user interface instructs the subscriber to reboot the computer so that BPR can give the computer its registered service level.

**Step 5** The computer is now a provisioned device with access to the appropriate level of service.

# Disable a Subscriber in Fixed Standard Mode

**Case:** A service provider needs to disable a subscriber from the network due to recurring tardiness in payment.

**Desired Outcome:** Disable an operational modem and computer so that the devices temporarily cannot access the service provider's network.

**Step 1** The service provider’s application disables the subscriber's device information.

**Step 2** The service provider's application uses a provisioning client program to request a list of all of the subscriber's devices from BPR. Then, the service provider's application uses the provisioning client to disable each of the subscriber's devices individually.

```
// The service provider’s application uses a provisioning client to get a list of
// all of the subscriber's devices.
get-new-batch(NO_ACTIVATION, NO_CONFIRMATION)

// NO_ACTIVATION is the activation mode because this is a query.

// NO_CONFIRMATION is the confirmation mode because we are not attempting to
// reset the devices.

getIPDevicesByOwnerID(
      "0123-45-6789");            // Query all the devices for this account number
// For each device in the retrieved list:
// We are only interested in the modems. If we disable network access
// of the modems, we disable network access for all the subscriber’s devices.
// If we are using roaming standard mode, we also change the computer’s
// network access (DHCPCriteria) because the subscriber may still be able
// to access the network from a different modem.

DeviceLoop:
      {
      if (Device.deviceType == DOCSIS_MODEM)
      {
      get-new-batch(AUTOMATIC, NO_CONFIRMATION)
      // AUTOMATIC is the activation mode because we are attempting to
      // reset the modem so that becomes disabled.

      // NO_CONFIRMATION is the confirmation mode because we do not
      // want the batch to fail if we cannot reset the modem.
      // If the modem is off, when it will be disabled when it is turned back on.

      changeDOCSISModemClassOfService(
      Device.MAC_ADDRESS,         // macAddress: unique identifier for this modem
      "Disabled");                // newClassOfService: disable network access
}
}
// end DeviceLoop
```

**Step 3** The subscriber is now disabled.

# Preprovisioned Modem/Self-Provisioned Computer in Roaming Standard Mode

**Case:** A new subscriber contacts the service provider and requests service. The subscriber has a computer installed in a single dwelling unit. The service provider preprovisions all of its cable modems in bulk.

**Desired Outcome:** Bring a preprovisioned cable modem and an unprovisioned computer online so that they have the appropriate level of service. Both devices will be registered.

**Step 1** The service provider chooses a subscriber username and password for the billing system.

**Step 2** The service provider selects the services that the subscriber can access.

**Step 3** The service provider's field technician installs the physical cable to the new subscriber's house and installs the preprovisioned device, connecting it to the subscriber's computer.

**Step 4** The technician turns on the modem and BPR gives it a provisioned IP address.

**Step 5** The technician turns on the computer and BPR gives it a private IP address.

**Step 6** The technician starts a browser application on the computer and points the browser to the service provider's user interface.

**Step 7** The technician uses the service provider's user interface to complete the steps required for registering the computer behind the provisioned cable modem.

```
// To provision a computer:
// MSO admin UI calls the provisioning API to provision a computer.

get-new-batch(NO_ACTIVATION, NO_CONFIRMATION)

// NO_ACTIVATION will generate a configuration for the computer however it will
// not attempt to reset the computer
// The configuration will be able to be generated because the computer has booted.
// NO_CONFIRMATION is the confirmation mode because we are not attempting to
// reset the computer because this cannot be done
addComputerByIPAddress(
     "10.0.14.37",           // ipAddress: restricted access lease, from Network
                             // Registrar DHCP
     null,                   // FQDN: not used in this example
     "0123-45-6789",         // ownerID: here, account number from billing system
     null,                   // ClassOfService: required, but value can be null
     "provisionedCPE",       // DHCP Criteria: Network Registrar uses this to select
                             // a CPE lease
                             // granting provisioned IP address
     null);                  // properties: what has been set earlier
```

**Step 8** The technician restarts the computer and the computer receives a new IP address. The cable modem and the computer are now both provisioned devices. The computer has access to the Internet through the service provider's network.

# Modify an Existing Modem in Roaming Standard Mode

**Case:** A service provider's subscriber currently has Silver service. The subscriber has decided to upgrade to Gold service. The subscriber has a computer installed in the home.

**Note** The intent of this use case is to show how to modify a device. You can apply this example to devices provisioned in modes other than roaming standard.

**Desired Outcome:** Modify an existing modem's class of service and pass that change of service to the service provider's external systems.

**Step 1** The subscriber phones the service provider and requests to have service upgraded. The service provider uses its user interface to change the class of service from Silver service to the Gold.

**Step 2** The service provider's application makes these API calls in BPR:

```
get-new-batch(AUTOMATIC, NO_CONFIRMATION)

// AUTOMATIC will generate a configuration for the device and will
// attempt to reset the device
// The configuration will be able to be generated because the device has booted.

// NO_CONFIRMATION is the confirmation mode because we don't want the batch to fail
// if the reset failed

// This use case is a perfect example of the different confirmation modes that
// could be used instead of NO_CONFIRMATION. These confirmation modes give you
// a method to test whether a configuration was taken by a device. Also,
// these modes will take more time because the batch has to wait for the modem to reset.

changeDOCSISModemClassOfService(
      "1,6,00:11:22:33:44:55",    // macAddress: unique identifier for this modem
      "Gold");                    // newClassOfService
```

**Step 3** The subscriber can now access the service provider's network using the Gold service.

# Delete a Subscriber's Devices in Roaming Standard Mode

**Case:** A service provider needs to delete a subscriber who has discontinued service.

**Note** The intent of this use case is to show how to delete a device. You can apply this example to devices provisioned in modes other than roaming standard.

**Desired Outcome:** Permanently remove all of the subscriber's devices from the service provider's network.

**Step 1** The service provider's user interface discontinues service to the subscriber.

**Step 2** The service provider's application uses the provisioning client program to request a list of all of the subscriber's devices from BPR.

**Step 3** The service provider's application uses the provisioning client program to delete and reset each of the subscriber's devices individually.

```
// MSO admin UI calls the provisioning API to get a list of all the subscriber's
// devices.

get-new-batch(NO_ACTIVATION, NO_CONFIRMATION)

// NO_ACTIVATION is the activation mode because this is a query

// NO_CONFIRMATION is the confirmation mode because we are not attempting to
// reset the devices

     getIPDevicesByOwnerID(
     "0123-45-6789");         // query all the devices for this account number
                              // for each device in the retrieved list:
DeviceLoop:
{

     // get a new batch for each modem being deleted
     get-new-batch(AUTOMATIC, NO_CONFIRMATION)

     // AUTOMATIC is the activation mode because we want
     // to generate a configuration because we are deleting the device.
     // This will also attempt to reset the device

     // NO_CONFIRMATION is the confirmation mode because we do not want
     // the batch to fail if the reset fails

     deleteIPDevice(
     Device.MAC_ADDRESS);     // macAddress: unique identifier for this device
}
// end DeviceLoop.
```

# Self-Provisioned First Time Activation in Promiscuous Mode

**Case:** The subscriber has a computer (with a browser application) installed in a single dwelling unit and has purchased a DOCSIS cable modem.

**Desired Outcome:** Bring a new unprovisioned DOCSIS cable modem and computer online with the appropriate level of service.

**Step 1** The subscriber purchases a DOCSIS cable modem and installs it in the home.

**Step 2** The subscriber turns on the modem, and BPR gives it restricted access.

**Step 3** The subscriber turns on the computer, and BPR gives it restricted access.

**Step 4** The subscriber starts a browser application on the computer and a spoofing DNS server points the browser to the service provider's registration server (for example, an OSS user interface or a mediator).

**Step 5** The subscriber uses the service provider's user interface to complete the steps required for registration, including selecting class of service.

**Step 6** The service provider's user interface passes the subscriber's information to BPR, including the selected classes of service and computer IP address. The subscriber is then registered with BPR.

```
get-new-batch(NO_ACTIVATION, NO_CONFIRMATION)

// NO_ACTIVATION is the activation mode because this is a query

// NO_CONFIRMATION is the confirmation mode because we are not attempting to
// reset the device

// First we query the computer's information to find the modems MAC address
// We use the computers IP address (the web browser received this when the
// subscriber opened the service providers web interface

Map computerLease = getDetailsforIPDevicebyIPAddress(
     "10.0.14.38")           // ipAddress: restricted access computer lease, from
                             // Network Registrar DHCP

// derive the modem MAC address from the computer's network information

// The 1,6, is a standard prefix for an Ethernet device
// The fully qualified MAC address is required by BPR

String modemMACaddress = "1,6," + computerLease.RELAY_AGENT_REMOTE_ID;

get-new-batch(NO_ACTIVATION, NO_CONFIRMATION)

// now let’s provision the modem
// NO_ACTIVATION will generate a configuration for the modem however it will
// not attempt to reset it
// The configuration will be able to be generated because the modem has booted.

// NO_CONFIRMATION is the confirmation mode because we are not attempting to
// reset the modem

// Create a Map for the properties of the modem

Map properties;

// Set the property ModemKeys.PROMISCUOUS_MODE_ENABLED to enable promiscuous
// mode on modem

properties.put(ModemKeys.PROMISCUOUS_MODE_ENABLED, "enabled");

addDOCSISModem(
     modemMACaddress,        // macAddress: the unique MAC for the modem
     null,                   // FQDN: not used in this example
     "0123-45-6789",         // ownerID: here, account number from billing system
     "Silver",               // ClassOfService
```

```
"provisionedCM",         // DHCP Criteria: Network Registrar uses this to
                         // select a modem
                         // lease granting provisioned IP address
"provisionedCPE",        // CPE DHCP Criteria: Network Registrar uses
                         // to grant a provisioned lease to the
                         // CPEs behind this modem
properties);             // properties: the properties of the modem
```

**Step 7** The user interface instructs the subscriber to reboot the computer.

**Step 8** The provisioning client calls resetIPDevice to reboot the modem and gives the modem provisioned access.

```
get-new-batch(AUTOMATIC, NO_CONFIRMATION)

// AUTOMATIC is the activation mode because we are attempting to reset the modem
// so that is receives its new class of service

// NO_CONFIRMATION is the confirmation mode because we don't want the batch to fail
// if we can't reset the modem. The user might have power cycled the modem when
// they rebooted their computer

// send a batch to reset the modem now that the user has been notified to reboot
// their computer

resetIPDevice(modemMACaddress);
```

**Step 9** When the computer is rebooted, it receives a new IP address. The cable modem is now a provisioned device. The computer is not registered with BPR, but it gains access to the Internet through the service provider's network. Computers that are online behind promiscuous modems are still available using the provisioning API.

# Bulk Provision 100 Modems in Promiscuous Mode

**Case:** A service provider wants to preprovision 100 cable modems for distribution by a customer service representative at a service kiosk.

**Desired Outcome:** Modem data for all modems to be distributed to new subscribers is already online, so the customer service representative has a list of modems available for assignment.

**Step 1** Modem MAC address data for new (or recycled) cable modems is collected into a list at the service provider's loading dock.

**Step 2** Modems being assigned to a particular kiosk are bulk loaded into BPR, flagged with the identifier for that kiosk.

**Step 3** When the modems are distributed to new subscribers at the kiosk, the customer service representative will enter new service parameters, and will change the Owner ID field on the modem to reflect the new subscriber's account number.

```
// get a single batch for bulk load or 100 modems

get-new-batch(NO_ACTIVATION, NO_CONFIRMATION)

// The activation mode for this batch should be NO_ACTIVATION
// NO_ACTIVATION should be used in this situation because no network information
// exists for the devices because they have not booted. A configuration cannot
// be generated if no network information is present. And because
// the devices have not booted, they are not online and therefore cannot be reset

// NO_CONFIRMATION is the confirmation mode because we are not attempting to
// reset the devices

// Create a Map for the properties of the modem

Map properties;

// Set the property ModemKeys.PROMISCUOUS_MODE_ENABLED to enable promiscuous
// mode on modem
// This could be done at a system level if promiscuous mode is your
// default provisioning mode

properties.put;(ModemKeys.PROMISCUOUS_MODE_ENABLED, "enabled")

// for each modem MAC-address in list:
ModemLoop:
{
     addDOCSISModem(
          List.MAC_ADDRESS,        // macAddress: unique identifier for this modem
               null,               // FQDN: not used in this example
               kioskIDNumber,      // ownerID: here, each modem is flagged with the
                                   // identifier for the kiosk at which it will
                                   // be assigned to a new subscriber
               "Bronze",           // ClassOfService
               "provisionedCM",    // DHCP Criteria: Network Registrar uses this to
                                   // select a modem
                                   // lease granting provisioned IP address
               "provisionedCPE",   // CPE DHCP Criteria: Network Registrar uses
                                   // to grant a provisioned lease to the
                                   // CPEs behind this modem
               properties);        // properties: the properties of the modem
}
     // end ModemLoop.
```

# Preprovisioned First Time Activation in Promiscuous Mode

**Case:** A new subscriber contacts the service provider and requests service. The subscriber has a computer installed in a single dwelling unit.

**Desired Outcome:** Bring a new unprovisioned cable modem and computer online so that they have the appropriate level of service.

---

**Step 1** The service provider chooses a subscriber username and password for the billing system.

**Step 2** The service provider selects the services that the subscriber can access.

**Step 3** The service provider registers the device using their own user interface.

**Step 4** The service provider's user interface passes information to BPR, such as the modem's MAC address and the class of service. Additionally, the modem gets a CPE DHCP Criteria setting that enables Network Registrar to select a provisioned address for any computers that will be connected behind the modem. The new modem is then registered with BPR.

**Step 5** The service provider's field technician installs the physical cable to the new subscriber's house and installs the preprovisioned device, connecting it to the subscriber's computer.

```
// MSO admin UI calls the provisioning API to pre-provision an HSD modem.

get-new-batch(NO_ACTIVATION, NO_CONFIRMATION)

// The activation mode for this batch should be NO_ACTIVATION
// NO_ACTIVATION should be used in this situation because no network information
// exists for the modem because it has not booted. A configuration cannot
// be generated if no network information is present. And because
// the modem has not booted, it is not online and therefore can not be reset

// NO_CONFIRMATION is the confirmation mode because we are not attempting to
// reset the modem

// Create a Map for the properties of the modem

Map properties;

// Set the property ModemKeys.PROMISCUOUS_MODE_ENABLED to enable promiscuous
// mode on modem

properties.put;(ModemKeys.PROMISCUOUS_MODE_ENABLED, "enabled")

addDOCSISModem(
     "1,6,00:11:22:33:44:55",   // macAddress: technician reads this off the modem
     null,                      // FQDN: not used in this example
     "0123-45-6789",            // ownerID: here, account number from billing system
     "Silver",                  // ClassOfService
     "provisionedCM",           // DHCP Criteria: Network Registrar uses this to
                                // select a modem
                                // lease granting provisioned IP address
     "provisionedCPE",          // CPE DHCP Criteria: Network Registrar uses
                                // to grant a provisioned lease to the
                                // CPEs behind this modem
     properties);               // properties: the properties of the modem
```

**Step 6** The technician turns on the modem and BPR gives it provisioned access.

**Step 7** The technician turns on the computer and BPR gives it provisioned access. The cable modem and the computer are now both provisioned devices. The computer has access to the Internet through the service provider's network.

# Replace an Existing Modem in Promiscuous Mode

**Case:** A service provider wants to replace a broken modem.

**Note** The intent of this use case is to show how to replace a device. You can apply this example to devices provisioned in modes other than promiscuous.

If the computer has the option restricting roaming from one computer to another, and the modem is replaced, the computer's MAC Address for the modem must also be changed.

**Desired Outcome:** Physically replace an existing modem with a new modem without changing the level of service provided to the subscriber.

**Step 1** The service provider changes the MAC address of the existing modem to that of the new modem.

```
get-new-batch(NO_ACTIVATION, NO_CONFIRMATION)

// NO_ ACTIVATION is the activation mode because we will not be able
// to generate a new configuration because the new modem has not booted on the
// network.

// NO_CONFIRMATION is the confirmation mode because we are not trying to
// reset the modem


// To change the MAC address of a DOCSIS modem:
changeDOCSISModemMacAddress (
     "1,6,00:11:22:33:44:55"    // old macAddress: unique identifier for the old modem
     "1,6,11:22:33:44:55:66");  // new macAddress: unique identifier for the new modem
```

**Step 2** The service provider replaces the modem and turns it on. The computer is also turned on.

**Step 3** The cable modem is now a fully provisioned device. The modem and the computer behind it have the appropriate level of service.

# Add a Second Computer in Promiscuous Mode

**Case:** A subscriber wishes to connect a second computer behind an installed cable modem.

**Desired Outcome:** Providing the subscriber's selected service permits connecting multiple CPEs, the subscriber has network access from both connected computers.

**Note** This case does not require calls to the provisioning API.

**Step 1** The subscriber connects a second computer behind the cable modem.

**Step 2** The subscriber turns on the computer.

**Step 3** If the subscriber's selected service permits connecting multiple CPEs, BPR gives the second computer access to the Internet.

# Self-Provisioning First Time Activation with NAT

**Case:** A university has purchased a DOCSIS cable modem with network address translation (NAT) and DHCP capability. The five occupants of the unit each have a computer installed with a browser application.

**Desired Outcome:** Use this workflow model to bring a new unprovisioned cable modem (with NAT) and the computers behind it online so that they have the appropriate level of service.

**Step 1** The subscriber purchases a cable modem with NAT and DHCP capability, and installs it in a multiple dwelling unit.

**Step 2** The subscriber turns on the modem and BPR gives it restricted access.

**Step 3** The subscriber connects a laptop computer to the cable modem, and the DHCP server in the modem provides an IP address to the laptop.

**Step 4** The subscriber starts a browser application on the computer and a spoofing DNS server points the browser to the service provider's registration server (for example, an OSS user interface or a mediator).

**Step 5** The subscriber uses the service provider's user interface to complete the steps required for registration of the modem. The registration user interface detects that the modem is using NAT and registers the modem, making sure that the modem gets a class of service that is compatible with NAT.

```
get-new-batch(NO_ACTIVATION, NO_CONFIRMATION)

// NO_ACTIVATION is the activation mode because this is a query

// NO_CONFIRMATION is the confirmation mode because we are not attempting to
// reset the device
```

```
// First we query the computer's information to find the modems MAC address
// With NAT, the computer is never seen by the Network Registrar DHCP server. Instead,
// the modem has translated the IP address it assigned to the computer, so
// the web browser sees the modem's IP address. When the lease data is
// examined, the MAC address for the device and the relay-agent-remote-id are the
// same, indicating that this is an unprovisioned modem device, which is
// therefore presumed to be performing NAT.

Map deviceDetails = getDetailsForIPDevice(
     "10.0.15.22");              // ipAddress

// MSO client registration program then calls the provisioning API to
// provision the NAT modem (with an appropriate class of service for NAT).

get-new-batch(NO_ACTIVATION, NO_CONFIRMATION)

// NO_ACTIVATION will generate a configuration for the modem however it will
// not attempt to reset it
// The configuration will be able to be generated because the modem has booted.

// NO_CONFIRMATION is the confirmation mode because we are not attempting to
// reset the modem

addDOCSISModem(
     deviceDetails.MAC_ADDRESS,  // macAddress: technician reads this off the modem
     null,                       // FQDN: not used in this example
     "0123-45-6789",             // ownerID: here, account number from billing system
     "Silver",                   // classOfService
     "provisionedCM",            // modem DHCP Criteria: Network Registrar uses
                                 // to select a modem lease granting
                                 // provisioned IP address
     null,                       // CPE DHCP Criteria: not used in this example
     null);                      // properties: not used (but map could contain
                                 // optional data, such as a description field)
```

**Step 6** The user interface instructs the subscriber to reboot the computer.

**Step 7** The provisioning client calls resetIPDevice to reboot the modem and gives the modem provisioned access.

```
get-new-batch(AUTOMATIC, NO_CONFIRMATION)

// AUTOMATIC is the activation mode because we are attempting to reset the modem
// so that it receives its new class of service.

// NO_CONFIRMATION is the confirmation mode because we don't want the batch to fail
// if we can't reset the modem. The user might have power cycled the modem when
// they rebooted their computer

// send a batch to reset the modem now that the user has been notified to reboot their
// computer

resetIPaddress {
deviceDetails.MAC_ADDRESS
};
```

**Step 8** The cable modem is now fully provisioned and the computers behind it have full access to the service provider's network.

**Note** Certain cable modems with NAT may require you to reboot the computer to get the new class of service settings.

If the cable modem and NAT device are separate devices, the NAT device would also have to be registered similar to registering a computer.

# Add a New Computer Behind a Modem with NAT

**Case:** The landlord of an apartment building has four tenants sharing a modem and accessing the service provider's network. The landlord wants to provide Internet access to a new tenant, sharing the building's modem. The modem has NAT and DHCP capability. The new tenant has a computer connected to the modem.

**Desired Outcome:** Use this workflow model to bring a new unprovisioned computer online with a previously provisioned cable modem so that the new computer has the appropriate level of service.

**Note** This case does not require calls to the provisioning API.

**Step 1** The subscriber turns on the computer.

**Step 2** The computer is now a provisioned device with access to the appropriate level of service.

**Note** The provisioned NAT modem hides the computers behind it from the network.

# Preprovisioned First Time Activation for DSTB

**Case:** A new subscriber contacts the service provider and requests service. The service provider determines the type of service the customer wants (for example, interactive digital video and data, or just digital video).

**Desired Outcome:** Use this workflow model to bring a new preprovisioned DSTB online so that it has the appropriate level of service.

**Step 1** The service provider chooses a subscriber username and password for the billing system.

**Step 2** The service provider selects the services that the subscriber can access.

**Step 3** The service provider selects an appropriate DSTB based on the services the subscriber can access. For example, digital video service requires a DVB DSTB, while digital video and interactive data require a DOCSIS/DVB DSTB.

