Cisco Broadband Access Center 3.8 Integration Developer's Guide

October 31, 2012

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

Text Part Number: OL-27177-01

Cisco Broadband Access Center 3.8 Integration Developer's Guide

CONTENTS

Preface

The *Cisco Broadband Access Center 3.8 Integration Developer's Guide* describes the Cisco Broadband Access Center (BAC) Application Programming Interface (API). You can use this API to integrate Cisco IP Solution Center (ISC) Multiprotocol Label Switching (MPLS) virtual private network (VPN) with your Business Support Systems (BSS) and Operational Support Systems (OSS).

This chapter provides an outline of the other chapters in this guide. It gives information about related documents that support this Cisco BAC release, and demonstrates the styles and conventions used in the guide.

This chapter contains the following sections:

Audience

System integrators, network administrators, and network technicians can use this integration guide to integrate the various BSS and OSS with Cisco BAC. Only experienced users should use these instructions. To use the instructions in this guide, you must be familiar with:

- Cisco BAC architecture.
- Java programming.

Organization

This guide includes the following sections:

Section	Title	Description
Chapter 1	Introduction	Describes the components that integrate with Cisco BAC.
Chapter 2	Cisco BAC Architecture	Describes the Cisco BAC architecture and the functions of each component.

Section	Title	Description
Chapter 3	Client and RDU Communication	Describes the communication between the Cisco BAC API and the Regional Distribution Unit (RDU).
Chapter 4	Batches and Commands	Describes the concepts and rules related to batches and how you can troubleshoot the errors that occur during integration.
Chapter 5	Events	Describes and explains how to register and handle Cisco BAC events.
Chapter 7	Getting Started with the BAC API	Describes how you can get started with the API.
Chapter 8	Use Cases	Describes some of the most common provisioning API use cases.

Conventions

This document uses the following conventions:

Convention	Indication
bold font	Commands and keywords and user-entered text appear in **bold** font.
italic font	Document titles, new or emphasized terms, and arguments for which you supply values are in *italic* font.
[ ]	Elements in square brackets are optional.
{x | y | z }	Required alternative keywords are grouped in braces and separated by vertical bars.
[x | y | z]	Optional alternative keywords are grouped in brackets and separated by vertical bars.
string	A non-quoted set of characters. Do not use quotation marks around the string or the string will include the quotation marks.
`courier` font	Terminal sessions and information the system displays appear in `courier` font.
< >	Nonprinting characters such as passwords are in angle brackets.
[ ]	Default responses to system prompts are in square brackets.
!, #	An exclamation point (!) or a pound sign (#) at the beginning of a line of code indicates a comment line.

Note Means *reader take note*.

Tip Means *the following information will help you solve a problem.*

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

Product Documentation

Note We sometimes update the printed and electronic documentation after original publication. Therefore, you should also review the documentation on http://www.cisco.com for any updates.

You can view the marketing and user documents for Cisco Broadband Access Center at:

http://www.cisco.com/en/US/products/sw/netmgtsw/ps529/tsd_products_support_series_home.html

The following document gives you the list of user documents for Cisco Broadband Access Center 3.8:
http://www.cisco.com/en/US/docs/net_mgmt/broadband_access_center/3.8/documentation/overview/Cisco_BAC38_DocOverview.html

Related Documentation

Note We sometimes update the printed and electronic documentation after original publication. Therefore, you should also review the documentation on Cisco.com for any updates.

The following document gives you the list of user documents for Cisco Prime Network Registrar 8.1:

http://www.cisco.com/en/US/docs/net_mgmt/prime/network_registrar/8.1/doc_overview/guide/CPNR_8_1_Doc_Guide.html

Note Cisco Network Registrar (CNR) is re-branded to Cisco Prime Network Registrar starting with the 8.0 release.

The following document gives you the list of user documents for Cisco Access Registrar 5.0:

http://www.cisco.com/en/US/docs/net_mgmt/access_registrar/5.0/roadmap/guide/PrintPDF/ardocgd.html

Obtaining Documentation and Submitting a Service Request

For information on obtaining documentation, submitting a service request, and gathering additional information, see the monthly *What's New in Cisco Product Documentation*, which also lists all new and revised Cisco technical documentation, at:

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

Subscribe to the *What's New in Cisco Product Documentation* as an RSS feed and set content to be delivered directly to your desktop using a reader application. The RSS feeds are a free service. Cisco currently supports RSS Version 2.0.

CHAPTER 1

Introduction

This chapter provides an overview of the Cisco Broadband Access Center (BAC) Application Programming Interface (API) and the API functions that you can use to perform the Regional Distribution Unit (RDU) tasks. This chapter contains the following sections:

Overview

Cisco BAC automates the tasks of provisioning and managing Customer Premises Equipment (CPE) in a broadband service-provider network.