**Step 4** The service provider uses their own user interface to complete the steps required to register the DSTB.

**Step 5** The service provider's user interface passes information to BPR, such as the MAC addresses and classes of service for each component of the DSTB (for example, the DOCSIS cable modem, DVB cable modem, and computer middleware components). The DSTB is then registered with BPR.

```
// MSO admin UI calls the provisioning API to pre-provision DOCSIS modem,
// DVB Modem, and DSTB middleware components of a DSTB.

get-new-batch(NO_ACTIVATION)       // get a single batch for all 2 DSTB components.
                                   // ActivationMode.NO_ACTIVATION
                                   // means that BPR will NOT reboot the modem(s).

addDOCSISModem(
        "1,6,00:11:22:33:44:55",   // macAddress: technician reads this off the modem
        null,                      // FQDN: not used in this example
        "0123-45-6789",            // ownerID: here, account number from billing system
        "Bronze",                  // ClassOfService
        "provisionedCM",           // modemClientClass: Network Registrar uses this to:
                                   //      select a modem
                                   //      lease granting provisioned access
        null,                      // cpeDHCPCriteria: not used in this example
        null);                     // properties: not used (but map could contain
                                   //      optional data, such as a description field)

addDSTB(
        "1,6,99:88:77:66:55:44",   // macAddress: technician reads this off the DSTB
        null,                      // FQDN: not used in this example
        "0123-45-6789",            // ownerID: here, account number from billing system
        null,                      // ClassOfService: not used in this example
        "provisionedDSTB",         // DHCP Criteria; Network Registrar uses this to
                                   //select a CPE lease granting provisioned access
        "1,6,00:11:22:33:44:55",   // relatedDOCSISModemID: MAC address for DOCSIS modem
                                   //      component
        null,                      // DVB component MAC address
        null);                     // properties: not used (but map could contain
                                   //      optional data, such as a description field)
```

**Step 6** The service provider's field technician installs the physical cable to the new subscriber's home and installs the preprovisioned DSTB.

**Step 7** The DSTB boots and BPR gives it provisioned access.

**Step 8** The technician verifies that the DSTB is operating properly for the services that the voice subscriber selected.

# Modify an Existing DSTB

**Case:** A subscriber currently has the Bronze service and wants to upgrade to the Silver service. The subscriber has a DSTB installed in the home.

**Desired Outcome:** Modify an existing DSTB's class of service and pass that change of service to the service provider's external systems.

**Step 1** The subscriber starts the DSTB and points the browser to the service provider's user interface. The subscriber uses the user interface to remove the current Bronze service and to select the Silver service.

**Step 2** The service provider's user interface passes information to BPR, such as the IP address of the DSTB computer middleware and the new class of service.

```
// MSO admin UI calls the provisioning API to change service on a DSTB.
// BPR queries the DSTB to find the related modem(s).
Map DSTB = getDetailsForDSTBByIPAddress(
        "10.0.17.49");                          // query the DSTB
```

**Step 3** BPR writes the changed class of service information for the DSTB's related modem components to the embedded database.

```
// MSO admin UI next calls the provisioning API to change service on
// the related DOCSIS modem components of the DSTB.
get-new-batch(NO_ACTIVATION)          // get a separate batch for each component modem

changeDOCSISModemClassOfService(
         DSTB.relatedDOCISModemID,    // macAddress: the identifier (MAC address) for
                                      //     the related DOCSIS modem component
         "Silver");                   // newClassOfService
```

**Step 4** The service provider instructs the subscriber to reboot the DSTB.

**Step 5** The user can now access the service provider's network using the Silver level of service.

# Replace an Existing DSTB

**Case:** A service provider wants to replace an existing DSTB.

**Desired Outcome:** Physically replace an in-service DSTB with a new one. Provision the new box with the same classes of service as the old DSTB.

**Step 1** The service provider, using its own user interface, accesses the data for the old DSTB. The new DSTB is then preprovisioned with the same class of service information as the old box.

**Note** Because BPR supports only preprovisioning of a DSTB, the service provider must perform step one.

```
// MSO admin UI calls the provisioning API to access data for the old
// DSTB and its related modem components.
get-new-batch()                 // 1st query

Map DSTB = getDetailsForDSTB(
      "1,6,99:88:77:66:55:44"); // query the DSTB

get-new-batch(NO_ACTIVATION)    // get a single batch for all 2 DSTB components.
```

```
                                    //    ActivationMode.NO_ACTIVATION
                                    //    means that BPR will NOT reboot the
                                    //    modem(s).

changeDOCSISModemMACAddress(DSTB.relatedDOCSISModemID,      // MAC address of the
                                                            // DOCSIS component of the
                                                            // old DSTB

                                    "1,6,99:88:77:66:55:35"); // MAC address of the new
                                                            // DSTB

changeDSTBMACAddress("1,6,99:88:77:66:55:44",               // MAC address of the old
                                                            // DSTB.

                                    "1,6,99:88:77:66:55:33"); // MAC address of the new DSTB
```

**Step 2** The service provider sends the new DSTB to the subscriber.

# Remove an Existing DSTB

**Case:** A subscriber wants to remove an existing DSTB currently installed in a subscriber's home.

**Desired Outcome:** Physically remove an existing DSTB from a subscriber's home.

**Step 1** The subscriber removes an existing DSTB from the home.

**Step 2** The subscriber sends the DSTB to the service provider.

**Step 3** The service provider receives the DSTB into inventory. An inventory control person uses their own registration server (for example, an OSS user interface or a mediator) to remove the DSTB from active duty and places it in inventory.

**Step 4** The service provider uses the registration server to remove each component of the DSTB from BPR.

```
// MSO admin UI calls the provisioning API to remove the inactive DSTB and its
// component modems from BPR.

deleteDSTB(
        "1,6,99:88:77:66:55:44",    // macAddress: unique identifier for this DSTB
        true);                      // deleteAllRelatedModems: true, so also delete
                                    //      related DOCSIS and/or DVB modems
```

# Preprovisioned xGCP Capable DOCSIS Modem

**Case:** A new subscriber contacts the service provider and requests voice service.

**Desired Outcome:** Use this workflow model to bring a new xGCP DOCSIS modem (MTA) online so that it has the appropriate level of service to place and receive VoIP calls over the service provider's network.

**Note** A call agent is required for end-to-end provisioning.

**Step 1** The service provider chooses a subscriber username and password for the billing system.

**Step 2** The service provider selects the number of telephone lines to enable on the MTA, and selects the VoIP services and features that the subscriber can access on those lines.

**Step 3** The service provider uses their own user interface to complete the steps required to register the device.

**Step 4** The service provider's user interface passes device information to BPR, such as MAC address, FQDN, classes of service, and VoIP services. The new device is then registered with BPR.

```
// First we add the DOCSIS Modem, then we add xGCP services to it

get-new-batch(NO_ACTIVATION, NO_CONFIRMATION)

// NO_ACTIVATION is the Activation mode because we will not be able
// to generate a new configuration because the new modem has not booted on the
// network.

// NO_CONFIRMATION is the Confirmation mode because we are not attempting to
// reset the modem

addDOCSISModem(
     "1,6,00:11:22:33:44:55",   // macAddress: the MAC address of the modem
     "mta.mso.com",             // FQDN: the FQDN of the modem
     "0123-45-6789",            // ownerID: here, account number from billing system
     "Silver",                  // classOfService
     "provisionedCM",           // DHCP Criteria: Network Registrar uses this to
                                // select a modem
                                // lease granting provisioned IP address
     null,                      // CPE DHCP Criteria: not used in this example
     null);                     // properties: not used in this example

// Now lets add a xGCP service. This is done in the same batch as the modem

addXGCPService(
     "1,6,ff:00:ee:11:dd:22",   // macAddress: the MAC address of the modem
     0,                         // portNumber: Number of line to be enabled.
                                // Line numbers are 0 based
     "ca",                      // cmsQualifier: CMS qualifier for CMS which
                                // will control this line
     "ca1.mso.net",             // cmsFQDN: CMS FQDN for CMS which
                                // will control this line
     5678,                      // cmsPortNumber: CMS Port Number for CMS which
                                // will control this line
     "(978)123-4567",           // telephoneNumber: telephone number for this
                                // line - for informational purposes
     null);                     // properties: not used in this example
```

**Step 5** The service provider's field technician installs the physical cable to the new subscriber's home and installs the pre-registered DOCSIS modem MTA. Analog phones are then connected to the MTA lines enabled through the provisioning workflow.

**Step 6** The MTA boots and BPR gives it provisioned VoIP service through the service provider's network.

# Activating an Additional Telephone Line on an xGCP Capable DOCSIS Modem

**Case:** A service provider needs to provision an additional telephone line on a provisioned xGCP DOCSIS modem (MTA).

**Desired Outcome:** The subscriber can place and receive VoIP calls over an additional line using the service provider's network.

**Step 1** The service provider uses their own user interface to enable an additional telephone line, as well as the VoIP services and features the subscribers can have access to for that new line.

**Step 2** The service provider's user interface passes the information to BPR. BPR updates the device information and resets the device.

```
get-new-batch(AUTOMATIC, NO_CONFIRMATION)

// AUTOMATIC is the Activation mode because we are attempting to reset the modem
// so that is receives its new phone line

// NO_CONFIRMATION is the Confirmation mode because we don't want the batch to fail
// if we can't reset the modem.

addXGCPService(
     "1,6,ff:00:ee:11:dd:22",      // macAddress: the MAC address of the modem
     1,                            // portNumber: Number of line to be enabled.
     "ca",                         // cmsQualifier: CMS qualifier for CMS which
                                   // will control this line
     "ca1.mso.net",                // cmsFQDN: CMS FQDN for CMS which
                                   // will control this line
     5678,                         // cmsPortNumber: CMS Port Number for CMS which
                                   // will control this line
     "(978)123-4567",              // telephoneNumber: telephone number for this
                                   // line - for informational purposes
     null);                        // properties: not used in this example
```

**Step 3** The subscriber can now place and receive telephone calls over the new telephone line.

# Disable an Existing Telephone Line on an xGCP Capable DOCSIS Modem

**Case:** A service provider needs to disable a telephone line for a subscriber who has voluntarily decided to discontinue service on the line, or due to recurring tardiness in payment.

**Desired Outcome:** Disable an operational telephone line.

**Step 1** The service provider uses their own user interface to disable a previously provisioned telephone line.

**Step 2** The service provider's user interface, acting as a BPR client, passes the information to BPR. BPR updates the device information and resets the device.

```
get-new-batch(AUTOMATIC, NO_CONFIRMATION)

// AUTOMATIC is the Activation mode because we are attempting to reset the modem
// so that a phone line is disabled

// NO_CONFIRMATION is the Confirmation mode because we don't want the batch to fail
// if we can't reset the modem.

// To delete service for a xGCP DOCSIS modem
deleteXGCPService(
"1,6,ff:00:ee:11:dd:22",      // macAddress: the MAC address of the modem
0);                           // port number
```

**Step 3** After being rebooted, the subscriber cannot place and receive telephone calls on the disabled phone line.

# Generate a Configuration

**Case:** The service provider has changed the properties of the DOCSIS Gold class of service. Now a configuration has to be generated for all the DOCSIS devices that have the Gold class of service.

**Desired Outcome:** Generate a configuration for a device using provisioning the API.

**Step 1** Get all devices with the DOCSIS class of service Gold.

```
get-new-batch(NO_ACTIVATION, NO_CONFIRMATION)

getIPDevicesByClassOfService("Gold", // COS name
                  ClassOfServiceType.DOCSIS, // Specifies DOCSIS COS Type
                  0) // Return all matches
```

**Step 2** Generate configuration for all the devices in the list created in Step 1.

```
get-new-batch(NO_ACTIVATION, NO_CONFIRMATION)

generateConfiguration(deviceID );  // Device ID from the list returned in Step 1
```

# Add DHCP Criteria

**Case:** A registered cable modem needs an IP address from a different Network Registrar scope.

**Desired Outcome:** A provisioning client adds the DHCP criteria, and the cable modem receives an appropriate IP address.

**Step 1** Create a DHCP criteria with the desired scope selection tags.

```
get-new-batch(NO_ACTIVATION, NO_CONFIRMATION)

addDHCPCriteria( "modemCriteria", // DHCP Criteria Name
                              null, // Client Class
                              "provisionedModemTag", // Include Selection Tag
                              "provisionedCPETag", // Exclude Selection Tag
                              null // Properties
                              );
BatchStatus stat = batch.post();

if (!stat.isError())
}
                              //Added DHCP criteria successfully.
}
```

**Step 2** Change the DOCSIS modem's DHCP criteria to that specified in modemCriteria. (See Step 1.)

```
get-new-batch(NO_ACTIVATION, NO_CONFIRMATION)

// This use case assumes that the DOCSIS modem has been previously added to the database

changeDOCSISModemDHCPCriteria(deviceID, // Modem's MAC address or FQDN
                              "modemCriteria");

BatchStatus stat = batch.post();

if (!stat.isError())
{

          // Configuration generation and device reset is successful.
}
```

**Step 3** The modem gets an IP address from the scope targeted by modemCriteria.

# Change System Defaults

**Case:** A service provider wants to enable Promiscuous mode for all CPEs.

**Desired Outcome:** A service provider can add DOCSIS modems and support promiscuous CPEs by enabling Promiscuous mode on a system level basis.

**Step 1** Use the changeSystemDefaults() command and enable the PROMISCUOUS_MODE_ENABLED property on the system level.

```
get-new-batch(NO_ACTIVATION, NO_CONFIRMATION)

Map newProp = new HashMap();

// This enables Promiscuous mode on a system level.
// You can override this value by setting PROMISCUOUS_MODE_ENABLED to false
// on a per device basis.

NewProp.add(ModemKeys.PROMISCUOUS_MODE_ENABLED,"enabled");

changeSystemDefaults(newProp, // Adds/Replaces these properties
                             null ); // No properties to remove
```

**Note** You can override these settings by disabling the Promiscuous mode on individual modems.

# Log Device Deletions

**Case:** A service provider has multiple provisioning clients and wants to log device deletions.

**Desired Outcome:** When any provisioning client deletes a device, the provisioning client logs an event in one place.

**Step 1** Create a listener for the device deletion event. This class must extend the DeviceAdapter abstract class or, alternatively, implement the DeviceListener interface. This class must also override the deletedDevice(DeviceEvent ev) method in order to log the event.

```
public DeviceDeletionLogger                    //Extend the DeviceAdapter class.
    extends DeviceAdapter
{
    public void deletedDevice(DeviceEvent ev)   //Override deletedDevice.
    {
          logDeviceDeletion(ev.getDeviceID())   //Log the deletion.
```

```
    }
}
```

**Step 2** Register the listener and the qualifier for the events using the PACEConnection interface.

```
DeviceDeletionLogger deviceDeletionLogger = new DeviceDeletionLogger();
qualifier = new DeviceEventQualifier();
qualifier.setDeletedDevice ();
conn.addDeviceListener(deviceDeletionLogger, qualifier);
```

**Step 3** When a device is deleted from the system, the event is fired, and the listener is notified.

# Monitor a Connection to the RDU

**Case:** A service provider is running a single provisioning client and wants notification if the connection between the provisioning client and the RDU breaks.

**Desired Outcome:** The Event interface notifies the service provider if the connection breaks.

**Step 1** Create a listener for the messaging event. This class must extend the MessagingAdapter abstract class or, alternatively, implement the MessagingListener interface. This class must override the connectionStopped(MessagingEvent ev) method.

```
// Extend the service provider's Java program using the provisioning client to receive
// Messaging events.

public MessagingNotifier                                  //Extend the MessagingAdapter
        extends MessagingAdapter                          // class.
{
   public void connectionStopped(MessagingEvent ev)     //Override connectionStopped.
   {
        doNotification(ev.getAddress(), ev.getPort());  //Do the notification.
   }
}
```

**Step 2** Register the listener and the qualifier for the events using the PACEConnection interface.

```
MessagingQualifier qualifier = new MessagingQualifier();
qualifier.setAllPersistentConnectionsDown();

MessagingNotifier messagingNotifier = new MessagingNotifier();
conn.addMessagingListener(messagingNotifier, qualifier);
```

**Step 3** If a connection breaks, the event is fired, and the listener is notified.

# Log Batch Completions

**Case:** A service provider has multiple provisioning clients and wants to log batch completions.

**Desired Outcome:** When any provisioning client complete a batch, an event is logged in one place.

**Step 1** Create a listener for the event. This class must extend the BatchAdapter abstract class or implement the BatchListener interface. This class must override the completion(BatchEvent ev) method in order to log the event.

```
public BatchCompletionLogger                                    //Extend the BatchAdapter
        extends BatchAdapter                                    // class.
{
    public void completion(BatchEvent ev)                       //Override completion.
    {
        logBatchCompletion(ev.BatchStatus().getBatchID());      //Log the completion.
    }
}
```

**Step 2** Register the listener and the qualifier for the events using the PACEConnection interface.

```
BatchCompletionLogger batchCompletionLogger = new BatchCompletionLogger();
QualifyAll qualifier = new QualifyAll();
conn.addBatchListener(batchCompletionLogger, qualifier);
```

**Step 3** When a batch completes, the event is fired, and the listener is notified.

# Get Detailed Information for a Device

**Case:** A service provider wants to allow an administrator to view detailed information for a particular device.

**Desired Outcome:** The service provider's administrative application displays all known details about a given device, including MAC address, lease information, provisioned status of the device, and the device type (if known).

**Step 1** The administrator enters either the MAC address or the IP address for the device being queried into the service provider's administrative user interface.

**Step 2** BPR queries the embedded database for the details for that device.

```
// MSO admin UI calls the provisioning API to query the details for the
// requested device. Query may be performed based on MAC address or IP address,
// depending on what is known about the device.
Map deviceDetails = getDetailsForIPDevice(
         "1,6,00:11:22:33:44:55");   // macAddress: unique identifier for the device
// - OR -
Map deviceDetails = getDetailsForIPDeviceByIPAddress(
          "10.0.17.129");             // ipAddress
```

**Step 3** The service provider's application then presents a detail page of device data, which can display everything that is known about the requested device. If the device has been connected to the service provider's network, this data will include lease information (for example, IP address, relay agent identifier). The data indicates whether or not the device has been provisioned, and, if it has, the data also includes the device type.

```
// extract device detail data from the map
String deviceType = (String)deviceDetails.get(DEVICE_TYPE);
String macAddress = (String)deviceDetails.get(MAC_ADDRESS);
String ipAddress = (String)deviceDetails.get(IP_ADDRESS);
String relayAgentID = (String)deviceDetails.get(RELAY_AGENT_ID);
Boolean isProvisioned = (Boolean)deviceDetails.get(IS_PROVISIONED);

// The admin UI now formats and prints the detail data to a view page
```

# Retrieve Devices Matching a Vendor Prefix

**Case:** A service provider wants to allow an administrator to view data for all devices matching a particular vendor prefix.

**Desired Outcome:** The service provider's administrative application returns a list of devices matching the requested vendor prefix.

**Step 1** The administrator enters the substring matching the desired vendor prefix into the service provider's administrative user interface.

**Step 2** BPR queries the embedded database for a list of all MAC addresses for the devices that match the requested vendor prefix.

```
// MSO admin UI calls the provisioning API to query all devices for a specified
// vendor prefix.
List deviceList = getIPDevicesByMACAddress(
     "1,6,ff:00:ee:*",             // macAddressPattern: the requested vendor prefix
     10000);                       // maximumReturned:
```


# Provisioning API Reference

The provisioning API consists of these packages:

- com.cisco.provisioning.cpe—Contains the interfaces and methods used to connect to the provisioning API command engine (PACE) and to manage batch operations.
- com.cisco.provisioning.cpe.api—Contains the interfaces and methods used to provision and manage customer premises equipment (CPE), such as high-speed data (HSD) devices, digital set-top box (DSTB) devices, voice capable (xGCP) devices, and computers.
- com.cisco.provisioning.cpe.constants—Contains the interfaces and constants that interpret status information and retrieve values associated with the device types that BPR provisions.
- com.cisco.provisioning.cpe.events—Contains the interfaces and methods used to implement events.

The sections that follow introduce each of the BPR provisioning API packages.

**Note** You can access complete descriptions of the provisioning API packages through the JavaDoc packaged with the BPR product. For a list of methods deprecated since the CSRC DPR product release, see Appendix B, "Deprecated Methods."

## Hierarchy

Table A-1 shows the cisco.com.provisioning.cpe hierarchy.

***Table A-1 cisco.com.provisioning.cpe Hierarchy***

| **com.cisco.provisioning.cpe** | |
|---|---|
| **Interfaces** | Batch |
| | BatchStatus |
| | CommandStatus |
| | CSRCAPI |
| | CSRCAPIStatus |
| | PACEConnection |
| | ProvAPI |
| | ProvAPIStatus |

*Table A-1 cisco.com.provisioning.cpe Hierarchy (continued)*

| com.cisco.provisioning.cpe | |
|---|---|
| **Classes** | ActivationMode |
| | APINames |
| | ConfirmationMode |
| | DataType |
| | PACEConnectionFactory |
| | PublishingMode |
| | SNMPVersion |
| **Exceptions** | AlreadyPostedException |
| | APINotFoundException |
| | BaseException |
| | BaseRunTimeException |
| | BPRAuthenticationException |
| | PACEConnectionException |
| | ProvisioningException |

*Table A-2 com.cisco.provisioning.cpe.api Hierarchy*

| com.cisco.provisioning.cpe.api | |
|---|---|
| **Interfaces** | Computer |
| | Configuration |
| | CustomCPE |
| | DeviceSearch |
| | DOCSIS |
| | DSTB |
| | Provisioning |
| | XGCP |

*Table A-3 com.cisco.provisioning.cpe.constraints Hierachy*

| com.cisco.provisioning.cpe.constants | |
|---|---|
| **Interfaces** | BatchStatusCodes |
| | ClassOfServiceKeys |
| | ClassOfServiceType |
| | CNRDetailsKeys |
| | CNRExtensionSettingKeys |
| | CNRServersKeys |
| | CommandStatusCodes |
| | DeviceDetailsKeys |
| | DeviceDisruptionStatus |
| | DeviceTypeValues |
| | DHCPCriteriaKeys |
| | DHCPOptionKeys |
| | DocsisDefaultKeys |
| | ETTHProfileKeys |
| | IPDeviceKeys |
| | LicenseCodes |
| | LicenseDataKeys |
| | ModemKeys |
| | PublishDetailsKeys |
| | RDUDetailsKeys |
| | SearchType |
| | ServerDefaultsKeys |
| | SNMPPropertyKeys |
| | TechnologyDefaultsKeys |
| | UserDetailsKeys |
| | VOIPServiceDetailsKeys |
| | XGCPCommandStatusCodes |
| | XGCPProvisioningKeys |

| com.cisco.provisioning.cpe.events | |
|---|---|
| **Interfaces** | BatchEvent |
| | BatchListener |
| | COSEvent |
| | COSListener |
| | DeviceEvent |
| | DeviceListener |
| | DHCPCriteriaEvent |
| | DHCPCriteriaListener |
| | ExternalFileEvent |
| | ExternalFileListener |
| | MessagingEvent |
| | MessagingListener |
| | ProvAPIEvent |
| | ProvAPIEventListener |
| | ProvAPIEventManager |
| | ProvGroupEvent |
| | ProvGroupListener |
| | SystemConfigEvent |
| | SystemConfigListener |

| com.cisco.provisioning.cpe.events (continued) | |
|---|---|
| **Classes** | BatchAdapter |
| | BatchEventQualifier |
| | COSAdapter |
| | COSEventQualifier |
| | DeviceAdapter |
| | DeviceEventQualifier |
| | DHCPCriteriaAdapter |
| | DHCPCriteriaEventQualifier |
| | ExternalFileAdapter |
| | ExternalFileEventQualifier |
| | MessagingAdapter |
| | MessagingQualifier |
| | ProvGroupAdapter |
| | ProvGroupEventQualifier |
| | Qualifier |
| | QualifyAll |
| | ServerConfigEventType |
| | SystemConfigAdapter |
| | SystemConfigEventQualifier |
| | SystemConfigEventType |
| **Exceptions** | RegistrationFailedException |

# com.cisco.provisioning.cpe Package

The com.cisco.provisioning.cpe package contains the interfaces and methods used to connect to the provisioning API command engine (PACE) and to manage batch operations. Table A-4 lists the com.cisco.provisioning.cpe interfaces.

***Table A-4 The com.cisco.provisioning.cpe Package Interfaces***

| Interface | Description |
|---|---|
| Batch | Provides access to the provisioning API commands to be executed. |
| BatchStatus | Provides the return type for a batch operation. |
| CommandStatus | Provides a standard return type for the individual commands submitted from a batch operation. |
| CSRCAPI | Is a marker interface that all BPR API interfaces implement. |
| CSRCAPIStatus | Is implemented by all interfaces that describe messages returned from the provisioning server to the client. |
| PACEConnection | Contains the commands used to create and manage communication with the provisioning API command engine (PACE). |

*Table A-4 The com.cisco.provisioning.cpe Package Interfaces*

| Interface | Description |
|---|---|
| ProvAPI | Is a marker interface that is implemented by all other BPR API interfaces. |
| ProvAPIStatus | Is a status interface that all BPR API interfaces implement. It describes messages returned from PACE to the client program. |

Table A-5 lists the com.cisco.provisioning.cpe classes.

*Table A-5 The com.cisco.provisioning.cpe Classes*

| Class | Description |
|---|---|
| ActivationMode | Provides an enumeration of batch activation request states. |
| APINames | Provides an enumeration of the BPR named API interfaces and is used in conjunction with the Batch.getAPI() command. |
| ConfirmationMode | Provides an enumeration of batch confirmation request states. |
| DataType | Provides an enumeration of DataTypes that can be specified as part of a custom property definition. |
| PACEConnectionFactory | Returns a singleton instance of the PACE connection using peer objects. |
| PublishingMode | Provides an enumeration of batch Publishing flags. |
| SNMPVersion | Provides an enumeration of SNMP versions. |

Table A-6 lists the cisco.com.provisioning.cpe package exceptions.

*Table A-6 The com.cisco.provisioning.cpe Package Exceptions*

| Exception | Description |
|---|---|
| ProvisioningException | Used to signal that PACE could not process the posted batch. |

Table A-7 lists the com.cisco.provisioning.cpe exceptions.

*Table A-7 The com.cisco.provisioning.cpe Package Runtime Exceptions*

| Exception | Description |
|---|---|
| AlreadyPostedException | Used to indicate that this batch has already been posted. |
| APINotFoundException | Used to signal an attempt to retrieve an API instance S that is not one of the APIs enumerated by APINames class. |
| BaseException | Allows for localization and exception cascading. This means adding additional explanations to an exception and rethrowing the enhanced exception. |
| BaseRuntimeException | Defines an exception that allows for localization and exception cascading. Exception cascading is adding additional explanations to an exception and rethrowing the enhanced exception. |

*Table A-7 The com.cisco.provisioning.cpe Package Runtime Exceptions*

| Exception | Description |
|---|---|
| BPRAuthenticationException | Used to indicate an authentication failure while getting an instance of PACEConnection through the PACEConnectionFactory. |
| PACEConnectionException | Used to indicate an error in getting an instance of PACEConnection through the PACEConnectionFactory. |

# Batch Interface

The Batch interface gives access to all the provisioning API methods in BPR. Each batch of commands has a unique identifier; therefore, you can post a batch only once. If you try to add calls to a batch that has already been posted or to repost a batch, BPR generates a runtime exception. Table A-8 lists the Batch methods.

*Table A-8 Batch Methods*

| Method | Description |
|---|---|
| addAPICall() | Adds an API Call object to the list of method calls in a batch.<br>**Note** This method is for BPR internal use only. BPR API clients should not call this method. |
| createdByClient() | Indicates whether the Batch was created by the client API. |
| forceBatchReliable() | Forces the batch to be reliable. This indicates that the batch will complete even if the RDU fails while the batch is in the middle of, or waiting to be, processed. |
| getActivationMode() | Obtains the Activation mode associated with this batch. |
| getAllAPIs() | Instantiates and returns all available APIs (which ones are available are determined by customer licensing). |
| getAPI(APINames api) | Instantiates and returns requested API, or throws a runtime exception if the user is not licensed for the requested API. |
| getAPICall(int index) | Gets an APICall object from the batch's list of method calls.<br>**Note** This method is for BPR internal use only. BPR API clients should not call this method. |
| getAPICallCount() | Obtains the count of API calls stored in the batch. |
| getBatchID() | Gets the batch identifier associated with the batch. |
| getConfirmationMode() | Obtains the Confirmation mode associated with this batch. The returned value is an enumerated type from the Confirmation mode class. |
| getProvAPI() | Instantiates and returns requested API, or throws a runtime exception if the user is not licensed for the requested API. |
| getPublishingMode() | Obtains the PublishingMode (enum) associated with this batch. |
| isBatchReliable() | Indicates whether the Batch will be completed even if the RDU goes down while the batch is processing or waiting to be processed. |
| post() | Posts this batch of commands for execution. |

*Table A-8 Batch Methods*

| Method | Description |
|---|---|
| post(long timeout) | Posts this batch of commands for execution. |
| postNoConfirmation() | Posts this command batch for execution. |
| postWithConfirmation() | Posts this command batch for execution. |
| wasPosted() | Indicates whether the batch was ever posted. |

# BatchStatus Interface

The BatchStatus interface provides the standard return type for a batch operation. Table A-9 lists the BatchStatus methods.

*Table A-9 BatchStatus Methods*

| Method | Description |
|---|---|
| getBatchID() | Returns the associated batch identifier for this status request. |
| getCommandCount() | Gets the number of commands in a batch that were processed. |
| getCommandStatus() | Returns the command status for the specified command or returns null if there is no corresponding command. |
| getFailedCommandIndex() | Gets the index for the specified command that failed. Returns a value of -1 if no command failed. |
| getFailedCommandStatus() | Gets the command status return for the failed command or returns null value if no command failed. |
| isWarning() | A helper method. |
| wasBatchReliable() | Returns true if the batch was treated as reliable by the server. |

# CommandStatus Interface

The Command Status interface provides the standard return type for the individual operations submitted in a batch, including command error status. For methods that return data to the caller, this interface also provides the data type code and the data itself.

The CommandStatus.getDataTypeCode() function returns status values to indicate that a command returned these data types:

- Byte array data
- List data
- Map of key/value data
- Null
- String data
- Void (the command did not return any data)

Table A-10 lists the CommandStatus methods.

*Table A-10 CommandStatus Methods*

| Method | Description |
|---|---|
| commandReturnsData() | Indicates whether a command returns data. |
| getData() | Returns the requested data for this command status. |
| getDataTypeCode() | Returns the data type code for this command status. |

# CSRCAPI Interface

The CSRCAPI interface is a marker interface that all the com.cisco.provising.cpe.api interfaces implement. Its subinterfaces are:

- Computer
- Configuration
- CustomCPE
- Device search
- DOCSIS
- DSTB
- Provisioning
- XGCP

# ProvAPIStatus Interface

The ProvAPIStatus interface is implemented by all interfaces that describe messages returned from PACE to the client program. This interface provides this status information:

- Status type of the batch operation
- Status type of the command
- Unknown status type
- Provisioning exception type

Table A-11 lists the ProvAPIStatus methods.

*Table A-11 ProvAPIStatus Methods*

| Method | Description |
|---|---|
| getBatchID() | Returns the batch identifier. |
| getErrorMessage() | Obtains an error message that describes a problem encountered in processing. |
| getStatusCode() | Obtains the status code for the results of processing. |
| getStatusType() | Returns the type of interface that the implementing object represents. |
| isError() | Determines whether a status code is an error message. |
| isSystemError() | Determines whether a status code is a system error message. |

# PACEConnection Interface

The PACEConnection interface contains the commands used to create and manage communication with the provisioning API command engine (PACE). Table A-12 lists the PACEConnection methods.

***Table A-12 PACEConnection Methods***

| Method | Description |
|---|---|
| dropConnection() | Drop an RDU connection. |
| getErrorString() | Return the error string from the last call to addListener() or removeListener(). |
| getHost() | Returns the hostname of the RDU to which you are connecting. |
| getPort() | Returns the port number of the RDU to which you are connecting. |
| getReturnString() | Return the status of the last Connection method invocation. |
| isAlive() | Returns true if the connection to the RDU is open, otherwise false. |
| joinBatch(java.lang.String batchID) | Joins a batch that has been executed. |
| joinBatch(java.lang.String batchID, long timeout) | Joins a batch that has been executed. |
| newBatch() | Gets a new batch with a guaranteed globally-unique batch identifier. |
| newBatch(ActivationMode activation) | Gets a new batch with a guaranteed globally-unique batch ID. |
| newBatch(ActivationMode activation, ConfirmationMode confirmation) | Gets a new batch with a guaranteed globally-unique batch ID. |
| newBatch(ActivationMode activation, ConfirmationMode confimation, PublishingMode publishing) | Gets a new batch with a guaranteed globally-unique batch ID. |
| newBatch(ActivationMode activation, PublishingMode publishing) | Gets a new batch with a guaranteed globally-unique batch ID. |
| newBatch(PublishingMode publishing) | Gets a new batch with a guaranteed globally-unique batch ID. |
| newBatch(String batchID) | Gets a new batch, using a user-supplied batch ID, which must be unique (or the batch will fail to execute when it is posted). |
| newBatch(String batchID, ActivationMode activation) | Gets a new batch, using a user-supplied batch ID, which must be unique (or the batch will fail to execute when it is posted). |
| newBatch(String batchID, ActivationMode activation, ConfirmationMode confirmation) | Gets a new batch, using a user-supplied batch ID, which must be unique (or the batch will fail to execute when it is posted). |
| newBatch(java.lang.String batchID, ActivationMode activation, ConfirmationMode confirmation, PublishingMode publishing) | Gets a new batch, using a user-supplied batch ID, which must be unique (or the batch will fail to execute when it is posted). |

*Table A-12 PACEConnection Methods (continued)*

| Method | Description |
|---|---|
| newBatch(java.lang.String batchID, ActivationMode activation, PublishingMode publishing) | Gets a new batch, using a user-supplied batch ID, which must be unique (or the batch will fail to execute when it is posted). |
| newBatch(java.lang.String batchID, PublishingMode publishing) | Gets a new batch, using a user-supplied batch ID, which must be unique (or the batch will fail to execute when it is posted). |
| openConnection() | Opens a connection to the RDU. |
| postBatch(Batch batch) | Posts a batch of commands for execution. |
| postBatch(Batch batch, long timeout) | Posts a batch for execution. |
| postBatchNoStatus(Batch batch) | Posts a batch for execution. No status is returned for this batch. |
| releaseConnection() | Releases an RDU connect. |

# ActivationMode Class

Use the Activation mode class for an enumeration of batch activation request states. This class has three fields:

- AUTOMATIC—Perform all necessary steps for activation of the devices in the batch, including device disruption.
- NO_ACTIVATION—Do not activate the devices in the batch. A new configuration is generated.

Table A-13 lists the ActivationModeClass methods.

*Table A-13 ActivationMode Class Methods*

| Method | Description |
|---|---|
| getNamed() | Returns the ActivationMode object with the selected name, or it returns null if the name does not exist. |
| getValued() | Returns the ActivationMode object with the selected value, or it returns null if the value does not exist. |

# APINames Class

Use this class, in conjunction with the Batch.getAPI() method, for an enumeration of the BPR named API interfaces.

Table A-14 lists the fields for the API names class.

***Table A-14 APINames Class Fields***

| Field | Description |
|---|---|
| Computer | References the com.cisco.provisioning.cpe.api.Computer interface. |
| Configuration | References the com.cisco.provisioning.cpe.api.Configuration interface. |
| CustomCPE | References the com.cisco.provisioning.cpe.api.CustomCPE interface. |
| DeviceSearch | References the com.cisco.provisioning.cpe.api.DeviceSearch interface. |
| DOCSIS | References the com.cisco.provisioning.cpe.api.DOCSIS interface. |
| DSTB | References the com.cisco.provisioning.cpe.api.DSTB interface. |
| Provisioning | References the com.cisco.provisioning.cpe.api.Provisioning interface. |
| XGCP | References the com.cisco.provisioning.cpe.api.XGCP interface. |

Table A-15 lists the APINames methods.

***Table A-15 APINames Methods***

| Method | Description |
|---|---|
| getNamed() | Returns the APIName object with the selected name, or it returns null value if the name does not exist. |
| getValued() | Returns the APIName object with the selected value, or it returns null value if the value does not exist. |

# ConfirmationMode Class

The ConfirmationMode class is used for an enumeration of batch confirmation request states. Table A-16 lists the ConfirmationMode class fields.

***Table A-16 ConfirmationMode Class Fields***

| Class | Description |
|---|---|
| CUSTOM_CONFIRMATION | If disruption error status is set, the batch status code will be set to BatchStatusCodes.BATCH_FAILED_DISRUPTION, and the whole batch will roll back. |
| NO_CONFIRMATION | If disruption fails, does not confirm device reboot and returns with a BatchStatusCode value of BATCH_WARNING. NO_CONFIRMATION is the default. |

Table A-17 lists the ConfirmationMode methods.

***Table A-17 ConfirmationMode Methods***

| Method | Description |
|---|---|
| getNamed() | Returns the APIName object with the selected name, or it returns null value if the name does not exist. |
| getValued() | Returns the APIName object with the selected value, or it returns null value if the value does not exist. |

# DataType Class

The DataType class provides an enumeration of the data types that can be specified as part of a custom property definition. Table A-18 identifies the fields implemented by the DataType class and Table A-19 lists the DataType methods.

***Table A-18 DataType Fields***

| Field | Description |
|---|---|
| BOOLEAN | References an object of type Boolean. |
| BYTE | References an object of type Byte. |
| BYTE_ARRAY | References an object of type Byte_Array. |
| CHARACTER | References an object of type Character. |
| DOUBLE | References an object of type Double. |
| FLOAT | References an object of type Float. |
| INTEGER | References an object of type Integer. |
| LIST | References an object of type List. |
| LONG | References an object of type Long. |
| MAP | References an object of type Map. |
| SHORT | References an object of type Short. |
| STRING | References an object of type String. |

***Table A-19 DataTypeClass Methods***

| Method | Description |
|---|---|
| getNamed() | Returns the DataType object with the selected name (or null if it does not exist). |
| getValued() | Returns the DataType object with the selected value (or null if it does not exist). |
| readExternal() | Identifies an externalizable interface. |
| writeExternal() | Identifies an externalizable interface. |

# PACEConnectionFactory Class

Use the PACE connection factory class to return singleton instances using peer objects. Table A-20 lists the PACEConnectionFactory methods.

***Table A-20 PACEConnectionFactory Methods***

| Method | Description |
|---|---|
| containsInstance(InetAddress address, int port) | Returns true if a singleton instance exists on the given port and host. |
| containsInstance(String host, int port) | Returns true if a singleton instance exists on the given port and host. |
| getInstance(InetAddress address, int port) | Returns the singleton instance of the anonymous PACEConnection on the given port and host. |
| getInstance(InetAddress address, int port, String username, String password) | Returns the singleton instance of the PACEConnection on the given port and host. |
| getInstance(com.cisco.csrc.messages.Peer peer) | Returns the singleton instance of the PACEConnection on the given Peer. |
| getInstance(com.cisco.csrc.messages.Peer peer, String username, String password) | Returns the singleton instance of the PACEConnection on the given Peer. |
| getInstance(String host, int port) | Returns the singleton instance of the PACEConnection on the given port and host. |
| getInstance(String host, int port, String username, String password) | Returns the singleton instance of the PACEConnection on the given port and host. |
| getInstance(String host, int port, String username, String password, boolean authenticateImmediately) | Returns the singleton instance of the PACEConnection on the given port and host. |

# SNMPVersion Class

The SNMPVersion class provides an enumeration of SNMP versions. Table A-21 lists the SNMPVersion methods.

The SNMPVersion class implements this field:

- V1—References SNMP version 1.

***Table A-21 SNMPVersion Methods***

| Method | Description |
|---|---|
| getNamed() | Returns the SNMPVersion object with the selected name (or null if the object does not exist). |
| getValued() | Returns the SNMPVersion object with the selected value (or null if the object does not exist). |

## Exceptions

The com.cisco.provisioning.cpe package implements these exceptions to handle processing of errors and other abnormal conditions:

- ProvisioningException—Thrown as a result of calling Batch.post(), and signals that PACE could not process the posted batch due to catastrophic failure.
- BaseException—A BaseException is a BPR defined exception that allows for localization and exception cascading. Exception cascading is adding additional explanations to an exception and rethrowing the enhanced exception.
- BaseRunTimeException—A BaseRunTimeException is a BPR defined exception that allows for localization and exception cascading. Exception cascading is adding additional explanations to an exception and rethrowing the enhanced exception.

The com.cisco.provisioning.cpe package implements these runtime exceptions to handle processing errors and other abnormal conditions:

- AlreadyPostedException—Thrown when an attempt is made to repost an already posted batch. The caller must get a new batch instance.
- APINotFoundException—Thrown when an attempt is made to retrieve an API Instance object that is not one of the APIs enumerated by APINames.
- PACEConnectionException—This exception can occur when getting an instance of PACEConnection through the PACEConnectionFactory.
- BPRAuthenticationException—Thrown when getting an instance of PACEConnection through the PACEConnectionFactory.

# com.cisco.provisioning.cpe.api Package

The com.cisco.provisioning.cpe.api package contains the interfaces and commands used to manage these device types:

- High-speed data (HSD) devices
- Digital set-top box (DSTB) devices
- Voice capable Gateway Control Protocol (xGCP) devices
- Computers
- Custom CPE devices

Table A-22 lists the com.cisco.provisioning.cpe.api package interfaces.

***Table A-22 com.cisco.provisioning.cpe.api Package Interfaces***

| Interface | Description |
|---|---|
| Computer | Provides methods that manipulate provisioned computers. |
| Configuration | Contains configuration operations common to all sectors of BPR. |
| CustomCPE | Provides commands that manipulate CPE for proprietary devices. |
| DeviceSearch | Provides the methods that search for multiple CPE devices and retrieve a list of matching device identifiers. |
| DOCSIS | Provides methods that manipulate provisioned DOCSIS modems. |

*Table A-22 com.cisco.provisioning.cpe.api Package Interfaces (continued)*

| Interface | Description |
|---|---|
| DSTB | Provides methods that manipulate provisioned DSTB devices. |
| Provisioning | Contains methods that manipulate provisioned CPE devices in the BPR system. |
| XGCP | Contains methods that configure batch operations associated with xGCP devices. |

The following sections discuss the com.cisco.provisioning.cpe.api interfaces.

# Computer Interface

The computer interface contains methods that manipulate provisioned computers. Table A-23 lists the Computer methods.

*Table A-23 Computer Methods*

| Method | Description |
|---|---|
| addComputer() | Adds a computer. |
| addComputerByIPAddress() | Adds a computer identified by its IP address. |
| changeComputerClassOfService() | Changes a computer's class of service. |
| changeComputerDefaults() | Changes a computer's defaults |
| changeComputerDHCPCriteria() | Changes a computer DHCP criteria definition. |
| changeComputerFQDN() | Changes the FQDN of a computer. |
| changeComputerMACAddress() | Changes a computer's MAC address. |
| changeComputerOwnerID() | Changes a computer's owner identifier (ownership). |
| changeComputerProperties() | Changes a computer's properties. |
| deleteComputer() | Deletes a computer. |
| getAllComputers() | Gets the list of all computers. |
| getComputerDefaults() | Gets the defaults for a computer. |
| getDetailsForComputer() | Gets the device details for a given computer. |
| getDetailsForComputerByIPAddress() | Gets the device details for a given computer, identified by IP address. |

# Configuration Interface

Use the Configuration interface to get and set default configuration parameters for CPE devices. In the BPR, API methods are defined with arguments and properties. Arguments indicate the most commonly used fields that an API call must specify. In many cases, you can set arguments to null. You must specify properties in the form of a map containing key/value pairs.

The Configuration interface provides commonly used configuration methods. Table A-24 lists the Configuration methods.