It enables secure provisioning and management of CPE by using the Broadband Forum's CPE WAN Management Protocol (CWMP). This is a standard defined in the TR-069 specification. Cisco BAC integrates the capabilities defined in TR-069 to increase operator efficiency and reduce network-management problems.

You can integrate Cisco BAC into new or existing environments, using an API that lets you control how Cisco BAC operates. Using the API, you can integrate various Business Support Systems (BSS) and Operational Support Systems (OSS) with Cisco BAC. The API is a programmatic interface through which the various BSS and OSS clients connect to the RDU. The RDU is the central server in a Cisco BAC deployment.

You can use the Cisco BAC API to:

- Register devices in the RDU database.
- Assign configuration policies for devices.
- Execute CWMP operations on the CPE.
- Configure the Cisco BAC provisioning system.

Use this guide along with the following resources that are integrated with the Cisco BAC software:

- API Javadocs, located at <*BPR_HOME*>/docs/nb-api/javadoc.
- Sample API client code, located at <*BPR_HOME*>/rdu/samples/nb-api.

<*BPR_HOME*> refers to the home directory in which you install Cisco BAC. The default home directory is /opt/CSCObac.

API Functions

Using the Cisco BAC API, you can perform the following:

- Provisioning operations.

  You can:

  - Add, modify, and search device records in the RDU database.
  - Associate device records with Classes of Service in the RDU database.
  - Associate device records with the groups in the RDU database.
  - Retrieve discovered device data stored in the RDU database.
  - Retrieve device operation history from the RDU database.
  - Retrieve device faults from the BAC servers.

- Device management operations.

  You can:

  - Retrieve live data, such as statistics, from a device.
  - Execute diagnostics on a device.
  - Reboot the device.
  - Reset the device settings to default configuration.
  - Perform individual sets on a device.
  - Execute any TR-069 remote procedure calls (RPC) in pass-through mode.

- System configuration and management operations.

  You can:

  - Configure Class of Service objects in the RDU.
  - Manage firmware rules, configuration templates, and other files.
  - Configure device grouping objects in the RDU.
  - Configure licenses.
  - Configure users.
  - Configure system settings for BAC.
  - Retrieve BAC server status and statistics.

You can perform all system configuration and management operations from the Cisco BAC administrator user interface as well. For details on how to perform these operations, see the *Cisco Broadband Access 3.8 Center Administrator Guide*.

For more details on how to perform provisioning and device management operations, see Use Cases.

CHAPTER 2

Cisco BAC Architecture

This chapter describes the basic Cisco BAC architecture and contains the following sections:

For more information on each of the components, see the *Cisco Broadband Access Center 3.8 Administrator Guide*.

Regional Distribution Unit

The Regional Distribution Unit (RDU) is the primary server in the Cisco BAC provisioning system. It is installed on a server running Solaris 10 or Linux 5 operating system.

The functions of the RDU include:

- Managing preprovisioned and discovered data from devices.
- Generating instructions for DPEs and distributing them to DPE servers for caching.
- Cooperating with DPEs to keep them current.
- Processing API requests for all Cisco BAC functions.
- Managing the Cisco BAC system.

The RDU enables you to add new technologies and services through an extensible architecture.

Device Provisioning Engine

The Device Provisioning Engine (DPE) communicates with the CPE on behalf of the RDU to perform any provisioning or management functions.

The RDU generates instructions that the DPE must carry out on the device. These instructions are distributed to the relevant DPE servers, on which they are cached. These instructions are then used during interactions with the CPE to accomplish tasks, such as device configuration, firmware upgrade, and data retrieval.

The DPE manages the following tasks:

- Synchronization with the RDU to retrieve the latest set of instructions for caching.
- Communication with the CPE, using CWMP and HTTP for file download services.
- Authentication and encryption of communication with the CPE.

Provisioning Group

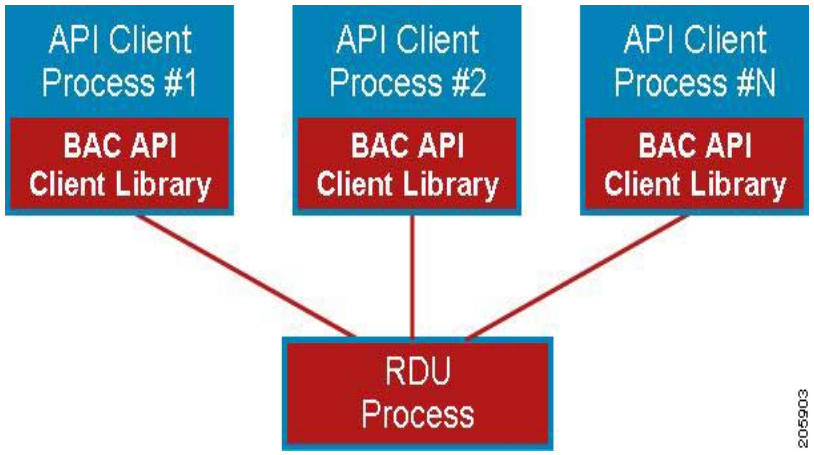
A provisioning group is a logical (typically geographic) grouping of servers that usually consist of one or more DPEs. Each DPE in a given provisioning group, caches identical sets of instructions from the RDU, thus enabling redundancy and load balancing.

Client API

The client API provides total client control over Cisco BAC capabilities. The API enables the client on a remote host to communicate with the RDU.

The API client library exposes the client to a single logical interface. For information on the objects and functions of this interface, see the API Javadocs in the Cisco BAC installation directory. Figure 2-1 shows three remote clients accessing the RDU through the API client library.

Figure 2-1 Embedded Client Library



The API client library is packaged in the bpr.jar and bacbase.jar files, located at *<BPR_HOME>*/lib, where *<BPR_HOME>* refers to the home directory on which you install Cisco BAC.

For client communication with the RDU to be successful, ensure that both .jar files are available in the Java classpath and compile against these libraries using the standard Java compilation process.

We recommend that you use Java version 1.6.0_05 or later to support the client API in Cisco BAC.

CHAPTER 3

Client and RDU Communication

This chapter details the communication between the client library and the RDU and describes on how to establish, maintain, and close the connection between the client library and the RDU.

This chapter contains the following sections:

Overview

The Cisco BAC API communicates with the RDU in a Cisco BAC deployment over TCP/IP.

The API client library initiates the connection between the API and the RDU. The RDU does not try to establish a connection between itself and the API.

Though the client library initiates and establishes connectivity between the API and the RDU, information flows in both directions, with the client library submitting requests to the RDU, and the RDU responding to those requests. The bilateral heartbeat messages enable the API client and the RDU to maintain a bidirectional connection.

The network administrator must ensure that:

- IP connectivity exists between the client and the RDU.
- The TCP port that the RDU listens on is opened through a firewall between the client API and the RDU. The default TCP port that the RDU listens on is 49187. The RDU uses this TCP port to bind itself to all network interfaces.

Establishing a Connection

The client establishes a connection with the RDU by passing the following parameters:

- Hostname of the RDU; for example, rdu.mso.com
- Port for communication with the RDU; the default port is 49187.
- Administrator username; the default administrator username is **bacadmin**.
- Administrator password; the default administrator password is **changeme**.

You can use the following code to establish a connection between the RDU and the client library:

```
final PACEConnection connection =
   PACEConnectionFactory.newInstance(
         "rdu.mso.com", 49187, "bacadmin", "changeme");
```

The connection between the client library and RDU is maintained until it is explicitly closed. See Closing a Connection, page 3-2 on how to close a connection.

Maintaining a Connection

The client library automatically maintains the connection between the client and RDU. In case the connection breaks in the network layer because of congestion, routing problems, or other issues, the client library automatically reconnects to the RDU. The client library tries to reconnect to the RDU until the connectivity is restored.

The reconnection process is automatic and does not impact your code while the RDU interacts with the library. For example, a synchronous call to submit a batch blocks the thread and returns the results when the results are available as usual; even if the client library had to automatically reconnect to the RDU.

Connection Concurrency

The client library maintains a single TCP connection to the RDU. This connection can be used for any number of requests and responses. Multiple threads can use the same single connection object.

While there is only a single underlying TCP connection, many Provisioning API Command Engine (PACE) connection instances can be created. If there is a need for multiple BAC users in a single client, then multiple PACE connections are required.

Closing a Connection

The connection between the RDU and the client library is maintained until you explicitly close the connection. You can use the following code to close the connection:

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

CHAPTER 4

Batches and Commands

This chapter provides an overview of batches and the commands contained in the batch. This chapter contains the following sections:

Overview

A batch object:

- Is a container for commands that the RDU must execute.
- Contains methods that control how the RDU executes the commands and returns results.

A command represents an operation that is performed on an object in the RDU database. For example, to add a new device, the client issues an Add command using the API to the RDU. To delete a device, the client issues the delete command to the RDU using the API.

The batch lifecycle (create, post, execute, return results) demands two entities to communicate over a network. For this communication, a provisioning client in Cisco BAC submits API requests to the RDU in the form of batches that contain single or multiple commands.

Figure 4-1 illustrates the concept of batch processing.

Figure 4-1 API Batch Object

Batches are atomic units; either all commands in the batch succeed or none of them succeed. If the batch fails, the RDU restores changes that were made to its database. The RDU executes the commands in the same sequence in which they are added to the batch.