*Table A-24 Configuration Methods*

| Method | Description |
|---|---|
| addCustomPropertyDefinition() | Adds a custom property definition. |
| addExternalFile() | Adds an external file. |
| addLicenseKey() | Adds a license key. |
| addUser() | Adds a user. |
| changeDPEDefaults() | Changes DPE default properties. |
| changeExtensionPointSettings() | Changes Network Registrar extension point settings. |
| changePublishingPluginSettings() | Changes the publishing plug-in default settings. |
| changeRDUDefaults() | Changes RDU default properties. |
| changeSystemDefaults() | Changes system default properties. |
| changeUser() | Changes user properties. |
| deleteCNR() | Deletes a Network Registrar server. |
| deleteDPE() | Deletes a DPE server. |
| deleteExternalFile() | Deletes an external file. |
| deleteUser() | Deletes a user. |
| disablePublishingPlugin() | Disables a publishing plug-in. |
| enablePublishingPlugin() | Enables a publishing plug-in. |
| getAllCNRs() | Retrieves a list of all currently registered Network Registrar servers. |
| getAllSystemPropertyDefinitions() | Retrieves a map of all currently defined custom property definitions. |
| getAllCustomPropertyDefinitions() | Gets a map of all custom property definitions currently defined in BPR. |
| getAllDPEs() | Retrieves a list of all currently registered DPE servers. |
| getAllProvGroups() | Retrieves a list of all provisioning groups. |
| getAllRDUs() | Retrieves a list of all currently registered RDUs. |
| getAllSystemPropertyDefinitions() | Gets a Map of all BPR pre-defined property definitions. |
| getAllUsers() | Retrieves a list of all users. |
| getCNRDefaults() | Retrieves a map of Network Registrar default properties. |
| getCNRDetails() | Retrieves the details for a Network Registrar server. |
| getCNRsByProvGroup() | Retrieves a list of all Network Registrar servers in a provisioning group. |
| getDPEDefaults() | Retrieves a map of DPE default properties. |
| getDPEDetails() | Retrieve the details for a DPE server. |
| getDPEsByProvGroup() | Retrieves a list of the DPE servers in a provisioning group. |
| getExtensionPointSettings() | Retrieves a map of extension point settings. |
| getExternalFile() | Retrieves an external file. |
| getLicenseKeyData() | Retrieves a list of maps of license key data. |

*Table A-24 Configuration Methods (continued)*

| Method | Description |
|---|---|
| getMatchingExternalFilenames() | Retrieves a list of known matching external filenames. |
| getPublishingPlugins() | Retrieves a list of publishing plug-ins. |
| getPublishingPluginSettings() | Retrieves the default settings for a specified plug-in. |
| getRDUDefaults() | Retrieves a map of RDU default properties. |
| getRDUDetails() | Retrieves the details for an RDU server. |
| getSystemDefaults() | Retrieves the default system properties. |
| getUserDetails() | Retrieves the details for a given user. |
| removeCustomPropertyDefinition() | Removes a custom property definition. |
| replaceExternalFile() | Replaces an external file. |

# CustomCPE Interface

Use the CustomCPE interface to manage proprietary technologies and equipment. Table A-25 lists the CustomCPE methods.

*Table A-25 CustomCPE Methods*

| Method | Description |
|---|---|
| addCustomCPE() | Adds a custom CPE device using a MAC address. |
| addCustomCPEByIPAddress() | Adds a custom CPE device using an IP address. |
| addCustomCPEType() | Adds a custom CPE device type. |
| changeCustomCPEClassOfService() | Changes the class of service for a custom CPE device. |
| changeCustomCPECPEDHCPCriteria() | Changes the CPE DHCP criteria of a custom CPE device. |
| changeCustomCPEDefaults() | Changes the defaults for custom CPE devices. |
| changeCustomCPEDHCPCriteria() | Changes the DHCP criteria for a custom CPE device. |
| changeCustomCPEFQDN() | Changes the FQDN for a custom CPE device. |
| changeCustomCPEMACAddress() | Changes the MAC address for a custom CPE device. |
| changeCustomCPEOwnerID() | Changes the owned identifier for a custom CPE device. |
| changeCustomCPEProperties() | Changes the properties for a custom CPE device. |
| changeCustomCPETypeProperties() | Changes the property type for a custom CPE device. |
| deleteCustomCPE() | Deletes a custom CPE device. |
| deleteCustomCPEType() | Deletes a custom CPE device type. |
| getAllCustomCPEs() | Retrieves a list of all custom CPE devices. |
| getAllCustomCPETypes() | Gets the list of all custom CPE types. |
| getCustomCPEDefaults() | Retrieves the defaults for custom CPE devices. |
| getCustomCPETypeDetails() | Retrieves the details for a custom CPE type. |

*Table A-25 CustomCPE Methods (continued)*

| Method | Description |
|---|---|
| getDetailsForCustomCPE() | Retrieves the details for a custom CPE device by identifier. |
| getDetailsForCustomCPEByIPAddress() | Retrieves the details for a custom CPE device by IP address. |

# DeviceSearch Interface

Use the Device Search interface to search for multiple CPE devices and to retrieve a list of matching device identifiers. Table A-26 lists the DeviceSearch methods.

*Table A-26 DeviceSearch Methods*

| Method | Description |
|---|---|
| getIPDevicesByClassOfService() | Looks up the IP devices with the designated class of service name. |
| getIPDevicesByCPEDHCPCriteria() | Looks up the IP devices with the designated CPE DHCP criteria name. |
| getIPDevicesByDHCPCriteria() | Looks up the IP devices with the designated DHCP criteria name. |
| getIPDevicesByFQDN() | Searches for IP devices by fully qualified domain name (FQDN). |
| getIPDevicesByIPAddress() | Looks up the IP devices whose IP addresses match the supplied ipAddress. |
| getIPDevicesByMACAddress() | Searches for IP devices by MAC address. |

# DOCSIS Interface

Use the DOCSIS interface to manage DOCSIS devices. Table A-27 lists the DOCSIS methods.

*Table A-27 DOCSIS Methods*

| Method | Description |
|---|---|
| addDOCSISModem() | Adds a DOCSIS modem. |
| addDOCSISModemByIPAddress() | Adds a DOCSIS modem identified by its IP address. |
| changeDOCSISDefaults() | Changes DOCSIS default properties. |
| changeDOCSISModemClassOfService() | Changes a DOCSIS modem's class of service. |
| changeDOCSISModemCPEDHCPCriteria() | Changes a DOCSIS modem’s CPE DHCP criteria. |
| changeDOCSISModemDHCPCriteria() | Changes a DOCSIS modem’s DHCP criteria. |
| changeDOCSISModemFQDN() | Changes a DOCSIS modem's FQDN. |
| changeDOCSISModemMACAddress() | Changes a DOCSIS modem's MAC address. |
| changeDOCSISModemOwnerID() | Changes a DOCSIS modem's owner identifier. |

*Table A-27 DOCSIS Methods (continued)*

| Method | Description |
|---|---|
| changeDOCSISModemProperties() | Changes a DOCSIS modem's properties. |
| deleteDOCSISModem() | Deletes a DOCSIS modem. |
| getAllDOCSISModems() | Gets the list of DOCSIS modems |
| getDetailsForDOCSISModem() | Gets the device details for a given DOCSIS modem. |
| getDetailsForDOCSISModemByIPAddress() | Gets the device details for a given DOCSIS modem, identified by IP address. |
| getDOCSISDefaults() | Gets the DOCSIS defaults. |

# DSTB Interface

Use the DSTB interface to configure digital set-top box components for BPR and to retrieve information about the components from BPR. Table A-28 lists the DSTB methods.

*Table A-28 DSTB Methods*

| Method | Description |
|---|---|
| addDSTB() | Adds a DSTB. |
| addDSTBByIPAddress() | Adds a DSTB identified by its IP address. |
| changeDSTBClassOfService() | Changes a DSTB's class of service. |
| changeDSTBDefaults() | Changes a DSTB's default properties. |
| changeDSTBDHCPCriteria() | Changes a DSTB's DHCP criteria. |
| changeDSTBFQDN() | Changes the FQDN of a DSTB. |
| changeDSTBMACAddress() | Changes a DSTB's MAC address. |
| changeDSTBOwnerID() | Changes the owner identifier of a DSTB. |
| changeDSTBProperties() | Changes the properties of a DSTB. |
| changeDSTBRelatedDOCSISModemID() | Changes the modem identifier of a DSTB's related DOCSIS modem. |
| deleteDSTB() | Deletes a DSTB and optionally deletes related DOCSIS modems related to the DSTB. |
| getAllDSTBs() | Gets the list of all DSTBs. |
| getDetailsForDSTB() | Gets the device details for a given DSTB. |
| getDetailsForDSTBByIPAddress() | Gets the device details for a given DSTB, identified by IP address. |
| getDSTBDefaults() | Retrieve the DSTB defaults in a map. |

# Provisioning Interface

Use the Provisioning interface to manipulate provisioned customer premises equipment (CPE) in the BPR system.

**Note** The Provisioning interface in CSRC DPR contained calls now deprecated in BPR. See Appendix B, "Deprecated Methods" for the list of deprecated calls.

Table A-29 lists the Provisioning methods.

***Table A-29 Provisioning Methods***

| Method | Description |
|---|---|
| addClassOfService() | Adds a class of service. |
| addDHCPCriteria() | Adds client class and scope selection tag definitions. |
| changeClassOfServiceProperties() | Changes a class of service's properties. |
| changeDHCPCriteriaClientClass() | Changes a DHCP criteria client class definition. |
| changeDHCPCriteriaExcludeSelectionTags() | Changes a DHCP criteria exclude selection tag definition. |
| changeDHCPCriteriaIncludeSelectionTags() | Changes a DHCP criteria include selection tag definition. |
| changeDHCPCriteriaProperties() | Changes a DHCP criteria property definition. |
| deleteClassOfService() | Deletes a class of service. |
| deleteDHCPCriteria() | Deletes a DHCP criteria definition. |
| deleteIPDevice() | Deletes an IP device (regardless of type). |
| generateConfiguration() | Generates a configuration for the device specified. |
| getAllClassesOfService() | Gets a list of the names of all classes of service, for a given type. |
| getAllDHCPCriterias() | Gets a list of all DHCP criteria definitions. |
| getClassOfServiceProperties() | Gets the properties for the given class of service. |
| getDetailsForIPDevice() | Gets the device details for a given IP device instance. |
| getDetailsForIPDeviceByIPAddress() | Gets the IP device details for a given IP address. |
| getDHCPCriteriaDetails() | Gets the details for a DHCP criteria definition. |
| getIPDeviceDetailsList() | Gets a list of IP device details for the list of deviceID strings supplied. |
| getIPDeviceDetailsListByIPAddress() | Gets a list of IP device details for the list of ipAddress strings supplied. |
| getIPDeviceForIPAddress() | Looks up the IP device for the specified IP address. |
| getIPDevicesBehindModem() | Lists all of the devices currently located behind the specified modem. |
| getIPDevicesByOwnerID() | Gets the list of IP devices for a specified owner ID. |
| resetIPDevice() | Resets (reboots) an IP device. |

## XGCP Interface

Use the XGCP interface to manage batch operations associated with xGCP devices. This interface contains the methods used to configure xGCP information for BPR and to retrieve xGCP information from BPR. Table A-30 lists the XGCP methods.

***Table A-30 XGCP Methods***

| Method | Description |
|---|---|
| addXGCPService() | Assigns an xGCP service to a specified port on a modem. |
| changeXGCPServiceCmsFQDN() | Changes the FQDN of the call management system (CMS) associated with an xGCP service. |
| changeXGCPServiceCmsPortNumber() | Changes the CMS port number associated with an xGCP service. |
| changeXGCPServiceCmsQualifier() | Changes the CMS Qualifier associated with an xGCP service. |
| changeXGCPServiceProperties() | Changes properties associated with an xGCP service. |
| changeXGCPServiceTelephoneNumber() | Changes the telephone number associated with an xGCP service. |
| deleteXGCPService() | Deletes the service from the specified port. |
| getXGCPServiceDetails() | Gets the xGCP port information. |

# com.cisco.provisioning.cpe.constants Package

The com.cisco.provisioning.cpe.constants package contains the interfaces and constants that interpret status information and retrieve values associated with the device types that BPR provisions. Table A-31 lists the interfaces in the com.cisco.provisioning.cpe.constants package.

***Table A-31 com.cisco.provisioning.cpe.constants Package***

| Interface | Description |
|---|---|
| BatchStatusCodes | Specifies constants that are used to interpret returned status information for a batch operation of commands. |
| ClassOfServiceKeys | Specifies constant strings that are used as keys in class of service objects. |
| ClassOfServiceType | Specifies the class of service types supported by BPR. |
| CNRDetailsKeys | Specifies constant strings that are used as keys in map objects returned by queries to CNR. |
| CNRExtensionSettingKeys | Specifies constant strings that are used as keys in map objects by the changeExtensionPointSettings() method and returned by the getExtensionPointSettings() method. |
| CNRServersKeys | Specifies constant strings that are used as system default keys for a BPR configuration of Network Registrar. |

*Table A-31 com.cisco.provisioning.cpe.constants Package (continued)*

| Interface | Description |
|---|---|
| CommandStatusCodes | Specifies constants that are used to interpret the returned command status. |
| DeviceDetailsKeys | Specifies character strings that are used as keys to retrieve values associated with IP devices, for example an IP address or a MAC address. |
| DeviceDisruptionStatus | Provides the different status codes returned by a device disruption. |
| DeviceTypeValues | Specifies the allowed values for a device type. |
| DHCPCriteriaKeys | Specifies constant strings that are used as keys in DHCPCriteria objects. |
| DHCPOptionKeys | Specifies constants that are used as keys in map objects to set and get properties that apply to BPR DHCP configuration and BPR client policy support. |
| DocisisDefaultKeys | Specifies the constant strings that are used as keys in map objects to set and get properties that apply to DOCSIS and RDU defaults. |
| DPEDetailsKeys | Specifies constants that are used as keys in map objects returned by query methods on DPE. |
| IPDeviceKeys | Specifies constant strings that are used as keys in properties map objects for IP device provisioning interfaces. |
| LicenseCodes | Specifies constant strings that are used as algorithms needed by the write method for license key data. |
| LicenseDataKeys | Specifies constant strings that are used as keys in map objects returned by the query method for license key data. |
| ModemKeys | Specifies constants that are used as keys in map objects returned by query methods on DOCSIS interfaces. |
| PublishDetailsKeys | Specifies constant strings, which are used as keys in Map objects returned by query methods on Publish objects. |
| RDUDetailsKeys | Specifies constants that are used as keys in map objects returned by query methods on the RDU. |
| SearchType | Enumerates the search types supported by BPR. |
| ServerDefaultKeys | Specifies the constants that servers use. |
| SNMPPropertyKeys | Specifies character strings that are used as keys to store and retrieve values associated with devices that respond to simple network management protocol (SNMP) requests, for example their read and write community strings. |
| TechnologyDefaultsKeys | Specifies constants that are used as keys in map objects to get and set properties that apply to technology defaults. |
| UserDetailsKeys | Specifies constants that are used as keys in map objects returned by query methods on user objects. |
| VOIPServiceDetailsKeys | Specifies character strings that are used as keys to retrieve values associated with xGCP devices, for example, an FQDN, a port number, or a telephone number. |

*Table A-31 com.cisco.provisioning.cpe.constants Package (continued)*

| Interface | Description |
|---|---|
| CommandStatusCodes | Specifies constants that are used to interpret the returned command status. |
| DeviceDetailsKeys | Specifies character strings that are used as keys to retrieve values associated with IP devices, for example an IP address or a MAC address. |
| DeviceDisruptionStatus | Provides the different status codes returned by a device disruption. |
| DeviceTypeValues | Specifies the allowed values for a device type. |
| DHCPCriteriaKeys | Specifies constant strings that are used as keys in DHCPCriteria objects. |
| DHCPOptionKeys | Specifies constants that are used as keys in map objects to set and get properties that apply to BPR DHCP configuration and BPR client policy support. |
| DocisisDefaultKeys | Specifies the constant strings that are used as keys in map objects to set and get properties that apply to DOCSIS and RDU defaults. |
| DPEDetailsKeys | Specifies constants that are used as keys in map objects returned by query methods on DPE. |
| IPDeviceKeys | Specifies constant strings that are used as keys in properties map objects for IP device provisioning interfaces. |
| LicenseCodes | Specifies constant strings that are used as algorithms needed by the write method for license key data. |
| LicenseDataKeys | Specifies constant strings that are used as keys in map objects returned by the query method for license key data. |
| ModemKeys | Specifies constants that are used as keys in map objects returned by query methods on DOCSIS interfaces. |
| PublishDetailsKeys | Specifies constant strings, which are used as keys in Map objects returned by query methods on Publish objects. |
| RDUDetailsKeys | Specifies constants that are used as keys in map objects returned by query methods on the RDU. |
| SearchType | Enumerates the search types supported by BPR. |
| ServerDefaultKeys | Specifies the constants that servers use. |
| SNMPPropertyKeys | Specifies character strings that are used as keys to store and retrieve values associated with devices that respond to simple network management protocol (SNMP) requests, for example their read and write community strings. |
| TechnologyDefaultsKeys | Specifies constants that are used as keys in map objects to get and set properties that apply to technology defaults. |
| UserDetailsKeys | Specifies constants that are used as keys in map objects returned by query methods on user objects. |
| VOIPServiceDetailsKeys | Specifies character strings that are used as keys to retrieve values associated with xGCP devices, for example, an FQDN, a port number, or a telephone number. |

*Table A-31 com.cisco.provisioning.cpe.constants Package (continued)*

| Interface | Description |
|---|---|
| XGCPCommandStatusCodes | Specifies constants that are used to interpret returned xGCP command and query status information. |
| XGCPProvisioningKeys | Specifies the constants that are used to interpret and set the values associated with xGCP devices, for example, the SGCP version string. |

The following sections provide a summary of the interfaces in this package.

# BatchStatusCodes Interface

Use the Batch Status Codes interface to obtain status information about a batch operation. Table A-32 lists the BatchStatus.getStatusCode values.

*Table A-32 BatchStatusCodes Values*

| Value | Description |
|---|---|
| BATCH_BEING_PROCESSED | Return value from BatchStatus.getStatusCode(). |
| BATCH_CANNOT_USE_ANONYMOUS_ CONNECTION | Return value from BatchStatus.getStatusCode(). |
| BATCH_COMPLETED | Return value from BatchStatus.getStatusCode(). |
| BATCH_DUPLICATE_DROPPED | Return value from BatchStatus.getStatusCode(). |
| BATCH_DUPLICATE_ID | Return value from BatchStatus.getStatusCode(). |
| BATCH_ERROR | Return value from BatchStatus.getStatusCode(). |
| BATCH_FAILED_ACTIVATE | Return value from BatchStatus.getStatusCode(). |
| BATCH_FAILED_COMMIT | Return value from BatchStatus.getStatusCode(). |
| BATCH_FAILED_CONFIGURATION_GENERATION | Return value from BatchStatus.getStatusCode(). |
| BATCH_FAILED_DISRUPTION | Return value from BatchStatus.getStatusCode(). |
| BATCH_FAILED_NO_DISRUPT_ACTIVATION | Return value from BatchStatus.getStatusCode(). |
| BATCH_FAILED_PREPARE | Return value from BatchStatus.getStatusCode(). |
| BATCH_FAILED_PUBLISH | Return value from BatchStatus.getStatusCode(). |

*Table A-32 BatchStatusCodes Values (continued)*

| Value | Description |
|---|---|
| BATCH_FAILED_QUERY | Return value from BatchStatus.getStatusCode(). |
| BATCH_FAILED_VALIDATION | Return value from BatchStatus.getStatusCode(). |
| BATCH_FAILED_WRITE | Return value from BatchStatus.getStatusCode(). |
| BATCH_ILLEGAL_MULTIPLE_ACTIVATIONS | Return value from BatchStatus.getStatusCode(). |
| BATCH_INVALID_LICENSES | Return value from BatchStatus.getStatusCode(). |
| BATCH_PASSED_VALIDATION | Return value from BatchStatus.getStatusCode(). |
| BATCH_POST_TIMEOUT | Return value from BatchStatus.getStatusCode(). |
| BATCH_RDU_BUSY | Return value from BatchStatus.getStatusCode(). |
| BATCH_SYSTEM_ERROR | Return value from BatchStatus.getStatusCode(). |
| BATCH_WARNING | Return value from BatchStatus.getStatusCode(). |

# ClassOfServiceKeys Interface

The ClassOfServiceKeys interface specifies constant strings that are used as keys in ClassOfService objects. Table A-33 lists the class of service keys.

*Table A-33 ClassOfServiceKeys*

| Key | Description |
|---|---|
| COS_DOCSIS_FILE | DOCSIS class of service file. |

# ClassOfServiceType Interface

The ClassOfServiceType interface provides constants that reference the classes of service that BPR can store. Table A-34 lists the class of service types.

*Table A-34 Class of Service Types*

| Type | Description |
|---|---|
| COMPUTER | References a Computer class of service. |
| CUSTOMCPE | References a CustomCPE class of service. |

*Table A-34 Class of Service Types (continued)*

| Type | Description |
|---|---|
| DOCSIS | References a DOCSIS class of service. |
| DSTB | References a DSTB class of service. |

# CNRDetailsKeys Interface

The CNRDetails interface specifies constants that are used as keys in map objects returned by query methods on Network Registrar. Table A-35 lists the CNR details keys.

*Table A-35 CNRDetailsKeys*

| Key | Description |
|---|---|
| CNR_BATCHES | Version number for the queried Network Registrar extension point/server. |
| CNR_HEALTH | Health of the queried Network Registrar extension point/server. |
| CNR_HOST_ID | Host identifier for the queried Network Registrar extension point/server. |
| CNR_HOST_PORT | Port number for the queried Network Registrar extension point/server. |
| CNR_PROPERTIES | Properties for the queried Network Registrar extension point/server. |
| CNR_PROV_GROUP_ID | Provisioning group identifier for the queried Network Registrar extension point/server. |
| CNR_STATE | State of the queried Network Registrar extension point/server. |
| CNR_UPTIME | Uptime for the queried Network Registrar extension point/server. |
| CNR_VERSION | Software version of the queried Network Registrar extension point/server. |
| DPES_STATUS_KEY | Identifies DPEs and their status. |

# CNRExtensionSettingKeys Interface

The CNRExtensionSettingKeys interface specifies constant strings that can be used as keys in the map objects used by change Extension Point Settings() and returned by getExtensionPointSettings(). Table A-36 lists the CNR extension setting keys.