For more information on batch identification, see Identifying a Batch, page 4-3. For more information on batch flags, see Batch Processing Flags, page 4-4.

Batch Rules

To execute a batch successfully, ensure that you follow rules listed below:

- A batch must contain between 1 and 100 commands. You cannot execute a batch with no commands, or one with more than 100 commands.
- Commands in a batch must either be **read** or **write**. You cannot combine **read** and **write** commands in a batch. For example, the same batch cannot contain a get device details command (**read**) as well as an add device command (**write**).

  Note Commands that perform device operations (such as a connection request) are Write commands.

- Batch commands must relate to device or system configuration. You cannot combine device-related and system-related commands in a batch. For example, you cannot combine a modify Class of Service command (**system**) and an add device command (**device**) in the same batch.
- When a batch includes a command that interacts with a device through a device operation or an automatic activation flag, all commands in the batch must relate to the same device. For example, you cannot submit a batch containing two connection request operations for two different devices.

- On-connect commands and immediate device operation commands must not be submitted in the same batch.

Identifying a Batch

Every batch that the RDU executes, has a unique batch identifier. The batch identifier that the client library generates includes the hostname of the local client server and a random number that increments.

The batch identifier helps you to:

- Retrieve batch status from the RDU.
- Correlates the respective batch events in the RDU.

While the client library automatically generates a batch identifier, you can specify your own batch identifier, based on your requirements.

Note We recommend that you use the batch identifiers that the client library generates for you.

If you generate your own batch identifier, ensure that you clearly identify the local client server.

Tip If you have a global transaction identifier, it can be a good idea to include it in the batch identifier in order to monitor the transaction throughout the entire system.

If the RDU detects a duplicate batch identifier, it rejects that batch. Submitting batches with batch identifiers that have already been processed, may lead to failure and unexpected results.

You can generate a batch identifier in one of the following ways:

- Using the client library —To use the client library, use the newBatch methods on the Provisioning API Command Engine (PACE) connection object for a batch without the batch identifier parameter.

  Use the following code to generate a batch identifier using a client library:

```
public Batch newBatch()

public Batch newBatch(ActivationMode activation)

public Batch newBatch(PublishingMode publishing)

public Batch newBatch(ActivationMode activation, ConfirmationMode confirmation)

public Batch newBatch(ActivationMode activation, ConfirmationMode confirmation,
                              PublishingMode publishing)

public Batch newBatch(ActivationMode activation,
                              PublishingMode publishing)
```

- By specifying your own identifier — To generate your own batch identifier, use the newBatch methods on the PACE connection object containing the batch identifier parameter.

  Use the following code to generate a batch identifier by specifying your own identifier:

```
public Batch newBatch(String batchId)

public Batch newBatch(String batchId,
                              ActivationMode activation)
```

```
public Batch newBatch(String batchId,
                              PublishingMode publishing)

public Batch newBatch(String batchId,
                              ActivationMode activation,
                              ConfirmationMode confirmation)

public Batch newBatch(String batchId,
                              ActivationMode activation,
                              ConfirmationMode confirmation,
                              PublishingMode publishing)

public Batch newBatch(String batchId,
                              ActivationMode activation,
                              PublishingMode publishing)
```

Batch Processing Flags

Batch processing flags control:

- Batch interaction with a device.
- Notifications of batches to external systems. These notifications detail the changes that are made by various operations in a batch.

Cisco BAC supports the following processing flags, each of which is described in subsequent sections:

- Reliable, see Setting the Reliable Flag, page 4-4.
- Activation, see Setting the Activation Flag, page 4-5.
- Confirmation, see Setting the Confirmation Flag, page 4-6.
- Publishing, see Setting the Publishing Flag, page 4-7.
- Optimistic Locking, see Setting the Optimistic Locking Flag, page 4-8.

Setting the Reliable Flag

Communication between the client and the RDU breaks if:

- The client restarts after posting a batch.
- The RDU restarts after receiving a batch.
- The network connection breaks when the results are being sent. Subsequently, the results are lost.

To handle such issues, Cisco BAC provides a reliable batch flag. When you enable the reliable flag for a batch, the RDU stores the batch on receiving it, and even if the RDU restarts, the batch is guaranteed to be executed after the restart.

Note You can enable the reliable batch flag for batches that contain **write** commands, such as **add, change,** or **delete.**

After the batch is executed, the RDU stores the results in its database. Subsequently, the client can obtain results for the batches even after an RDU restart. To obtain the results, the client uses a join operation and the thread blocks till the results are returned or a timeout occurs.

If the RDU did not receive the batch, or cleared the results from its database, an error appears. At a time, the RDU stores the results of 2000 reliable batches that were last executed.