*Table A-36 CNRExtensionSettingKeys*

| Key | Description |
|---|---|
| CNR_ATTRIBUTES_TO_READ_FROM_ENVIRONMENT_DICTIONARY | List of all the attributes to be pulled from Network Registrar environment dictionary. |
| CNR_ATTRIBUTES_TO_READ_FROM_REQUEST_DICTIONARY | List of all the attributes to be pulled from Network Registrar request dictionary. |

# CNRServersKeys Interface

Use the CNRServersKeys interface to specify the constants that are used as system default keys during BPR configuration. Table A-37 lists the CNRServersKeys.

***Table A-37 CNRServersKeys***

| Key | Description |
|---|---|
| CNR_CLIENT_PORT | The listening port of the API when communicating with the Network Registrar servers. |
| CNR_NUMBER_RETRIES | The number of retries when communicating with the Network Registrar servers. |
| CNR_SERVERS_LIST | Returns a list of Network Registrar servers to query. |
| CNR_TIMEOUT | The time in milliseconds between retries when communicating with the Network Registrar servers. |
| USE_CLIENT_ID | If true, uses the client ID instead of the MAC address. |

# CommandStatusCodes Interface

The Command Status Codes interface specifies a number of constants that are used to interpret the return status of HSD commands. The CommandStatus.getStatusCode() function returns values to indicate that HSD command processing succeeded or that a command processing error occurred. Table A-38 lists CommandStatusCodes values.

***Table A-38 CommandStatusCodes Values***

| Value | Description |
|---|---|
| CMD_DB_LOCK | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_ADD_USER | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_CLASS_OF_SERVICE_EXISTS | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_CLASS_OF_SERVICE_ UNKNOWN | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_CNR_UNKNOWN | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_COMPUTER_CLASS_OF_ SERVICE_UNKNOWN | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_COMPUTERS_BEHIND_ MODEM | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_SYSTEM_PROPERTY_ REDEFINITION | Return value from CommandStatus.getStatusCode(). |

*Table A-38 CommandStatusCodes Values (continued)*

| Value | Description |
|---|---|
| CMD_ERROR_CUSTOM_CPE_CLASS_OF_ SERVICE_UNKNOWN | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_CUSTOM_CPE_TYPE_EXISTS | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_CUSTOM_CPE_TYPE_ UNKNOWN | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_CUSTOM_CPES_BEHIND_ MODEM | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_CUSTOM_PROPERTY_ILLEGAL _NAME | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_CUSTOM_PROPERTY_ REDEFINITION | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_CUSTOM_PROPERTY_ UNKNOWN | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_DATA_UNCHANGED | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_DB_UNREACHABLE | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_DEVICE_FQDN_EXISTS | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_DEVICE_FQDN_UNKNOWN | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_DEVICE_HOSTID_EXISTS | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_DEVICE_HOSTID_UNKNOWN | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_DEVICE_ID_UNKNOWN | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_DEVICE_MAC_EXISTS | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_DEVICE_SEARCH_NO_MATCH | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_DEVICE_UNKNOWN | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_DEVICEID_INVALID | Return value from CommandStatus.getStatusCode(), representing a command with an invalid deviceID |
| CMD_ERROR_DHCP_CRITERIA_EXISTS | Return value from CommandStatus.getStatusCode(). Indicates that a DHCP Criteria with the specified name already exists in the database. |

*Table A-38 CommandStatusCodes Values (continued)*

| Value | Description |
|---|---|
| CMD_ERROR_DHCP_CRITERIA_UNKNOWN | Return value from CommandStatus.getStatusCode(). Indicates that a DHCP Criteria with the specified name is not present in the database. |
| CMD_ERROR_DISABLE_PLUGIN | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_DOCSIS_CLASS_OF_SERVICE_UNKNOWN | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_DPE_UNKNOWN | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_DSTB_CLASS_OF_SERVICE_UNKNOWN | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_ENABLE | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_ENABLE_PLUGIN | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_EVAL_LICENSE_EXTENDER_IS_LESS_THAN_EXISTING_EVAL_LICENSE_EXTENDER | Return value from CommandStatus.getStatusCode(). Indicates that the evaluation license extender is less than the previous evaluation extender. |
| CMD_ERROR_EVAL_LICENSE_REPLACING_EXISTING_PERM_LICENSE | Return value from CommandStatus.getStatusCode(). Indicates that an evaluation license key cannot replace an existing permanent license. |
| CMD_ERROR_EVALUATION_HAS_EXPIRED | Return value from CommandStatus.getStatusCode(). Indicates that the encrypted key's evaluation duration has expired. |
| CMD_ERROR_EXTERNAL_FILE_UNKNOWN | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_FILENAME_INVALID_REGEX | Return value from CommandStatus.getStatusCode(), representing a command with an invalid filename regex. |
| CMD_ERROR_FQDN_INVALID | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_GET_ALL_USERS | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_HOSTID_INVALID | Return value from CommandStatus.getStatusCode(), representing a command with an invalid hostID. |
| CMD_ERROR_ILLEGAL_PARAM_VALUE | Return value from CommandStatus.getStatusCode(). |

*Table A-38 CommandStatusCodes Values (continued)*

| Value | Description |
|---|---|
| CMD_ERROR_INTERNAL | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_INVALID_DEVICE_ TECHNOLOGY | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_INVALID_LICENSE_KEY | Return value from CommandStatus.getStatusCode()., Represents failure of the addLicense() command in a batch. |
| CMD_ERROR_IPADDR_UNKNOWN | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_IPADDRESS_INVALID | Return value from CommandStatus.getStatusCode(), representing a command with an invalid IPADDRESS parameter. |
| CMD_ERROR_LEASE_UNKNOWN | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_LICENSE_IS_INVALID | Return value from CommandStatus.getStatusCode(). Indicates that the decoded data (from decoding the encrypted key) is invalid. |
| CMD_ERROR_LICENSE_IS_NOT_PERM_OR_ EVAL | Return value from CommandStatus.getStatusCode(). Indicates that the encrypted key does not have permanent or evaluation license. |
| CMD_ERROR_LICENSE_IS_NULL | Return value from CommandStatus.getStatusCode(). Indicates that the encrypted key is null or empty. |
| CMD_ERROR_LIST_PLUGIN | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_MAC_AND_FQDN_BOTH_NULL | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_MACADDRESS_INVALID | Return value from CommandStatus.getStatusCode(), representing a command with an invalid MACADDRESS parameter. |
| CMD_ERROR_MTAS_BEHIND_MODEM | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_OWNERID_UNKNOWN | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_PERM_LICENSE_KEY_ ALREADY_EXISTS | Return value from CommandStatus.getStatusCode(). Indicates that the permanent license key already exists. |
| CMD_ERROR_PLUGIN_UNKNOWN | Return value from CommandStatus.getStatusCode(). |

*Table A-38 CommandStatusCodes Values (continued)*

| Value | Description |
|---|---|
| CMD_ERROR_PREPARE | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_PROPERTY_UNKNOWN | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_PROVISIONING_GROUP_ UNKNOWN | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_QUERY | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_RDU_UNKNOWN | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_ROLLBACK_ENABLE | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_ROLLBACK_PREPARE | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_SEARCH_NO_MATCH | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_SYSTEM_PROPERTY_REDEFIN ITION | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_TOO_MANY_IN_LIST | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_UNKNOWN_TECHNOLOGY_ NAME | Return value from CommandStatus.getStatusCode(). Indicates that the encrypted key does not have a recognizable technology name. |
| CMD_ERROR_USER_ADMIN | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_USER_ALREADY_EXISTS | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_USER_IS_NOT_ADMIN | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_USER_MODIFICATION_ERROR | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_USER_UNKNOWN | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_VALIDATE | Return value from CommandStatus.getStatusCode(). |
| CMD_ERROR_WRITE | Return value from CommandStatus.getStatusCode(). |
| CMD_OK | Return value from CommandStatus.getStatusCode(). |
| CMD_RDU_BUSY | Return value from CommandStatus.getStatusCode(). |

*Table A-38 CommandStatusCodes Values (continued)*

| Value | Description |
|---|---|
| CMD_ROLLBACK_PREV_CMD_FAILED | Return value from CommandStatus.getStatusCode(). |
| CMD_ROLLBACK_SUBS_CMD_FAILED | Return value from CommandStatus.getStatusCode(). |

# DeviceDetailsKeys Interface

The Device Details Keys interface specifies the character strings that are used as keys to retrieve values associated with IP devices. This interface defines keys that provide detailed information about IP devices that you are querying. This interface also defines keys that provide information related to a specific technology. Table A-39 lists the device details keys.

*Table A-39 DeviceDetailsKeys*

| Key | Description |
|---|---|
| CLIENT_CLASS | DHCP criteria for the queried device. |
| CLIENT_ID | Retrieves the client identifier of a device. |
| CLIENT_REQUESTED_HOST_NAME | Retrieves the host name that a client requested. |
| COMPUTER_COS | Computer class of service for the queried device. |
| CPE_DHCP_CRITERIA | DHCP criteria for the queried device. |
| CPE_SELECTION_TAGS | CPE DHCP criteria for the queried device. |
| CUSTOM_COS | Gets custom class of service data about a device. |
| CUSTOM_CPE_TYPE | Custom CPE type for the queried device. |
| DEVICE_TYPE | Retrieves the device type. |
| DHCP_CRITERIA | DHCP criteria for the queried device. |
| DHCP_ENVIRONMENT | DHCP configuration data - Environment dictionary. |
| DHCP_INFORM | DHCP configuration data - Inform dictionary. |
| DHCP_REQUEST | DHCP configuration data - Request dictionary. |
| DHCP_RESPONSE | DHCP configuration data - Response dictionary. |
| DOCSIS_COS | Gets the DOCSIS class of service of a device. |
| DOCSIS_VERSION | Obtains the DOCSIS version of a device. |
| DSTB_COS | DSTB class of service for the queried device. |
| DSTB_RELATED_DOCSIS_ MODEM_ID | Related DOCSIS modem ID for the queried DSTB device. |
| FQDN | FQDN (fully qualified domain name) for the queried device. |
| IP_ADDRESS | IP address for the queried device. |
| IS_BEHIND_DEVICE | Device this device is behind as determined by device detection. |
| IS_PROVISIONED | Provisioned state for this device. |

*Table A-39 DeviceDetailsKeys (continued)*

| Key | Description |
|---|---|
| IS_REGISTERED | Registered state for this device. |
| LEASE_PROPERTIES | Lease properties |
| MAC_ADDRESS | MAC address for the queried device. |
| OWNER_ID | Owner ID for the queried device. |
| PROPERTIES | Properties for the queried device. |
| PROV_GROUP | Provisioning group for the queried device. |
| RELAY_AGENT_CIRCUIT_ID | Relay agent circuit ID for the queried device. |
| RELAY_AGENT_REMOTE_ID | Relay agent remote ID for the queried device. |
| XGCP_MTA_PORTS_IN_SERVICE_ LIST | List of ports in service for the queried XGCP device. |
| XGCP_MTA_RELATED_DOCSIS_ MODEM_ID | Related DOCSIS modem ID for the queried XGCP MTA device. |

# DeviceTypeValues Interface

The Device TypeValues interface specifies the allowed values for a device type. These are returned as the values for the DeviceDetailsKeys.DEVICE_TYPE key. Table A-40 identifies these values.

*Table A-40 DeviceTypeValues*

| Value | Description |
|---|---|
| COMPUTER | Device type value for a computer. |
| CUSTOM_CPE | Device type value for a custom CPE device. |
| DOCSIS_MODEM | Device type value for a DOCSIS modem. |
| DSTB | Device type value for a DSTB. |
| XGCP_MODEM_MTA | Device type value for a XGCP modem MTA. |

# DHCPCriteriaKeys Interface

The DHCPCriteriaKeys interface contains constant strings that are used as keys in DHCPCriteria objects. Table A-41 lists the DHCPCriteriaKeys.

*Table A-41 DHCPCriteriaKeys*

| Key | Description |
|---|---|
| DHCP_CRITERIA_CLIENT_CLASS | DHCP criteria client class. |
| DHCP_CRITERIA_EXCLUDE_SELECTION_TAGS | DHCP criteria exclude selection tags. |
| DHCP_CRITERIA_INCLUDE_SELECTION_TAGS | DHCP criteria include selection tags. |
| DHCP_CRITERIA_NAME | DHCP criteria name. |
| DHCP_CRITERIA_PROPERTIES | DHCP criteria properties. |

# DHCPOptionKeys Interface

The DHCPOptionKeys interface specifies constant strings that are used as keys in map objects to set and get properties that apply to BPR DHCP configuration and BPR client-policy support. Table A-42 lists the DHCP option keys.

***Table A-42 DHCPOptionKeys***

| Key | Description |
|---|---|
| DHCP_CLIENT_POLICY_ALL_SUBNETS_LOCAL | The all-subnets-local client-policy response option. |
| DHCP_CLIENT_POLICY_ARP_CACHE_TIMEOUT | The arp-cache-timeout client-policy response option. |
| DHCP_CLIENT_POLICY_BOOT_FILE | The boot-file client-policy response option. |
| DHCP_CLIENT_POLICY_BOOT_SIZE | The boot-size client-policy response option. |
| DHCP_CLIENT_POLICY_BROADCAST_ADDRESS | The broadcast-address client-policy response option. |
| DHCP_CLIENT_POLICY_CHADDR | The chaddr client-policy response option. |
| DHCP_CLIENT_POLICY_CIADDR | The ciaddr client-policy response option. |
| DHCP_CLIENT_POLICY_CISCO_CLIENT_LAST_TRANSACTION_TIME | The cisco-client-last-transaction-time client-policy response option. |
| DHCP_CLIENT_POLICY_CISCO_CLIENT_REQUESTED_HOST_NAME | The cisco-client-requested-host-name client-policy response option. |
| DHCP_CLIENT_POLICY_CISCO_LEASED_IP | The cisco-leased-ip client-policy response option. |
| DHCP_CLIENT_POLICY_CLIENT_DOMAIN_NAME | The client-domain-name client-policy response option. |
| DHCP_CLIENT_POLICY_CLIENT_FQDN | The client-fqdn client-policy response option. |
| DHCP_CLIENT_POLICY_CLIENT_HOST_NAME | The client-host-name client-policy response option. |
| DHCP_CLIENT_POLICY_CLIENT_ID | The client-id client-policy response option. |
| DHCP_CLIENT_POLICY_CLIENT_MAC_ADDRESS | The client-mac-address client-policy response option. |
| DHCP_CLIENT_POLICY_CLIENT_OS_TYPE | The client-os-type client-policy response option. |
| DHCP_CLIENT_POLICY_CLIENT_REQUESTED_HOST_NAME | The client-requested-host-name client-policy response option. |
| DHCP_CLIENT_POLICY_COOKIE_SERVERS | The cookie-servers client-policy response option. |
| DHCP_CLIENT_POLICY_DEFAULT_IP_TTL | The default-ip-ttl client-policy response option. |
| DHCP_CLIENT_POLICY_DEFAULT_TCP_TTL | The default-tcp-ttl client-policy response option. |
| DHCP_CLIENT_POLICY_DHCP_CLASS_IDENTIFIER | The dhcp-class-identifier client-policy response option. |

*Table A-42 DHCPOptionKeys (continued)*

| Key | Description |
|---|---|
| DHCP_CLIENT_POLICY_DHCP_CLIENT_IDENTIFIER | The dhcp-client-identifier client-policy response option. |
| DHCP_CLIENT_POLICY_DHCP_LEASE_TIME | The dhcp-lease-time client-policy response option. |
| DHCP_CLIENT_POLICY_DHCP_MAX_MESSAGE_SIZE | The dhcp-max-message-size client-policy response option. |
| DHCP_CLIENT_POLICY_DHCP_MESSAGE | The dhcp-message client-policy response option. |
| DHCP_CLIENT_POLICY_DHCP_MESSAGE_TYPE | The dhcp-message-type client-policy response option. |
| DHCP_CLIENT_POLICY_DHCP_OPTION_OVERLOAD | The dhcp-option-overload client-policy response option. |
| DHCP_CLIENT_POLICY_DHCP_PARAMETER_REQUEST_LIST | The dhcp-parameter-request-list client-policy response option. |
| DHCP_CLIENT_POLICY_DHCP_PARAMETER_REQUEST_LIST_BLOB | The dhcp-parameter-request-list-blob client-policy response option. |
| DHCP_CLIENT_POLICY_DHCP_REBINDING_TIME | The dhcp-rebinding-time client-policy response option. |
| DHCP_CLIENT_POLICY_DHCP_RENEWAL_TIME | The dhcp-renewal-time client-policy response option. |
| DHCP_CLIENT_POLICY_DHCP_REQUESTED_ADDRESS | The dhcp-requested-address client-policy response option. |
| DHCP_CLIENT_POLICY_DHCP_SERVER_IDENTIFIER | The dhcp-server-identifier client-policy response option. |
| DHCP_CLIENT_POLICY_DHCP_USER_CLASS_ID | The dhcp-user-class-id client-policy response option. |
| DHCP_CLIENT_POLICY_DOMAIN_NAME | The domain-name client-policy response option. |
| DHCP_CLIENT_POLICY_DOMAIN_NAME_CHANGED | The domain-name-changed client-policy response option. |
| DHCP_CLIENT_POLICY_DOMAIN_NAME_SERVERS | The domain-name-servers client-policy response option. |
| DHCP_CLIENT_POLICY_DUMP_PACKET | The dump-packet client-policy response option. |
| DHCP_CLIENT_POLICY_ENV_ACCEPTABLE | The acceptable client-policy environment option. |
| DHCP_CLIENT_POLICY_ENV_ACTION | The action client-policy environment option. |
| DHCP_CLIENT_POLICY_ENV_AUTHENTICATE_UNTIL | The authenticate-until client-policy environment option. |
| DHCP_CLIENT_POLICY_ENV_CLIENT_CLASS_NAME | The client-class-name client-policy environment option. |
| DHCP_CLIENT_POLICY_ENV_CLIENT_SPECIFIER | The client-specifier client-policy environment option. |

*Table A-42 DHCPOptionKeys (continued)*

| Key | Description |
|---|---|
| DHCP_CLIENT_POLICY_ENV_CNR_ FORWARD_DHCP_REQUEST | The cnr-forward-dhcp-request client-policy environment option. |
| DHCP_CLIENT_POLICY_ENV_CNR_ REQUEST_FORWARD_ADDRESS_LIST | The cnr-request-forward-address-list client-policy environment option. |
| DHCP_CLIENT_POLICY_ENV_DEFAULT_ CLIENT_CLASS_NAME | The default-client-class-name client-policy environment option. |
| DHCP_CLIENT_POLICY_ENV_DOMAIN_ NAME | The domain-name client-policy environment option. |
| DHCP_CLIENT_POLICY_ENV_DROP | The drop client-policy environment option. |
| DHCP_CLIENT_POLICY_ENV_ EMBEDDED_POLICY | The embedded-policy client-policy environment option. |
| DHCP_CLIENT_POLICY_ENV_E XCLUSION_MASK | The exclusion-mask client-policy environment option. |
| DHCP_CLIENT_POLICY_ENV_HOST_ NAME | The host-name client-policy environment option. |
| DHCP_CLIENT_POLICY_ENV_I NCLUSION_MASK | The inclusion-mask client-policy environment option. |
| DHCP_CLIENT_POLICY_ENV_LOG_ DROP_MESSAGE | The log-drop-message client-policy environment option. |
| DHCP_CLIENT_POLICY_ENV_POLICY_ NAME | The policy-name client-policy environment option. |
| DHCP_CLIENT_POLICY_ENV_RELEASE_ BY_IP | The release-by-ip client-policy environment option. |
| DHCP_CLIENT_POLICY_ENV_ SELECTION_CRITERIA | The selection-criteria client-policy environment option. |
| DHCP_CLIENT_POLICY_ENV_ SELECTION_CRITERIA_EXCLUDED | The selection-criteria-excluded client-policy environment option. |
| DHCP_CLIENT_POLICY_ENV_SKIP_ CLIENT_LOOKUP | The skip-client-lookup client-policy environment option. |
| DHCP_CLIENT_POLICY_ENV_ UNAUTHENTICATED_CLIENT_ CLASS_NAME | The unauthenticated-client-class-name client-policy environment option. |
| DHCP_CLIENT_POLICY_ENVIRONMENT_ PREFIX | The client policy prefix for CNR environment dictionary. |
| DHCP_CLIENT_POLICY_EXTENSIONS_ PATH | The extensions-path client-policy response option. |
| DHCP_CLIENT_POLICY_FAILOVER_ ABSOLUTE_TIME | The failover-absolute-time client-policy response option. |
| DHCP_CLIENT_POLICY_FAILOVER_ CLIENT_FLAGS | The failover-client-flags client-policy response option. |
| DHCP_CLIENT_POLICY_FAILOVER_ CLIENT_OS_TYPE | The failover-client-os-type client-policy response option. |