Note We recommend that you store all batch identifiers of reliable batches to the disk, before you post a batch. By storing the batch identifiers, the client library can query for results even if a client restart occurs.

- To join a reliable batch with a batch identifier using the PACE Connection object:
  - With a timeout:

```
final BatchStatus batchStatus = connection.join(batchId, 5000);
```

Note We recommend that you use a timeout value when using the join feature for reliable batches. Also, because reliable batches add a significant load to the RDU, use it only when client and network reliability outweigh the performance impact.

  - Without a timeout:

```
final BatchStatus batchStatus = connection.join(batchId);
```

- To force a batch to be reliable before submitting a synchronous or asynchronous post, use the following code:

```
// make it reliable
batch.forceBatchReliable();
```

For information on synchronous and asynchronous batches, see Batch Processing Modes, page 4-10.

Setting the Activation Flag

You can use the activation flag in batches that contain **write** commands and operate on a single device. The activation flag is of two types:

- No Activation—Executes by updating the RDU database and the appropriate DPE caches.

  Batches that include commands for on-connect device operations must use the no-activation flag.

- Automatic Activation—Executes by persisting the changes in the RDU database and by trying to establish contact with the device to obtain the latest configuration. For CWMP devices, this contact involves sending a connection request to the device and obtaining a connection.

  Batches that include commands for all immediate device operations, must use the automatic-activation flag.

You can mark a batch using the no-activation flag or the automatic-activation flag.

For example, consider a batch that contains a change Class of Service command for a device. If you execute the batch with the no-activation flag, the device's Class of Service is changed and the resulting new configuration is sent to the DPEs in the provisioning group. The new data is available in the appropriate DPEs for the next device session.

On the other hand, if you execute the same batch with an automatic-activation flag, the RDU not only sends the new configuration to the provisioning group, but also issues a connection request to the device to start a new session.

Activation only involves making a connection request. It does not verify if the configuration was successfully applied on the device. When you execute a batch with the automatic-activation flag, the batch becomes reliable.

Also, as activation involves both updating the RDU database and generating a connection request to the DPE, the batch may then return a warning, indicating that the database was successfully updated but that the connection request did not occur, if the device is offline.

For details on controlling this behavior using the Confirmation flag, see Setting the Confirmation Flag, page 4-6.

You can augment or replace the activation logic in the RDU during deployment using an extension. For more information, see the *Cisco Broadband Access Center 3.8 Administrator Guide*.

- You can create a batch with no activation in one of two following ways:
  - Without specifying the flag. Because no-activation is the default, batches are created with the no-activation flag.

    ```
    final Batch batch = connection.newBatch();
    ```

  - By explicitly setting the flag.

    ```
    final Batch batch = connection.newBatch(
                            ActivationMode.NO_ACTIVATION);
    ```

- You can create a batch with automatic activation using the following code:

  ```
  final Batch batch = connection.newBatch(
                          ActivationMode.AUTOMATIC);
  ```

Setting the Confirmation Flag

You can use the confirmation flag to control the behavior of batch activation. You must use the confirmation flag only in batches that have the automatic-activation flag set.

The confirmation flag communicates with the RDU on how the processing of a batch should proceed if there are warnings or errors during activation. For more information on warnings or errors during activation, see Batch Warnings, page 4-16.

Cisco BAC supports two types of confirmation flags:

- No confirmation
- Custom confirmation.

Unless you specify otherwise, a batch is created with the no-confirmation flag.

When you execute a batch with the no-confirmation flag, warnings or errors during activation do not cause the batch to fail. Instead, the batch results contain a warning indicating that activation issues occurred. The batch proceeds and database updates are committed.

When you execute a batch with the custom-confirmation flag and a warning occurs during activation, the batch results contain the warning. The batch proceeds, committing the database updates. However, if an error occurs during activation, and the batch results contain the error, the batch fails, and the database updates get rolled back.

Note You can replace or augment the activation code in the RDU so that the errors or warnings that appear depend on the code in use. In case a connection request to a device fails, the default CWMP code does not produce any warnings. However, it returns errors.

You can create a batch with a no-confirmation flag or a custom-confirmation flag.

- You can create a batch with the no-confirmation flag, using the following code:

```
final Batch batch = connection.newBatch(
                                ActivationMode.AUTOMATIC);
```

- You can create a batch with the custom-confirmation flag, using the following code:

```
final Batch batch = connection.newBatch(
                            ActivationMode.AUTOMATIC,
                            ConfirmationMode.CUSTOM_CONFIRMATION);
```

Setting the Publishing Flag

You can use publishing plug-ins to include custom code that help to notify the external entities of changes the batch make to the RDU database. For information on creating publishing plug-ins in the RDU, see the *Cisco Broadband Access Center 3.8 Administrator Guide*.