*Table A-42 DHCPOptionKeys (continued)*

| Key | Description |
|---|---|
| DHCP_CLIENT_POLICY_FAILOVER_ LAST_TRANSACTION_TIME | The failover-last-transaction-time client-policy response option. |
| DHCP_CLIENT_POLICY_FAILOVER_ LEASE_EXPIRATION_TIME | The failover-lease-expiration-time client-policy response option. |
| DHCP_CLIENT_POLICY_FAILOVER_ LEASE_STATE | The failover-lease-state client-policy response option. |
| DHCP_CLIENT_POLICY_FAILOVER_ MAC_ADDR | The failover-mac-addr client-policy response option. |
| DHCP_CLIENT_POLICY_FAILOVER_MCLT | The failover-mclt client-policy response option. |
| DHCP_CLIENT_POLICY_FAILOVER_NUM BER_ADDRESSES | The failover-number-addresses client-policy response option. |
| DHCP_CLIENT_POLICY_FAILOVER_ REJECT_REASON | The failover-reject-reason client-policy response option. |
| DHCP_CLIENT_POLICY_FAILOVER_ RESERVED | The failover-reserved client-policy response option. |
| DHCP_CLIENT_POLICY_FAILOVER_ START_TIME_OF_STATE | The failover-start-time-of-state client-policy response option. |
| DHCP_CLIENT_POLICY_FAILOVER_ STATE_SERIAL_NUMBER | The failover-state-serial-number client-response option. |
| DHCP_CLIENT_POLICY_FILE | The file client-policy response option. |
| DHCP_CLIENT_POLICY_FINGER_ SERVERS | The finger-servers client-policy response option. |
| DHCP_CLIENT_POLICY_FLAGS | The flags client-policy response option. |
| DHCP_CLIENT_POLICY_FONT_SERVERS | The font-servers client-policy response option. |
| DHCP_CLIENT_POLICY_GIADDR | The giaddr client-policy response option. |
| DHCP_CLIENT_POLICY_HLEN | The hlen client-policy response option. |
| DHCP_CLIENT_POLICY_HOPS | The hops client-policy response option. |
| DHCP_CLIENT_POLICY_HOST_NAME | The host-name client-policy response option. |
| DHCP_CLIENT_POLICY_HOST_NAME_ CHANGED | The host-name-changed client-policy response option. |
| DHCP_CLIENT_POLICY_HOST_NAME_ IN_DNS | The host-name-in-dns client-policy response option. |
| DHCP_CLIENT_POLICY_HTYPE | The htype client-policy response option. |
| DHCP_CLIENT_POLICY_IEEE8023_ ENCAPSULATION | The ieee802.3-encapsulation client-policy response option. |
| DHCP_CLIENT_POLICY_IMPRESS_ SERVERS | The impress-servers client-policy response option. |
| DHCP_CLIENT_POLICY_INTERFACE_MTU | The interface-mtu client-policy response option. |
| DHCP_CLIENT_POLICY_IP_FORWARDING | The ip-forwarding client-policy response option. |
| DHCP_CLIENT_POLICY_IRC_SERVERS | The irc-servers client-policy response option. |

*Table A-42 DHCPOptionKeys (continued)*

| Key | Description |
|---|---|
| DHCP_CLIENT_POLICY_LEASE_RELAY_AGENT_CIRCUIT_ID | The lease-relay-agent-circuit-id client-policy response option. |
| DHCP_CLIENT_POLICY_LEASE_RELAY_AGENT_REMOTE_ID | The lease-relay-agent-remote-id client-policy response option. |
| DHCP_CLIENT_POLICY_LOG_SERVERS | The log-servers client-policy response option. |
| DHCP_CLIENT_POLICY_LPR_SERVERS | The lpr-servers client-policy response option. |
| DHCP_CLIENT_POLICY_MAC_ADDRESS | The mac-address client-policy response option. |
| DHCP_CLIENT_POLICY_MASK_SUPPLIER | The mask-supplier client-policy response option. |
| DHCP_CLIENT_POLICY_MAX_DGRAM_REASSEMBLY | The max-dgram-reassembly client-policy response option. |
| DHCP_CLIENT_POLICY_MCNS_SEC_SERVER | The mcns-sec-server client-policy response option. |
| DHCP_CLIENT_POLICY_MERIT_DUMP | The merit-dump client-policy response option. |
| DHCP_CLIENT_POLICY_MOBILE_IP_HOME_AGENTS | The mobile-ip-home-agents client-policy response option. |
| DHCP_CLIENT_POLICY_NAME_SERVERS | The name-servers client-policy response option. |
| DHCP_CLIENT_POLICY_NETBIOS_DD_SERVER | The netbios-dd-server client-policy response option. |
| DHCP_CLIENT_POLICY_NETBIOS_NAME_SERVERS | The netbios-name-servers client-policy response option. |
| DHCP_CLIENT_POLICY_NETBIOS_NODE_TYPE | The netbios-node-type client-policy response option. |
| DHCP_CLIENT_POLICY_NETBIOS_SCOPE | The netbios-scope client-policy response option. |
| DHCP_CLIENT_POLICY_NIS_DOMAIN | The nis-domain client-policy response option. |
| DHCP_CLIENT_POLICY_NIS_SERVERS | The nis-servers client-policy response option. |
| DHCP_CLIENT_POLICY_NISPLUS_DOMAIN | The nisplus-domain client-policy response option. |
| DHCP_CLIENT_POLICY_NISPLUS_SERVERS | The nisplus-servers client-policy response option. |
| DHCP_CLIENT_POLICY_NNTP_SERVERS | The nntp-servers client-policy response option. |
| DHCP_CLIENT_POLICY_NON_LOCAL_SOURCE_ROUTING | The non-local-source-routing client-policy response option. |
| DHCP_CLIENT_POLICY_NTP_SERVERS | The ntp-servers client-policy response option. |
| DHCP_CLIENT_POLICY_OP | The op client-policy response option. |
| DHCP_CLIENT_POLICY_OS_TYPE | The os-type client-policy response option. |
| DHCP_CLIENT_POLICY_PAD | The pad client-policy response option. |
| DHCP_CLIENT_POLICY_PATH_MTU_AGING_TIMEOUT | The path-mtu-aging-timeout client-policy response option. |
| DHCP_CLIENT_POLICY_PATH_MTU_PLATEAU_TABLE | The path-mtu-plateau-table client-policy response option. |

*Table A-42 DHCPOptionKeys (continued)*

| Key | Description |
|---|---|
| DHCP_CLIENT_POLICY_PERFORM_ MASK_DISCOVERY | The perform-mask-discovery client-policy response option. |
| DHCP_CLIENT_POLICY_POLICY_FILTER | The policy-filter client-policy response option. |
| DHCP_CLIENT_POLICY_POP3_SERVERS | The pop3-servers client-policy response option. |
| DHCP_CLIENT_POLICY_RELAY_AGENT_ CIRCUIT_ID | The relay-agent-circuit-id client-policy response option. |
| DHCP_CLIENT_POLICY_RELAY_AGENT_ INFO | The relay-agent-info client-policy response option. |
| DHCP_CLIENT_POLICY_RELAY_AGENT_ REMOTE_ID | The relay-agent-remote-id client-policy response option. |
| DHCP_CLIENT_POLICY_REPLY_ IPADDRESS | The reply-ipaddress client-policy response option. |
| DHCP_CLIENT_POLICY_REPLY_PORT | The reply-port client-policy response option. |
| DHCP_CLIENT_POLICY_RESOURCE_ LOCATION_SERVERS | The resource-location-servers client-policy response option. |
| DHCP_CLIENT_POLICY_RESPONSE_ PREFIX | The client policy prefix for CNR response dictionary. |
| DHCP_CLIENT_POLICY_REVERSE_ NAME_IN_DNS | The reverse-name-in-dns client-policy response option. |
| DHCP_CLIENT_POLICY_ROOT_PATH | The root-path client-policy response option. |
| DHCP_CLIENT_POLICY_ROUTER_ DISCOVERY | The router-discovery client-policy response option. |
| DHCP_CLIENT_POLICY_ROUTER_ SOLICITATION_ADDRESS | The router-solicitation-address client-policy response option. |
| DHCP_CLIENT_POLICY_ROUTERS | The routers client-policy response option. |
| DHCP_CLIENT_POLICY_SECS | The secs client-policy response option. |
| DHCP_CLIENT_POLICY_SIADDR | The siaddr client-policy response option. |
| DHCP_CLIENT_POLICY_SMTP_SERVERS | The smtp-servers client-policy response option. |
| DHCP_CLIENT_POLICY_SNAME | The sname client-policy response option. |
| DHCP_CLIENT_POLICY_STATIC_ROUTES | The static-routes client-policy response option. |
| DHCP_CLIENT_POLICY_STREETALK_ DA_SERVERS | The streetalk-da-servers client-policy response option. |
| DHCP_CLIENT_POLICY_STREETTALK_ SERVERS | The streettalk-servers client-policy response option. |
| DHCP_CLIENT_POLICY_SUBNET_MASK | The subnet-mask client-policy response option. |
| DHCP_CLIENT_POLICY_SWAP_SERVER | The swap-server client-policy response option. |
| DHCP_CLIENT_POLICY_TCP_KEEPALIVE_ GARBAGE | The tcp-keepalive-garbage client-policy response option. |
| DHCP_CLIENT_POLICY_TCP_KEEPALIVE_ INTERVAL | The tcp-keepalive-interval client-policy response option. |

*Table A-42 DHCPOptionKeys (continued)*

| Key | Description |
|---|---|
| DHCP_CLIENT_POLICY_TFTP_SERVER | The tftp-server client-policy response option. |
| DHCP_CLIENT_POLICY_TIME_OFFSET | The time-offset client-policy response option. |
| DHCP_CLIENT_POLICY_TIME_SERVERS | The time-servers client-policy response option. |
| DHCP_CLIENT_POLICY_TRAILER_ ENCAPSULATION | The trailer-encapsulation client-policy response option. |
| DHCP_CLIENT_POLICY_VENDOR_ ENCAPSULATED_OPTIONS | The vendor-encapsulated-options client-policy response option. |
| DHCP_CLIENT_POLICY_WWW_SERVERS | The www-servers client-policy response option. |
| DHCP_CLIENT_POLICY_X_DISPLAY_ MANAGER | The x-display-manager client-policy response option. |
| DHCP_CLIENT_POLICY_XID | The xid client-policy response option. |
| DHCP_CLIENT_POLICY_YIADDR | The yiaddr client-policy response option. |
| DHCP_SUPPORTED_OPTIONS_ ENVIRONMENT | Comma-separated list of DHCP environment dictionary option names. |
| DHCP_SUPPORTED_OPTIONS_INFORM | Comma-separated list of BPR inform dictionary option names. |
| DHCP_SUPPORTED_OPTIONS_PREDNS_ ENVIRONMENT | Comma-separated list of DHCP pre-dns-add-forward dictionary option names. |
| DHCP_SUPPORTED_OPTIONS_REQUEST | Comma-separated list of DHCP request dictionary option names. |
| DHCP_SUPPORTED_OPTIONS_RESPONSE | Comma-separated list of DHCP response dictionary option names. |

# DocsisDefaultKeys Interface

The DocsisDefaultsKeys interface specifies constant strings that are used as keys in map objects to set and get properties that apply to DOCSIS defaults and RDU defaults. Table A-43 lists the DOCSIS default keys.

*Table A-43 DocsisDefaultKeys*

| Key | Description |
|---|---|
| CMTS_MIC_ENABLED | The DOCSIS default strings. |
| CMTS_MIC_SHARED_SECRET | The CMTS MIC shared secret. |
| DYNAMIC_FILE_EXTENSION | Filename extension for dynamic DOCSIS files. |
| STATIC_FILE_EXTENSION | Filename extension for static DOCSIS files. |
| TFTP_MODEM_ADDRESS_OPTION_ ENABLED | Enables the TFTP modem address option. |
| TFTP_TIMESTAMP_OPTION_ENABLED | The TFTP time stamp option enabled flag. |

# DPEDetailsKeys Interface

The DPEDetails interface specifies constant strings that are used as keys in map objects returned by query methods on a DPE. Table A-44 lists the DPE details keys.

***Table A-44 DPEDetailsKeys***

| Key | Description |
|---|---|
| DPE_AGENT_HOST_ID | Host identifier of the agent associated with a DPE. |
| DPE_CONFIGS | Cached configurations of the queried DPE. |
| DPE_FILES | Cached files of the queried DPE. |
| DPE_HEALTH | Health of the queried DPE. |
| DPE_HITS | Cache hits by the queried DPE. |
| DPE_HOST_ID | Host identifier for the queried DPE. |
| DPE_MISSES | Cache misses by the queried DPE. |
| DPE_PRIMARY_PROV_GROUP_LIST | List of primary provisioning group identifiers for the queried DPE. |
| DPE_PROPERTIES | Properties for the queried DPE. |
| DPE_REQ | TFTP file requests to the TFTP server on the queried DPE. |
| DPE_SECONDARY_PROV_GROUP_LIST | List of secondary provisioning group identifiers for the queried DPE. |
| DPE_SENT | TFTP files successfully sent by the TFTP server on the queried DPE. |
| DPE_STATE | State of the queried DPE. |
| DPE_UPTIME | Uptime for the queried DPE. |
| DPE_VERSION | Software version of the queried DPE. |

# IPDeviceKeys Interface

The IPDeviceKeys interface specifies constant strings that are used as keys in properties map objects for IP device provisioning interfaces. Table A-45 lists the IP device keys.

***Table A-45 IPDeviceKeys***

| Key | Description |
|---|---|
| MUST_BE_BEHIND_DEVICE | If a device is not behind the modem or DSTB set in this property, return the device to unprovisioned status. |
| MUST_BE_IN_PROV_GROUP | If the provisioning group is not one that is specified, insert this property to return the device to the unprovisioned status.<br>**Note** If this is set to the name of a provisioning group, the device will only get provisioned access while in the designated provisioning group. |
| UNCOMMITED_CONFIG_TIME_TO_LIVE | Specifies the amount of time (in milliseconds) that a DPE should hold on to a configuration file that has not yet been committed for this device. |

# ModemKeys Interface

The ModemKeys interface specifies constant strings that are used as keys in map objects returned by query methods on DOCSIS interfaces. Table A-46 lists the modem keys.

***Table A-46 ModemKeys***

| Key | Description |
|---|---|
| CUSTOM_CPE_FIRMWARE_VERSION | Firmware version. |
| DOCSIS_MODEM_FIRMWARE_VERSION | Firmware version. |
| PROMISCUOUS_MODE_ENABLED | Tells whether the promiscuous mode is enabled or not. |

# RDUDetailsKeys Interface

The GetDetailsThisRDU interface specifies constant strings that are used as keys in map objects returned by query methods on the RDU. Table A-47 lists the RDU details keys.

***Table A-47 RDUDetailsKeys***

| Key | Description |
|---|---|
| RDU_AGENT_HOST_ID | Host identifier of the agent associated with the RDU. |
| RDU_HEALTH | Health of the RDU. |

*Table A-47 RDUDetailsKeys (continued)*

| Key | Description |
|---|---|
| RDU_HOST_ID | Host identifier for the queried RDU. |
| RDU_HOST_PORT | Port number for the queried RDU. |
| RDU_PACE_AVERAGE_BATCH_TIME | PACE average batch time for the queried RDU. |
| RDU_PACE_AVERAGE_PROCESSING_TIME | PACE average processing time for the queried RDU. |
| RDU_PACE_BATCHES_DROPPED | PACE batches dropped for the queried RDU. |
| RDU_PACE_BATCHES_FAILED | PACE batches failed for the queried RDU. |
| RDU_PACE_BATCHES_PROCESSED | PACE batches processed for the queried RDU. |
| RDU_PACE_BATCHES_SUCCEEDED | PACE batches succeeded for the queried RDU. |
| RDU_PACE_UPTIME | PACE uptime for the queried RDU. |
| RDU_PROPERTIES | Properties for the queried RDU. |
| RDU_STATE | State of the queried RDU. |
| RDU_TOTAL_COMPUTERS | Total number of computer in the database for the queried RDU. |
| RDU_TOTAL_DOCSIS_MODEMS | Total number of DOCSIS modems in the database for the queried RDU. |
| RDU_TOTAL_DSTB | Total number of DSTB devices in the database for the queried RDU. |
| RDU_TOTAL_XGCP | Total number of XGCP devices in the database for the queried RDU. |
| RDU_UPTIME | Uptime for the queried RDU. |
| RDU_VERSION | Software version of the queried RDU. |

# SearchType Interface

The SearchType interface provides an enumeration of search types. Table A-48 lists the SearchType keys.

*Table A-48 SearchType Keys*

| Key | Description |
|---|---|
| ByFQDN | References an FQDN search type. |
| ByMAC | References a MAC address search type. |
| ByMACAndFQDN | References a combined MAC address and FQDN search type. |

# ServerDefaultsKeys Interface

The ServerDefaultsKeys interface contains constants used by servers. Table A-49 lists the server defaults keys.

***Table A-49 ServerDefaultsKeys***

| Key | Description |
|---|---|
| DPE_CONFIG_POPULATION_BACKOFF | The delay times between device configuration population attempts. |
| DPE_CONFIG_POPULATION_DELAY | The time to wait between generation requests for device configurations during cache population. |
| DPE_CONFIG_REQUEST_BUFFERS | The number of buffers for holding configuration generation requests. |
| DPE_CONFIG_REQUEST_THREADS | The maximum number of concurrent configuration generation requests. |
| DPE_CONFIG_SYNCHRONIZATION_ BACKOFF | The delay times between getSyncRevNumberForIPDevices retries. |
| DPE_CONFIG_TIMEOUT | The time to wait for a response to a configuration generation request. |
| DPE_FILE_POPULATION_BACKOFF | The delay times between external file population attempts. |
| DPE_FILE_POPULATION_DELAY | The time to wait between requests for external file updates during cache population. |
| DPE_FILE_REQUEST_BUFFERS | The number of buffers for holding external file requests. |
| DPE_FILE_REQUEST_THREADS | The maximum number of concurrent external file requests. |
| DPE_FILE_SYNCHRONIZATION_ BACKOFF | The delay times between getSyncRevNumberForExternalFiles retries. |
| DPE_FILE_TIMEOUT | The time to wait for a response to an external file request. |
| DPE_MIXING_IP_ADDRESS_ENABLE_ KEY | Enables mixing the DPE IP address into the DHCP response. |
| DPE_SYNCHRONIZATION_DELAY | The time to wait between synchronization attempts. |
| DPE_SYNCHRONIZATION_TIMEOUT | The time to wait for a response to a server synchronization request. |
| DPE_TIMESERVER_ENABLE_KEY | The enable flag for the time of day server. |
| RDU_CONFIGURATION_EXTENSION_ POINT | The common RDU technology configuration extension point. |
| RDU_DEVICE_DETECTION_EXTENSION_ POINT | The RDU device detection extension point. |
| RDU_MAX_GET_SYNC_REV_NUMBERS_ FOR_IP_DEVICES | The maximum number of concurrent calls to getSyncRevNumbersForIPDevices the RDU will support. |

*Table A-49 ServerDefaultsKeys (continued)*

| Key | Description |
|---|---|
| SERVER_CONNECTION_DELAY | The time to wait between connection attempts. |
| SERVER_REGISTRATION_DELAY | The time to wait between registration attempts. |
| SERVER_REGISTRATION_TIMEOUT | The time to wait for a response to a server registration request. |
| SERVER_REGISTRATION_VERSIONINFO | Implementation-Version of the server being registered. |
| SERVER_SYSLOG_ENABLE | The log level for the syslog (standard out). |
| SERVER_TRACE_AGENT | Represents all the tracing properties that can be set via the API. Set the flag to enabled to enable tracing category, and to disabled to disable it. The disable flag is set by default. |
| UNCOMMITTED_CONFIG_TIME_TO_ LIVE | Specifies the default amount of time, in milliseconds, a DPE should hold on to a configuration file that has not been committed yet. |

# SNMPPropertyKeys Interface

The SNMP Property Keys interface specifies character strings that are used as keys to store and retrieve values associated with devices that respond to SNMP requests. Table A-50 lists the SNMPPropertyKeys.

*Table A-50 SNMPPropertyKeys*

| Key | Description |
|---|---|
| READ_COMMUNITY_STRING | Gets the SNMP read community string. |
| WRITE_COMMUNITY_STRING | Gets the SNMP write community string. |

# TechnologyDefaultsKeys Interface

The TechnologyDefaultsKeys interface specifies constant strings that are used as keys in map objects to set and get properties that apply to technology defaults. Table A-51 lists the TechnologyDefaultsKeys.

*Table A-51 TechnologyDefaultsKeys*

| Key | Description |
|---|---|
| COMPUTER_DEFAULT_CLASS_ OF_SERVICE | The default class of service for computers. |
| COMPUTER_DEFAULT_DHCP_CRITERIA | The default DHCP criteria for computers. |
| COMPUTER_DEFAULT_PROV_ PROMISCUOUS_DHCP_CRITERIA | The default provisioned promiscuous mode dhcp criteria for the computer technology. |
| COMPUTER_DISRUPTION_EXTENSION_ POINT | The device disruption extension point for computers. |
| COMPUTER_EXTENSION_POINT | The extension point for computers. |

*Table A-51 TechnologyDefaultsKeys (continued)*

| Key | Description |
|---|---|
| CUSTOM_CPE_DEFAULT_CLASS_OF_SERVICE | The default class of service for custom CPE devices. |
| CUSTOM_CPE_DEFAULT_DHCP_CRITERIA | The default DHCP criteria for custom CPE devices. |
| CUSTOM_CPE_DISRUPTION_EXTENSION_POINT | The device disruption extension point for custom CPE devices. |
| CUSTOM_CPE_EXTENSION_POINT | The extension point for custom CPE devices. |
| DOCSIS_DEFAULT_CLASS_OF_SERVICE | The default class of service for DOCSIS devices. |
| DOCSIS_DEFAULT_DHCP_CRITERIA | The default DHCP criteria for DOCSIS devices. |
| DOCSIS_DISRUPTION_EXTENSION_POINT | The device disruption extension point for DOCSIS devices. |
| DOCSIS_EXTENSION_POINT | The extension point for DOCSIS devices. |
| DSTB_DEFAULT_CLASS_OF_SERVICE | The default class of service for DSTB devices |
| DSTB_DEFAULT_DHCP_CRITERIA | The default DHCP criteria for DSTB devices. |
| DSTB_DISRUPTION_EXTENSION_POINT | The device disruption extension point for DSTB devices. |
| DSTB_EXTENSION_POINT | The extension point for DSTB devices. |
| XGCP_SIGNALING_TYPE | The XGCP signaling type: S = Simple |
| XGCP_USE_OLD_STYLE | Specified whether to use the old format for merit-dump string. |
| XGCP_VERSION_NUMBER | The version number. |

# UserDetailsKeys Interface

The UserDetailsKeys interface specifies constant strings that are used as keys in map objects returned by query methods on user objects. Table A-52 lists the user details keys.

*Table A-52 UserDetailsKeys*

| Key | Description |
|---|---|
| USER_DESCRIPTION | User description for the queried user object. |
| USER_ID | User ID for the queried user object. |
| USER_ISADMIN | Checks if the user an administrator for the queried user object. |
| USER_PASSWORD | User password for the queried user object. |

# VOIPServiceDetailsKeys Interface

The VOIPServiceDetailsKeys interface specifies constant strings that are used as keys in map objects returned by query methods on VoIP services. Table A-53 lists the VoIP service details keys.

***Table A-53 VOIPServiceDetailsKeys***

| Key | Description |
|---|---|
| CMS_FQDN | CMS FQDN for the queried service. |
| CMS_PORT_NUMBER | CMS port number for the queried service. |
| CMS_QUALIFIER | CMS qualifier for the queried service. |
| PROPERTIES | Properties for the queried service. |
| TELEPHONE_NUMBER | Telephone number for the queried service. |

# XGCPCommandStatusCodes Interface

The XGCPCommandStatusCodes interface specifies constants that are used to interpret returned xGCP command and query status information. The CommandStatus.getStatusCode() command returns values to indicate that an xGCP command processing error occurred. Command failure occurs as a result of these causes:

- The xGCP service already exists
- The command is trying to operate on an unknown xGCP service.

# XGCPProvisioningKeys Interface

The XGCPProvisioningKeys interface specifies the constants that are used to interpret and set the values associated with xGCP devices, for example, the SGCP version string (SGCP_VERSION). Table A-54 lists the XGCPProvisioningKeys.

***Table A-54 XGCPProvisioningKeys***

| Key | Description |
|---|---|
| XGCP_SIGNALING_TYPE | Signaling type for an xGCP port. |
| XGCP_USE_OLD_STYLE | Indicates whether to use the old style merit-dump string. |
| XGCP_VERSION_NUMBER | xGCP version used for an xGCP port. |

# com.cisco.provisioning.cpe.events Package

The com.cisco.provisioning.cpe.events package contains the interfaces and methods used to manage events. Table A-55 lists the interfaces in the com.cisco.provisioning.cpe.events package.

***Table A-55 The com.cisco.provisioning.cpe.events Package Interfaces***

| Interface | Description |
|---|---|
| BatchEvent | Defines the batch events. |
| BatchListener | The listener interface for BatchEvents. |
| ProvAPI | The base interface from which individual event category objects inherit. |
| ProvAPIListener | Tags event listeners as being associated with the BPR event subsystem. All BPR event listeners inherit from this interface. |
| ProvAPIManager | Outlines the methods for managing events. |
| COSEvent | The interface that defines the object to be returned as a result of a class of service event notification. |
| COSListener | The listener interface for COSEvents. |
| DeviceEvent | Defines the object to be returned as a result of a DeviceEvent notification. |
| DeviceListener | The listener interface for DeviceEvent. |
| DHCPCriteriaEvent | The interface that defines the object to be returned as a result of a DHCPCriteriaEvent notification. |
| DHCPCriteriaListener | The listener interface for DeviceCriteriaEvent. |
| ExternalFileEvent | Defines the object to be returned as a result of an ExternalFileEvent notification. |
| ExternalFileListener | The listener interface for ExternalFileEvent. |
| MessagingEvent | Defines the object to be returned as a result of a MessagingEvent notification. |
| MessagingListener | The listener interface for MessagingEvent. |
| ProvGroupEvent | Defines the object to be returned as a result of an event in a provisioning group. |
| ProvGroupListener | The listener interface for ProvGroupEvent. |
| SystemConfigEvent | The interface that defines the object to be returned as a result of a SystemConfig event notification. |
| SystemConfigListener | The listener interface for SystemConfigEvent. |

Table A-56 lists the com.cisco.provisioning.cpe.events classes.

*Table A-56 The com.cisco.provisioning.cpe.events Classes*

| Class | Description |
|---|---|
| BatchAdapter | An abstract class that provides no-op definitions of all methods in the BatchListener interface. |
| BatchEventQualifier | An implementation of the qualifier class for use with BatchEvents. |
| COSAdapter | The listener interface for DeviceMod events. |
| COSEventQualifier | An implementation of the qualifier class for use with COSEvents. |
| DeviceAdapter | An abstract class that provides no-op definitions of all methods in the DeviceListener interface. |
| DeviceEventQualifier | An implementation of the qualifier class for use with DeviceEvents. |
| DHCPCriteriaAdapter | The listener interface for DeviceMod events. |
| DHCPCriteriaEventQualifier | An implementation of the qualifier class for use with DHCPCriteriaEvents. |
| ExternalFileAdapter | An abstract class that provides no-op definitions of all methods in the ExternalFileListener interface. |
| ExternalFileEventQualifier | An implementation of the qualifier class for use with ExternalFileEvents. |
| MessagingAdapter | An abstract class that provides no-op definitions of all methods in the MessagingListener interface. |
| MessagingQualifier | An implementation of the qualifier class for use with Messaging events. |
| ProvGroupAdapter | An abstract class that provides no-op definitions of all methods in the ProvGroupListener interface. |
| ProvGroupEventQualifier | An implementation of the qualifier class for use with ProvGroupEvents. |
| Qualifier | The class that determines whether or not to notify the associated listener for a given event. |
| QualifyAll | An implementation of the Qualifier class that returns true all the time. Therefore, all listener's that register with this qualifier receive all events. |
| ServerConfigEventType | Provides an enumeration of SystemConfigEventTypes signaled by a SystemConfigEvent of subclass SystemConfigListener.SERVER_DEFAULTS_CHANGE. |
| SystemConfigAdapter | An abstract class that provides no-op definitions of all methods in the SystemConfigListener interface. |
| SystemConfigEventQualifier | This implementation of the Qualifier class is to be used with SystemConfigEvents. |
| SystemConfigEventType | Provides an enumeration of SystemConfigEventTypes signaled by SystemConfigEvent. |

Table A-57 lists the com.cisco.provisioning.cpe.events exceptions.

*Table A-57 The com.cisco.provisioning.cpe.events Exceptions*

| Exception | Description |
|---|---|
| RegistrationFailedException | Thrown when registration of an event listener fails. |

The following sections provide a summary of the com.cisco.provisioning.cpe.events interfaces, classes, and exceptions.

# BatchEvent Interface

The BatchEvent interface defines batch events. Table A-58 lists the BatchEvent interface methods.

*Table A-58 BatchEvent Methods*

| Method | Description |
|---|---|
| getBatchStatus() | Returns the status of the batch. |

# BatchListener Interface

The BatchListener interface is the listener interface for batch events. Table A-59 lists the BatchListener events.

*Table A-59 BatchListener Methods*

| Method | Description |
|---|---|
| completion() | The method invoked when a batch event arrives. |

# ProvAPIEvent Interface

The ProvAPI interface is the base interface from which individual event category objects inherit. Table A-60 lists the ProvAPI methods.

*Table A-60 ProvAPI Methods*

| Method | Description |
|---|---|
| getID() | Returns the identifier of the specific event category member. |
| getSource() | Returns the source of the specific event. |
| setID() | Sets the identifier of the specific event category member. |
| setSource() | Sets the source of the specific event. |
| toString() | Returns a string representation of this object. |

# ProvAPIEventListener Interface

The ProvAPIListener interface tags the listening entity as associated with the event subsystem. All event listeners inherit from this interface. Table A-61 lists the ProvAPIListener methods.

***Table A-61 ProvAPIListener Methods***

| Method | Description |
|---|---|
| getOneShot() | Gets the oneShot mode value, specifying whether or not the listener is registered for a single occurrence of the events. |
| setOneShot() | Sets the oneShot mode value, specifying whether or not the listener is registered for a single occurrence of the events. |

# ProvAPIEventManager Interface

The ProvAPIManager contains the methods for managing events. Table A-62 lists the ProvAPIManager methods.

***Table A-62 ProvAPIManager Methods***

| Method | Description |
|---|---|
| addBatchListener() | Adds a BatchListener interface. |
| addCOSListener() | Adds a COSListener interface to receive those COSEvents that satisfy the given Qualifier. |
| addDeviceListener() | Adds a DeviceListener interface. |
| addDHCPCriteriaListener() | Adds a DHCPCriteriaListener interface to receive those DHCPCriteriaEvents that satisfy the given Qualifier. |
| addExternalFileListener() | Adds an ExternalFileListener interface. |
| addMessagingListener() | Adds a MessagingListener interface. |
| addProvGroupListener() | Adds a ProvGroupListener interface. |
| addSystemConfigListener() | Adds a SystemConfigListener interface. |
| fireBatchEvent() | Notifies all registers that a BatchEvent has occurred. |
| fireCOSEvent() | Notifies all registered listeners that the class of service event has occurred. |
| fireDeviceEvent() | Notifies all registers that a DeviceEvent has occurred. |
| fireDHCPCriteriaEvent() | Notifies all registered listeners that the given event has occurred. |
| fireExternalFileEvent() | Notifies all registered listeners that the given event has occurred. |
| fireExternalFileEvent() | Notifies all registers that a ExternalFileEvent has occurred. |
| fireMessagingEvent() | Notifies all registers that a MessagingEvent has occurred. |
| fireProvGroupEvent() | Notifies all registers that a ProvGroupEvent has occurred. |
| fireSystemConfigEvent() | Notifies all registers that a SystemConfigEvent has occurred. |
| removeBatchListener() | Stops the BatchListener from receiving event notifications. |

*Table A-62 ProvAPIManager Methods (continued)*

| Method | Description |
|---|---|
| removeCOSListener() | Stops the given COSListener from receiving event notifications of those events that satisfy the given qualifier. |
| removeDeviceListener() | Stops the DeviceListener from receiving event notifications. |
| removeDHCPCriteriaListener() | Stops the given DHCPCriteriaListener from receiving event notifications of events that satisfy the given qualifier. |
| removeExternalFileListener() | Stops the ExternalFileListener from receiving event notifications. |
| removeMessagingListener() | Stops the MessagingListener from receiving event notifications. |
| removeProvGroupListener() | Stops the ProvGroupListener from receiving event notifications. |
| removeSystemConfigListener() | Stops the SystemConfigListener from receiving event notifications. |

# COSEvent Interface

The COSEvent interface contains the methods that define the object returned as a result of a class of service event notification. Table A-63 lists the COSEvent interface methods.

*Table A-63 COSEvent Interface Methods*

| Method | Description |
|---|---|
| getName() | Returns the name of the class of service. |
| getProperties() | Returns the properties of the class of service. |
| getType() | Returns the type of the class of service. |

# COSListener Interface

The COSListener is the listener interface for device modification (G50

) events.

*Table A-64 COSListener Methods*

| Method | Description |
|---|---|
| deletedCOS() | The method invoked when a COSEvent arrives after a class of service is deleted. |
| newCOS() | The method invoked when a COSEvent arrives after a class of service is added. |

# DeviceEvent Interface

The DeviceEvent interface defines the object to be returned after a DeviceMod event notification. Table A-65 lists the DeviceEvent methods.

***Table A-65 DeviceEvent Methods***

| Method | Description |
|---|---|
| getDeviceFQDN() | Returns the FQDN of the device. |
| getDeviceID() | Returns the device identifier (MAC address). |
| getDeviceIP() | Returns the IP address of the device. |
| getNewData() | Returns the new data for the event including IP address, device identifier, class of service, and provisioning group. |
| getOldData() | Returns the old data for the event including IP address, device identifier, class of service, and provisioning group. |

# DeviceListener Interface

The DeviceListener interface is the listener interface for DeviceMod events. Table A-66 lists the DeviceListener methods.

***Table A-66 DeviceListener Methods***

| Method | Description |
|---|---|
| changedCoS() | Invoked when a DeviceEvent arrives after a new device receives a new class of service. |
| changedIP() | Invoked when a DeviceEvent arrives after a device receives a new IP address. |
| deletedDevice() | Invoked when a DeviceEvent arrives after the deletion of a device. |
| deletedVoiceService() | Invoked when a DeviceEvent arrives as a result of a deletion of voice service. |
| newProvDevice() | Invoked when a DeviceEvent arrives after the provisioning of a new device. |
| newUnprovDevice() | Invoked when a DeviceEvent arrives after the provisioning of a new, unprovisioned device. |
| newVoiceService() | Invoked when a DeviceEvent arrives after the addition of a new voice service. |
| roaming() | Invoked when a DeviceEvent arrives after relocation of a device. |

# DHCPCriteriaEvent

The DHCPCriteriaEvent interface defines the object returned as a result of a DHCP criteria event notification. Table A-67 lists the DHCPCriteriaEvent methods.

***Table A-67 DHCPCriteriaEvent Interface***

| Method | Description |
|---|---|
| getClientClass() | Returns the client class. |
| getExcludeSelectionTags() | Returns the exclude selection tags. |
| getIncludeSelectionTags() | Returns the include selection tags. |
| getName() | Returns the DHCP criteria name. |
| getProperties() | Returns the DHCPcriteria properties. |

# DHCPCriteriaListener

The DHCPCriteriaListener interface is the listener interface for DeviceMod events. Table A-68 lists the DHCPCriteriaListener methods.

***Table A-68 DHCPCriteriaListener Interface***

| Method | Description |
|---|---|
| deletedDHCPCriteria() | The method invoked when a DHCPCriteriaEvent arrives after a DHCP criteria is deleted. |
| newDHCPCriteria() | The method invoked when a DHCPCriteriaEvent arrives after a DHCP criteria is added. |

# ExternalFileEvent Interface

The ExternalFileEvent interface defines the object to be returned after an ExternalFile event notification. Table A-69 lists the ExternalFileEvent methods.

***Table A-69 ExternalFileEvent Methods***

| Method | Description |
|---|---|
| getFilename() | Returns the name of the external file. |

# MessagingEvent Interface

The MessagingEvent interface defines the object to be returned after a MessagingEvent notification. Table A-70 lists the MessagingEvent methods.

***Table A-70 MessagingEvent Methods***

| Method | Description |
|---|---|
| getAddress() | Returns the internet address of this connection. |
| getPersistence() | Returns the persistence of the connection. |
| getPort() | Returns the port number of this connection. |

# MessagingListener Interface

The MessagingListener interface is the listener interface for MessagingEvents. Table A-71 lists the MessagingListener methods.

***Table A-71 MessagingListener Methods***

| Method | Description |
|---|---|
| connectionStarted() | Invoked when a MessagingEvent arrives after a connection starts. |
| connectionStopped() | Invoked when a MessagingEvent arrives after a connection stops. |

# ProvGroupEvent Interface

The ProvGroupEvent interface defines the object to be returned after an event in a provisioning group. Table A-72 lists the ProvGroupEvent methods.

***Table A-72 ProvGroupEvent Methods***

| Method | Description |
|---|---|
| getProvGroupID() | Returns the identifier of the provisioning group. |
| getProvisioningGroup() | Returns the attribute tree containing information about the provisioning group. |

# ProvGroupListener Interface

The ProvGroupListener interface is the listener for ProvGroup events. Table A-73 lists the ProvGroupListener methods.

***Table A-73 ProvGroupListener Methods***

| Method | Description |
|---|---|
| modified() | Invoked when a ProvGroup event arrives. |

# Qualifier Interface

Use the Qualifier interface to determine whether to notify the associated listener for a given event. Table A-74 lists the Qualifier methods.

***Table A-74 Qualifier Methods***

| Method | Description |
|---|---|
| qualifies() | Returns true after a determination that an event satisfies the requirements for notifying the associated listener. |

# SystemConfigEvent Interface

The SystemConfigEvent interface defines the object to be returned after a SystemConfig event notification. Table A-75 lists the SystemConfigEvent methods.

***Table A-75 SystemConfigEvent Methods***

| Method | Description |
|---|---|
| getAddedAndChanged Settings() | Returns the new or changed system defaults or properties. |
| getRemovedSettings() | Returns the removed system defaults or properties. |
| getServerHostID() | Returns the server host identifier, if the SystemConfigEvent identifier indicates a change to a particular server. |
| getType() | Returns the SystemConfigEvent type. |

# SystemConfigListener Interface

The SystemConfigListener interface is the listener for ExternalFile events. Table A-76 lists the SystemConfigListener methods.

***Table A-76 SystemConfigListener Methods***

| Method | Description |
|---|---|
| serverDefaultsChanged() | Invoked when a SystemConfigEvent arrives after a change in server defaults. |
| serverPropertiesChanged() | Invoked when a SystemConfigEvent arrives after a change in a server's properties. |
| systemConfigChanged() | Invoked when a SystemConfigEvent arrives after a change in system configuration. |
| systemDefaultsChanged() | Invoked when a SystemConfigEvent arrives after a change in system defaults. |

# BatchAdapter Class

The BatchAdapter class provides definitions of all methods in the BatchListener interface. Table A-77 lists the BatchAdapter methods.

***Table A-77 BatchAdapter Methods***

| Method | Description |
|---|---|
| completion() | Invoked when a BatchEvent arrives. |
| getOneShot() | Gets the oneShot mode value, which specifies whether the listener is registered for just one occurrence of the event. |
| setOneShot() | Sets the oneShot mode value, which specifies that the registration request is for just one occurrence of the event. |

# BatchEventQualifier Class

This implementation of the BatchEventQualifier class is used with BatchEvents. Table A-78 lists the BatchEventQualifier methods.

***Table A-78 BatchEventQualifier Methods***

| Method | Description |
|---|---|
| qualifies() | Returns true if the implementation has determined that this event satisfies the requirements to notify the associated listener. |
| setBatchID() | Sets the qualifies method to return true to all BatchEvents related to the Batch specified. |

# COSEventQualifier Class

This implementation of the COSEventQualifier class is used with COSEvents. Table A-79 lists the COSEventQualifier methods.

***Table A-79 COSEventQualifier Type Methods***

| Method | Description |
|---|---|
| clearAll() | Clears all options that have been set. |
| qualifies() | Returns true if the implementation has determined that this event satisfies the requirements to notify the associated listener. |
| setDeletedCOS() | Sets the qualifies method to return true to all COSEvents that are DELETED_COS sub-events. |
| setNewCOS() | Sets the qualifies met. |

# DeviceAdapter Class

The DeviceAdapter class provides no-op definitions of all methods in the DeviceListener interface. Table A-80 lists the DeviceAdapter methods.

***Table A-80 DeviceAdapter Methods***

| Method | Description |
|---|---|
| changedIP() | Invoked when a DeviceEvent arrives after a device receives a new IP address. |
| deletedDevice() | Invoked when a DeviceEvent arrives after a device has been deleted. |
| deletedVoiceService() | Invoked when a DeviceEvent arrives after a voice service has been deleted. |
| getOneShot() | Gets the oneShot mode value, which specifies whether the listener is registered for just one occurrence of the event. |
| newCos() | Invoked when a DeviceEvent arrives after a new device receives a new class of service. |
| newProvDevice() | Invoked when a DeviceEvent arrives after a new device receives provisioning. |
| newUnprovDevice() | Invoked when a DeviceEvent arrives after the detection of a new, unprovisioned device. |
| newVoiceService() | Invoked when a DeviceEvent arrives after a voice service has been added. |
| roaming() | Invoked when a DeviceEvent arrives after the relocation of a device. |
| setOneShot() | Sets the oneShot mode value, which specifies that the registration request is for just one occurrence of the event. |

# DeviceEventQualifier Class

This implementation of the DeviceEventQualifier class is used with DeviceEvents. Table A-81 lists the SystemConfigEventType methods.

***Table A-81 DeviceEventQualifier Type Methods***

| Method | Description |
|---|---|
| clearAll() | Clears all options that have been set. |
| qualifies(ProvAPIEvent event) | Returns true if the implementation has determined that this event satisfies the requirements to notify the associated listener. |
| setChangedCOS() | Sets the qualifies method to return true to all DeviceEvents that are CHANGED_COS subevents. |
| setChangedIP() | Sets the qualifies method to return true to all DeviceEvents that are CHANGED_IP subevents. |
| setDeletedDevice() | Sets the qualifies method to return true to all DeviceEvents that are DELETED_DEVICE subevents. |

*Table A-81 DeviceEventQualifier Type Methods (continued)*

| Method | Description |
|---|---|
| setDeletedVoiceService() | Sets the qualifies method to return true to all DeviceEvents that are DELETED_VOICE_SERVICE subevents. |
| setDeviceID(java.lang.String deviceMac) | Sets the qualifies method to return true to the DeviceEvents related to the device specified. |
| setNewDevice() | Sets the qualifies method to return true to all DeviceEvents that corresponds to addition of new devices (both registered and unregistered). |
| setNewProvDevice() | Sets the qualifies method to return true to all DeviceEvents that are NEW_PROV_DEVICE subevents. |
| setNewUnprovDevice() | Sets the qualifies method to return true to all DeviceEvents that are NEW_UNPROV_DEVICE subevents. |
| setNewVoiceService() | Sets the qualifies method to return true to all DeviceEvents that are NEW_VOICE_SERVICE subevents. |
| setRoamingDevice() | Sets the qualifies method to return true to all DeviceEvents that are ROAMING_DEVICE subevents. |

# DHCPCriteriaEventQualifier Class

This implementation of the DHCPCriteriaEventQualifier class is used with DHCPCriteriaEvents. Table A-82 lists the DHCPCriteriaEventQualifier methods.

*Table A-82 DHCPCriteriaEventQualifier Methods*

| Method | Description |
|---|---|
| clearAll() | Clears all options that have been set. |
| qualifies(ProvAPIEvent event) | Returns true if the implementation has determined that this event satisfies the requirements to notify the associated listener. |
| setDeletedDHCPCriteria() | Sets the qualifies method to return true to all DHCPCriteriaEvents that are DELETED_DHCP_CRITERIA subevents. |
| setNewDHCPCriteria() | Sets the qualifies method to return true to all DHCPCriteriaEvents that corresponds to addition of new DHCP criteria (NEW_DHCP_CRITERIA subevents). |

# ExternalFileAdapter Class

The ExternalFileAdapter class provides no-op definitions of all methods in the ExternalFileListener interface. Table A-83 lists the ExternalFileAdapter methods.