You can set the publishing flag in one of three ways:

- No publishing—The publishing plug-in is not called within the batch.
- Publishing with no confirmation—The publishing plug-in is executed. If an error occurs, the batch proceeds and any database change is updated.
- Publishing with confirmation—The publishing plug-in is executed. If an errors occurs, the batch fails and the database updates are rolled back.

Note When you mark a batch with the publishing with confirmation flag, the batch automatically becomes reliable.

You must explicitly specify if a batch is to be created with publishing; otherwise, batches are created using the no-publishing flag.

- You can create a batch with the no-publishing flag in one of the following ways:
  - Without setting any flag. Because the no-publishing flag is the default setting, a batch is thus created:

```
final Batch batch = connection.newBatch();
```

  - By explicitly setting the no-publishing flag:

```
final Batch batch = connection.newBatch(
                            PublishingMode.NO_PUBLISHING);
```

- You can create a batch with the publishing no-confirmation flag using:

```
final Batch batch = connection.newBatch(
                          PublishingMode.PUBLISHING_NO_CONFIRMATION);
```

- You can create a batch with the publishing-with-confirmation flag using:

```
final Batch batch = connection.newBatch(
                          PublishingMode.PUBLISHING_CONFIRMATION);
```

Setting the Optimistic Locking Flag

Because the API client executes in a client-server model, a time interval occurs between a get and a modify cycle. You can use the optimistic locking flag to prevent inconsistent changes being made to devices by different clients, simultaneously.

When you perform a **get** operation for an object (such as a device), the details map contains the GenericObjectKeys.OID_REVISION_NUMBER key. The value for this key is an object identifier that is encoded with the current revision number for the object.

You can add this revision number to the batch to ensure that the object is not changed before the changes in your batch are applied. If the object has changed, as indicated by a different revision number, the batch returns the following error: `BatchStatusCodes.BATCH_NOT_CONSISTENT`.

For example, consider a batch that retrieves a device and change its Class of Service using optimistic locking:

Note This example uses the CWMP device identifier 00000C-1234567890.

```
final DeviceID deviceId = new CWMPDeviceID("00000C-1234567890");

final Batch batchForGet = connection.newBatch();
batchForGet.getDetails(deviceId, null);

final BatchStatus batchStatusForGet = batchForGet.post(10000);

if (batchStatusForGet.isError())
{
    // handle error
}

// we know that we only submitted one command in the
// batch so we can get the first command status

final CommandStatus commandStatus =
             batchStatusForGet.getCommandStatus(0);

// we know we submitted a get details command so we are
// expecting a result of a map
if (commandStatus.getDataTypeCode != CommandStatus.DATA_MAP)
{
      // throw an exception or log a message
      // we are expecting a map and didn't get one
}

final Map<String, Object> result =
    (Map<String, Object>)commandStatus.getData();

final Object consistencyValue = result.get(
            GenericObjectKeys.OID_REVISION_NUMBER);

// change the class of service
final Batch batchForMod = connection.newBatch();
batchForMod.changeClassOfService(deviceId, "gold");
```

```
// now do the optimistic locking
final List<Object> list = new ArrayList<Object>();
list.add(consistencyValue);
batchForMod.ensureConsistency(list);

// now when we post we know the device has not been changed
// since our get and our change
// if it has it will be an error
```

Submitting the Batch

The API client submits batches to the RDU synchronously or asynchronously. The API submits batches to the RDU in two modes:

- Submitting in Synchronous Mode, page 4-9
- Submitting in Asynchronous Mode, page 4-9

Submitting in Synchronous Mode

When the API client submits a synchronous batch, the batch blocks the current thread till:

- The RDU returns the results on the batch.
- The batch times out before the RDU returns results.

If the client library does not receive a response from the RDU within the specified timeout, a `ProvTimeoutException` is thrown. The error message in the exception indicates that the client library did not receive the batch result in the specified time but that the batch execution did not necessarily fail.

You can submit your batch to the RDU in synchronous mode with or without a timeout.

- You can submit a synchronous batch on a PACE connection object with a timeout, using:

  ```
  // posting with timeout (in milliseconds)
  final BatchStatus batchStatus = connection.postBatch(batch, 5000);
  ```

  We recommend that you post a batch in synchronous mode with a timeout configured. For batches that read or update the database, you can configure a timeout of 30,000 milliseconds (msec). For batches that perform operations on live devices, you can configure a timeout of 60,000 msec.

- You can submit a synchronous batch on a PACE connection object without a timeout, using:

  ```
  // posting with no timeout
  final BatchStatus batchStatus = connection.postBatch(batch);
  ```

Submitting in Asynchronous Mode

When the client submits an asynchronous batch, the client library thread that posts a batch to the RDU becomes active again. The client library obtains the results using the batch events or, if preferred, does not obtain results at all. For information on asynchronous mode, see *Cisco Broadband Access Center 3.8 Administrator Guide*.