***Table A-83 ExternalFileAdapter Methods***

| Method | Description |
|---|---|
| added() | Invoked when an ExternalFile event arrives after the creation of a new external file. |
| deleted() | Invoked when an ExternalFile event arrives after the deletion of an external file. |
| getOneShot() | Gets the oneShot mode value, which specifies whether the listener is registered for just one occurrence of the event. |
| replaced() | Invoked when an ExternalFile event arrives after the replacement of an external file. |
| setOneShot() | Sets the oneShot mode value, which specifies that the registration request is for just one occurrence of the event. |

# ExternalFileEventQualifier Class

This implementation of the ExternalFileEventQualifier class is used with ExternalFileEvents. Table A-84 lists the e ExternalFileEventQualifier type methods.

***Table A-84 ExternalFileEventQualifier Methods***

| Method | Description |
|---|---|
| clearAll() | Clears all options that have been set. |
| qualifies(ProvAPIEvent event) | Returns true if the implementation has determined that this event satisfies the requirements to notify the associated listener. |
| setDeletedExternalFile() | Sets the qualifies method to return true to all ExternalFileEvents that are DELETE_FILE subevents. |
| setNewExternalFile() | Sets the qualifies method to return true to all ExternalFileEvents that corresponds to addition of a new External File (ADD_FILE subevents). |
| setReplaceExternalFile() | Sets the qualifies method to return true to all ExternalFileEvents that are REPLACE_FILE subevents. |

# MessagingAdapter Class

The MessagingAdapter class provides definitions of all methods in the MessagingListener interface. Table A-85 lists the MessagingAdapter methods.

***Table A-85 MessagingAdapter Methods***

| Method | Description |
|---|---|
| connectionStarted() | Invoked when a MessagingEvent arrives after a connection starts. |
| connectionStopped() | Invoked when a MessagingEvent arrives after a connection stops. |
| getOneShot() | Gets the oneShot mode value that specifies whether the listener is registered for just one occurrence of the event. |
| setOneShot() | Sets the oneShot mode value that specifies a registration request for just one occurrence of the event. |

# MessagingQualifier Class

The MessagingQualifier class is the implementation of the Qualifier class to be used with the MessagingEvents interface. Table A-86 lists the MessagingQualifier methods.

***Table A-86 MessagingQualifier Methods***

| Method | Description |
|---|---|
| clearAll() | Clears all options. |
| qualifies() | Returns true if the implementation has determined that this event satisfies the requirements to notify the associated listener. |
| setAllConnectionsDown() | Sets the qualifies method to return true to all MessagingEvents that are CONNECTION_DOWN subevents. |
| setAllPersistentConnectionsDown() | Sets the qualifies method to return true to all MessagingEvents that are CONNECTION_DOWN subevents and are persistent connections. |

# ProvGroupAdapter Class

The ProvGroupAdapter class provides definitions of all methods in the ProvGroupListener interface. Table A-87 lists the ProvGroupAdapter methods.

***Table A-87 ProvGroupAdapter Methods***

| Method | Description |
|---|---|
| getOneShot() | Gets the oneShot mode value that specifies whether the listener is registered for just one occurrence of the event. |

*Table A-87 ProvGroupAdapter Methods (continued)*

| Method | Description |
|---|---|
| modified() | Invoked when a ProvGroup event arrives and the event identifier is PROV_GROUP_MODIFIED. |
| setOneShot() | Sets the oneShot mode value that specifies a registration request for just one occurrence of the event. |

# QualifyAll Class

The QualifyAll class is an implementation of the Qualifier class. All listeners that register with this qualifier receive all events. Table A-88 lists the QualifyAll methods.

*Table A-88 QualifyAll Methods*

| Method | Description |
|---|---|
| qualifies() | Returns true if the implementation has determined that this event satisfies the requirements to notify the associated listener. |

# SystemConfigAdapter Class

The SystemConfigAdapter class provides definitions of all methods in the SystemConfigListener interface. Table A-89 lists the SystemConfigAdapter methods.

*Table A-89 SystemConfigAdapter Methods*

| Method | Description |
|---|---|
| getOneShot() | Gets the oneShot mode value that specifies whether the listener is registered for just one occurrence of the event. |
| serverDefaultsChanged() | Invoked when a SystemConfigEvent arrives after a change in server defaults. |
| serverPropertiesChanged() | Invoked when a SystemConfigEvent arrives after a change in server properties. |
| setOneShot() | Sets the oneShot mode value that specifies a registration request for just one occurrence of the event. |
| systemConfigChanged() | Invoked when a SystemConfigEvent arrives after a change in system configuration. |
| systemDefaultsChanged() | Invoked when a SystemConfigEvent arrives after a change in system defaults. |

# SystemConfigEventQualifier Class

This implementation of the SystemConfigEventQualifier class is used with SystemConfigEvents. Table A-90 lists the SystemConfigEventQualifier methods.

***Table A-90 SystemConfigEventQualifier Methods***

| Method | Description |
|---|---|
| clearAll() | Clears all options that have been set. |
| qualifies(ProvAPIEvent event) | Returns true if the implementation has determined that this event satisfies the requirements to notify the associated listener. |
| setServerDefaultsChange() | Sets the qualifies method to return true to all SystemConfigEvents that are SERVER_DEFAULTS_CHANGE sub-events. |
| setServerPropertiesChange() | Sets the qualifies method to return true to all SystemConfigEvents that are SERVER_PROPERTIES_CHANGE sub-events. |
| setSystemConfigChange() | Sets the qualifies method to return true to all SystemConfigEvents that are SYSTEM_CONFIG_CHANGE sub-events. |
| setSystemDefaultsChange() | Sets the qualifies method to return true to all SystemConfigEvents that are SYSTEM_DEFAULTS_CHANGE sub-events. |

# ServerConfigEventType Class

The ServerConfigEventType class provides an enumeration of the SystemConfigEventTypes that can be signalled by a SystemConfigEvent of subclass SystemConfigListener.SERVER_DEFAULTS_CHANGE. Table A-91 lists the SystemConfigEventType methods.

***Table A-91 ServerConfigEventType Methods***

| Method | Description |
|---|---|
| getNamed() | Returns the ServerConfigEventType object with the selected name (or null if it does not exist). |
| getValued() | Returns the ServerConfigEventType object with the selected value (or null if it does not exist). |

# Exceptions

The com.cisco.provisioning.cpe.events package implements this exception to handle processing of errors and other abnormal conditions:

- RegistrationFailedException—Thrown when registration of an event listener fails.


# Deprecated Methods

The CSRC Device Provisioning Registrar (CSRC DPR) methods are a subset of the Broadband Provisioning Registrar (BPR) methods. The calls listed in this appendix have been deprecated in BPR. This means that the CSRC DPR methods may not be supported in your release of the provisioning API. Table B-1 compares the interfaces.

***Table B-1 Comparison of Interfaces***

| CSRC DPR com.cisco.provisioning.cpe.api | BPR com.cisco.provisioning.cpe.api |
|---|---|
| Configuration | Configuration |
| | CustomCPE (new in BPR) |
| DeviceSearch | DeviceSearch |
| DSTB | DSTB |
| Provisioning | Provisioning |
| Provisioning (Computers) | Computer |
| Provisioning (DOCSIS modems) | DOCSIS |
| XGCP | XGCP |

Table B-2 lists the methods deprecated from CSRC DPR and their equivalents in BPR.

***Table B-2 Methods Deprecated from the Provisioning Interface***

| CSRC DPR Provisioning Interface and Method | BPR Interface and Method |
|---|---|
| addComputer | Computer.addComputer |
| addComputerByIPAddress | Computer.addComputerByIPAddress |
| addDOCSISModem | DOCSIS.addDOCSISModem |
| addDOCSISModemByIPAddress | DOCSIS.addDOCSISModemByIPAddress |
| addXGCPModemMTA | DOCSIS.addDOCSISModem |
| addXGCPModemMTAByIPAddress | DOCSIS.addDOCSISModemByIPAddress |
| changeComputerClassOfService | Computer.changeComputerClassOfService |
| changeComputerClientClass | Computer.changeComputerDHCPCriteria |
| changeComputerFQDN | Computer.changeComputerFQDN |

*Table B-2 Methods Deprecated from the Provisioning Interface (continued)*

| CSRC DPR Provisioning Interface and Method | BPR Interface and Method |
|---|---|
| changeComputerMACAddress | Computer.changeComputerMACAddress |
| changeComputerOwnerID | Computer.changeComputerOwnerID |
| changeComputerProperties | Computer.changeComputerProperties |
| changeDOCSISModemClassOf Service | DOCSIS.changeDOCSISModemClassOfService |
| changeDOCSISModemClientClass | DOCSIS.changeDOCSISModemDHCPCriteria |
| changeDOCSISModemFQDN | DOCSIS.changeDOCSISModemFQDN |
| changeDOCSISModemMACAddress | DOCSIS.changeDOCSISModemMACAddress |
| changeDOCSISModemOwnerID | DOCSIS.changeDOCSISModemOwnerID |
| changeDOCSISModemProperties | DOCSIS.changeDOCSISModemProperties |
| changeDSTBDHCPCriteria | DSTB.changeDSTBDHCPCriteria |
| changeXGCPModemMTAClient Class | DOCSIS.changeDOCSISModemDHCPCriteria |
| changeXGCPModemMTACPE SelectionTags | DOCSIS.changeDOCSISModemCPEDHCPCriteria |
| changeXGCPModemMTADevice ClassOfService | No longer in use. |
| changeXGCPMotemMTAFQDN | DOCSIS.changeDOCSISModemFQDN |
| changeXGCPModemMTAClassOf Service | DOCSIS.changeDOCSISModemClassOfService |
| changeXGCPModemMTAOwnerID | DOCSIS.changeDOCSISModemMTAOwnerID |
| changeXGCPModemMTAProperties | DOCSIS.changeDOCSISModemMTAProperties |
| deleteComputer | Computer.deleteComputer |
| deleteDOCSISModem | DOCSIS.deleteDOCSISModem |
| deleteXGCPModemMTA | DOCSIS.deleteDOCSISModem |
| getDetailsForComputer | Computer.getDetailsForComputer |
| getDetailsForComputerByIPAddress | Computer.getDetailsForComputerByIPAddress |
| getDetailsForDOCSISModem | DOCSIS.getDetailsForDOCSISModem |
| getDetailsForDOCSISModemByIP Address | DOCSIS.getDetailsForDOCSISModemByIPAddress |
| getDetailsForXGCPModemMTA | DOCSIS.getDetailsForDOCSISModem |
| getDetailsForXGCPModemMTA ByIPAddress | DOCSIS.getDetailsForDOCSISModemByIPAddress |


## A

**active logs** These log files contain data that has not yet written into the database. It is important to keep active log files until they become redundant. *See also* redundant logs and removable logs.

**administrator user interface** The administrative application used to manage and configure broadband network devices.

**agent** A process that resides in all managed devices and reports the values of specified variables to management stations.

**alert** Message notifying an operator or administrator of a network problem.

**API** Application programming interface. Specification of function-call conventions that defines an interface to a service.

## C

**cable** Transmission medium of copper wire or optical fiber wrapped in a protective cover.

**cable modem termination system** *See* CMTS.

**caching** Form of replication in which information learned during a previous transaction is used to process later transactions.

**client class** Device groupings determined by a service provider's service definitions.

**CMTS** Cable modem termination system. A CMTS is a component that exchanges digital signals with cable modems on a cable network. The CMTS is usually located in the cable provider's local office.

**CMTS shared secret** *See* shared secret.

**configuration file** A file containing configuration parameters for the DOCSIS cable modem.

**CPE** Customer premises equipment. Terminating equipment, such as telephones and modems, supplied and installed at a customer location.

**customer premises equipment** *See* CPE.

## D

**Data-Over-Cable Systems Interface Specifications** *See* DOCSIS.

**demonstration interface** A program that demonstrates how to call the provisioning API to perform various tasks, through a web-based interface.

**device provisioning engine** *See* DPE.

**DHCP** Dynamic Host Configuration Protocol (RFC 2131). The DHCP server dispenses and maintains IP addresses. DHCP associates an IP address with a device in a lease that grants use of the IP address for a specified period of time.

**DHCP Option 14** The merit dump file option that contains signaling type, port number, call agent qualifier, and other information used in the provisioning of an xGCP port on a DOCSIS modem.

**DHCP Option 82** The relay agent option that can contain a circuit ID suboption used for device detection and a remote ID suboption used to send provisioning information back to a device.

**digital set top box** *See* DSTB.

**digital video broadcast** *See* DVB.

**DOCSIS** Data-Over-Cable Systems Interface Specifications. Defines technical specifications for equipment at both subscriber locations and cable operators' headends. Adoption of this specification will accelerate deployment of data-over-cable services and ensure interoperability of equipment throughout system operators' infrastructures.

**downstream traffic** Services travel as downstream traffic from the headend to devices (cable modems and DSTBs). The network may also support upstrean traffic from devices to the headend. Upstream traffic includes requests for special services such as pay-for-view events.

**DPE** Device provisioning engine. A device communicating with the DHCP and is the first point of BPR contact for a device to receive its configuration. The DPE caches device information to ensure BPR scalability and handles configuration requests including downloading configuration files to devices.

**DSTB** Digital set-top box. A device that enables a television to become a user interface to the Internet and to receive and decode digital television signals.

**Dynamic Host Configuration Protocol** *See* DHCP.

## E

**extension point** Network Registrar extension points are the interface between Network Registrar and CSRC BPR. They allow CSRC BPR to inject information such as client class into the Network Registrar flow.

**external file** An external file is a file outside of the CSRC BPR file system. External file types include templates.

## F

**FQDN** Fully qualified domain name. FQDN is the full name of a system, rather than just its hostname. For example, cisco is a hostname and www.cisco.com is an FQDN.

**fully qualified domain name** *See* FQDN.

## G

**Gateway Control Protocol** Gateway Control Protocols (xGCPs), such as the Simple Gateway Control Protocol (SGCP) for residential voice services.

## H

**high-speed data** *See* HSD.

**HSD** A high-speed data service, such as DOCSIS.

## I

**Internet** Largest global internetwork, connecting tens of thousands of networks worldwide and having a "culture" that focuses on research and standardization based on real-life use. Many leading-edge network technologies come from the Internet community. The Internet evolved in part from ARPANET.

**IOS images** These are images stored in firmware for a Cisco device. The Cisco device can upload the image to upgrade its functionality. BPR treats this file type like any other binary file.

**IP address** An IP address is a 32-bit number that identifies each sender or receiver of information that is sent in packets across the Internet.

## M

**MAC** Media access control. Lower of the two sublayers of the data link layer defined by the IEEE. The MAC sublayer handles access to shared media, such as whether token passing or contention will be used.

**MAC address** Standardized data link layer address that is required for every port or device that connects to a LAN. Other devices in the network use these addresses to locate specific ports in the network and to create and update routing tables and data structures. MAC addresses are 6 bytes long and are controlled by IEEE, Also known as hardware address, MAC-layer address, or physical address. Compare with *network address*.

**Management Information Base** *See* MIB.

**Media Access Control** *See* MAC.

**Media Terminal Adapter** *See* MTA.

**Message Integrity Check** *See* MIC.

**MIB** Management information base. An object defined by SNMP and used by DOCSIS cable modems for functions such as reset.

**MIC** Message integrity check. A message that helps validate transmissions between cable modems and the CMTS.

**modem** Modulator-demodulator. Device that converts digital and analog signals. At the source, a modem converts digital signals to a form suitable for transmission over analog communication facilities. At the destination, the analog signals are returned to their digital form. Modems allow data to be transmitted over voice-grade telephone lines.

**MSO** Multiple system operator. A company that operates more than one cable TV or broadband system.

**MTA** Media Terminal Adapter. Equipment at the customer end of a broadband network.

**multiple service operator** *See* MSO.

## N

**NAT** Network address translation. Mechanism for reducing the need for globally unique IP addresses. NAT allows an organization with addresses that are not globally unique to connect to the internet by translating those addresses into globally routeable address space. This is also known as Network Address Translator.

**network address** Network layer address referring to a logical, rather than physical, network device. Compare with *MAC address*.

**network address translation** *See* NAT.

**network administrator** Person responsible for operation, maintenance, and management of a network. *See also* network operator.

**network operator** Person who routinely monitors and controls a network, performing such tasks as reviewing and responding to alarms, monitoring throughput, configuring new circuits, and resolving problems. *See also* network administrator.

**Network Registrar** *See* NR.

**NR** Cisco Network Registrar. A software product that provides IP addresses, configuration parameters, and DNS names to DOCSIS cable modems and PCs, based on network and service policies.

## O

**open system interconnection** *See* OSI.

**operations support system** *See* OSS.

**OSI** Open system interconnection. Network architectural model developed by ISO and ITU-T. The model consists of seven layers, each of which specifies particular network functions such as addressing, flow control, error control, encapsulation, and reliable message transfer. The lowest layer (the physical layer) is closest to the media technology. The lower two layers are implemented in the hardware and software, while the upper five layers are implemented only in software. The highest layer (the application layer) is closest to the user. The OSI reference model is used universally as a method for teaching and understanding network functionality.

**OSS** Operations support system. Network management system supporting a specific management function, such as alarm surveillance and provisioning, in a carrier network. Many OSSs are large centralized systems running on mainframes or minicomputers.

## P

**packet internet groper** *See* ping.

**ping** Packet internet groper. ICMP echo message and its reply. Often used in IP networks to test the reachability of a network device.

**port** An IP terminology, and upper-layer process that receives information from lower layers. Ports are numbered, and each numbered port is associated with a specific process. For example, SMTP is associated with port 25. A port number is also called a well-known address.

**provisioning API** A series of functions that programs can use to make the operating system perform various functions.

**provisioning groups** Groupings of DPE and Network Registrar, based on either network topology or geography, to improve network performance.

**publishing** Copying provisioning information to an external datastore in real time. Publishing plug-ins must be developed to write data to a datastore.

## Q

**QoS** Quality of Service. Measure of performance for a transmission system that reflects its transmission quality and service availability.

**Quality of Service** *See* QoS.

## R

**RDU** Regional distribution unit. The RDU is the primary server in the BPR provisioning system. It manages generation of device configurations, forwards all API requests, and manages the BPR system.

**redundancy** In internetworking, the duplication of devices, services, or connections so that, in the event of a failure, the redundant devices, services, or connections can perform the work of those that failed.

**redundant logs** Log files become redundant once its data has been written into the database. *See also* active logs and removable logs.

**reload** The event of a Cisco router or piece of software rebooting, or the command that causes the router or software to reboot.

**removable logs** Log files become removable after either being backed up, or when the complete database that contains data for this log file has been backed up. *See also* active logs and redundant logs.

**response file** A file that contains the values for parameters required to install the BPR packages. The installer uses the values in the response file instead of prompting for the information.

## S

**scope** A range of IP addresses associated by means of selection tags with client classes, usually two client classes for unprovisioned cable modems and CPEs, and two for provisioned cable modems and CPEs.

**selection tags** Selection tags associated with Network Registrar scopes. An identifier for provisioning components, such as *provisioned cable modem* or *unprovisioned CPE*, used to help determine scopes.

**shared secret** A password known only to the remote DOCSIS device and the CSRC BPR administrator. This password is checked to authenticate messages between the device and the CSRC administrator. It works in conjunction with the CMTS MIC in order to prevent services from being stolen. The CMTS MIC option, when enabled, lets the CMTS validate the cable modem.

**Simple Network Management Protocol** *See* SNMP.

**SNMP** Simple Network Management Protocol is is the standard operations and maintenance protocol for the Internet and is a network management protocol used almost exclusively in TCP/IP networks. SNMP provides a means to monitor and control network devices, and to manage configurations, statistics collection, performance, and security.

**SNMP community strings** — The SNMP community strings are clear-text strings in SNMP packets that serve as simple passwords. The community strings help control which SNMP managers (such as servers that want to reset cable modems) have read-only, read-write, or no access to the SNMP agent in devices (for example, DOCSIS cable modems). This mechanism provides minimal protection to prevent a neighbor from resetting a subscriber's device.

- The SNMP Community Write String controls which SNMP managers have write access to the SNMP agent in devices (for example, DOCSIS cable modems).
- The SNMP Community Read String controls which SNMP managers have read-only, read-write, or no access to the SNMP agent in devices (for example, DOCSIS cable modems).

**static configuration files** — These files are used as a configuration file for a device. For example, a static configuration file called gold.cm would identify the gold DOCSIS class of service. BPR treats this file type like any other binary file.

## T

**Telnet** — Standard terminal emulation protocol in the TCP/IP protocol stack. Telnet is used for remote terminal connection, enabling users to log in to remote systems and use resources as if they were connected to a local system. Telnet is defined in RFC 854.

**template files** — Text files that contain DOCSIS options and values that, when used in conjunction with a DOCSIS class of service, provide dynamic DOCSIS file generation

**TFTP** — Trivial File Transfer Protocol. Simplified version of file transfer protocol (FTP) that allows files to be transferred from one computer to another over a network.

**TOD** — Time-of-Day (TOD) server. This server provides the time required by some technologies, such as DOCSIS.

**trivial file transfer protocol** — *See* TFTP.

## U

**uBr** — Universal Broadband Router (also known as the Cisco 7246 or 7223 router). The Cisco router implementation of a DOCSIS CMTS.

**URL** — Universal resource locator. Standardized addressing scheme for accessing hypertext documents and other services using a browser. *See also* browser.

## V

**Voice over IP** — *See* VoIP.

**VoIP** — Voice over IP. VoIP is the ability to make telephone calls and send faxes over IP-based data networks with a suitable quality of service (QoS) and superior cost/benefit.

## W

**WWW** World Wide Web. The large network of Internet servers providing hypertext and other services to terminals running client applications such as a browser. *See also* browser.

## X

**XGCP** A Gateway Control Protocol used to pass data between networks. This includes that M (for Media) GCP and S (for Simple) GCP.


## A



## B

## C

## D

## E

## F



## G

## H

## I



## J



## K

## L



## M

## N

## O



## P

## Q



## R



## S

## T



## U



## V



## W



## X