You can submit an asynchronous batch on a PACE connection object, using:

```
// posting async
connection.postBatchNoStatus(batch);
```

To obtain batch results via batch events, the client library registers a listener class that implements batch listener using the PACE connection with an appropriate qualifier.

The batch listener interface exposes a completed method that has a batch event as its argument, and this method is called for each qualified batch when it completes.

The batch event, in turn, provides access to the batch status object, which contains the results of the batch. To correlate between the submitted batch and the results, use the batch identifier.

To receive the results, ensure that the listener is registered before the batch is submitted. See Events to view the various events posted by Cisco BAC.

Batch Processing Modes

Depending on the commands contained in the batch, the RDU executes the batch in one of two following modes:

- Concurrent
- Nonconcurrent

The concurrent and nonconcurrent modes provide higher throughput at the RDU, without losing data integrity.

When the RDU receives a batch, the commands in the batch determine the mode in which a batch is executed. The RDU executes most batches in concurrent mode.

A batch must include either concurrent or nonconcurrent commands. The RDU does not process a mix of concurrent and nonconcurrent commands in a single batch. When running one concurrent batch, you can execute other concurrent batches as well.

If the RDU has to process a batch in nonconcurrent mode, all the batches currently being run in the RDU must have completed execution, and no new batches must have started. Batches you submit at this time are queued.

The RDU executes the new batches in the mode in which they are marked, after completing the processing of the nonconcurrent batch; by so doing, the RDU avoids lock conflicts and consistency issues.

Only a few commands cause a batch to run in nonconcurrent mode. These commands relate to the following system configuration operations:

- Configuring Class of Service objects in the RDU.
- Managing firmware rules, configuration templates and other files.
- Configuring device grouping objects in the RDU.
- Configuring licenses.
- Configuring users.
- Configuring system settings.

Batch Results

A batch result is the outcome of a batch that the RDU executes. Results are returned either as exceptions or as batch status objects.

When posting a batch, an exception is thrown if:

- The batch has already been posted.
- A connection to the RDU cannot be established.
- A timeout occurred when submitting a batch in synchronous mode.

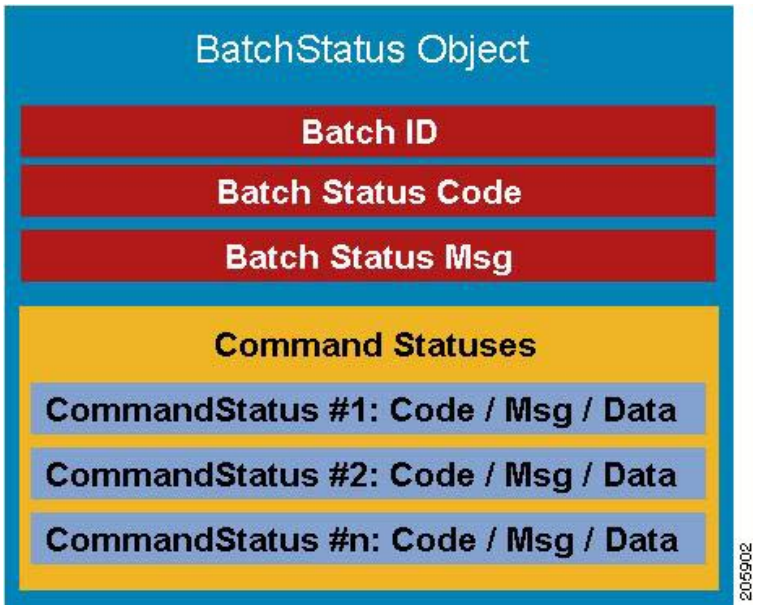
Note These exceptions are rare and are raised as a `ProvisioningException` object.

If there is no `ProvisioningException` thrown, a batch status object is returned. Similar to batches and commands, there are batch status objects and command status objects.

A batch status object contains command status entries for each of the commands in the corresponding batch object that was executed. The order of the command status entries matches that of the commands in the batch object.

Figure 4-2 illustrates the structure of a batch status object.

Figure 4-2 Batch Status Object



The batch status object, like a batch, serves as a container. If a single command fails, you can query the batch status to determine if there was a failure and to obtain the command status that contains the details. You can also check the batch status to determine if all the commands succeeded.

Note A batch status object does not always contain a command status. An invalid batch construction, for example, one with a combination of **read** and **write** commands, returns a batch status object without command status objects.

- You can query the batch status object to determine:
  - Whether a single command in a batch failed.
  - The success of all commands in the batch.

- You can query the command status object to determine the details of a command failure. For more information on the status objects, see Batch and Command Errors, page 4-16.

To check whether the batch successfully passes, and to handle errors, if any, use the following code:

```
final BatchStatus batchStatus = connection.post(batch); if (!batchStatus.isError())
{
    // batch passed so all commands passed
}
else
{
    // we need to determine if it was a batch error or a
    // command error that caused this failure

    if (batchStatus.getFailedCommandIndex() == -1)
    {
        // this is a batch only error
        // get the error code and get the error message
        final StringBuilder msg = new StringBuilder(128);
        msg.append("Batch with ID [");
        msg.append(batchStatus.getBatchID());
        msg.append("] failed with error code [");
        msg.append(batchStatus.getStatusCode());
        msg.append("]. [");
        msg.append(batchStatus.getErrorMessage());
        msg.append("].");

        // throw an exception or log the message
    }
    else
    {
        // this is a batch error caused by a command
        final CommandStatus commandStatus =
                batchStatus.getFailedCommandIndex();

        // get the error code and get the error message
        final StringBuilder msg = new StringBuilder(128);
        msg.append("Batch with ID [");
        msg.append(batchStatus.getBatchID());
        msg.append("] failed with command error code [");
        msg.append(commandStatus.getStatusCode());
        msg.append("]. [");
        msg.append(commandStatus.getErrorMessage());
        msg.append("].");

        // throw an exception or log the message
    }
}
```

If a batch successfully passed and you want to view the results before retrieving the details of a device, use the following code.

```
final BatchStatus batchStatus = connection.post(batch); if (batchStatus.isError())
{
    // handle error
}
else
{
    // we know that we only submitted one command in the
    // batch so we can get the first command status

    final CommandStatus commandStatus =
                batchStatus.getCommandStatus(0);
```

```
        // we know we submitted a get details command so we are
        // expecting a result of a map
        if (commandStatus.getDataTypeCode !=
            CommandStatus.DATA_MAP)
        {
            // throw an exception or log a message
            // we are expecting a map and didn't get one
        }
        else
        {
            final Map<String, Object> result =
                    (Map<String, Object>)commandStatus.getData();
            // now handle the result
        }
    }
```

Queuing a Batch

When the RDU receives a batch from a client, it queues the batch for execution. The priority of a batch determines the queue that the RDU uses for a successful execution of the batch. In case the selected queue is full, the batch is dropped, and the client notified.

There are seven batch queues, each with the capacity to hold 1000 batches in the order that they were received. Each queue has a different priority. Each queue could contain batches that originate internally or externally.

Internal batches are those designated from the DPE and the RDU, and the batches submitted to the client library. External batches are those designated from the API client.

Of the seven batch queues:

- Two queues are meant for RDU API client batches (for example, those relating to the administrator user interface and the OSS).
- Five queues are meant for internal batches that relate to:
  - Configuration generation of Cisco Network Registrar DHCP extensions.
  - BAC server registration.
  - DPE cache synchronization.
  - DPE configuration regeneration.
  - Legacy IP updates.

The RDU has 100 threads dedicated to execute batches. At a time, the server can execute a maximum number of threads as defined in Table 4-1.

PACE also processes batches from the Instruction Generation Service (IGS) and a maximum of one IGS batch is executed for every five batches from the RDU batch queues.

Table 4-1 lists the various batch queues, with the maximum executing threads for each queue.

Table 4-1 Batch Queue

Queue	Batch Origin	Maximum Executing Threads
No Activation	External	50
Automatic Activation		25
Configuration Generation	Internal	25
Configuration Regeneration		25
DPE Synchronization		1
Server Registration		1
IP Update		10

Retrying a Batch

If you are unable to receive results, you will have to retry the batch posting. You will not receive results if:

- A timeout occurred.
- Issues exist in batch submission.
- The client that posted the batch restarts.

Though the client library allows you to submit batches only once, you can create a copy of the original batch and re-post it.

There are four basic groups of commands for retrying a batch. Commands that:

- Add new objects to the RDU, such as add a device or a Class of Service.
- Delete objects from the RDU, such as delete a device or a Class of Service.
- Manipulate existing objects in the RDU, such as change the Class of Service for a device, get device details, or get details on a Class of Service.
- Communicate with a live device without manipulating any objects in the RDU, such as CWMP device operations including add object, delete object, and set parameter values.

Note While batches support running commands across groups, mixing commands from different groups adversely impacts batch retrying.

Table 4-2 describes the four different command groups for retrying a batch.

Table 4-2 Command Groups for Retrying a Batch

Command Group	Description
Add new objects to the RDU	For batches that contain commands to add new objects to the RDU, retrying causes issues if the original batch succeeds. You will get a command error code that the object already exists. For example, if you try to add objects that already exist, the following batch and command status codes are returned: • Batch status code: `BatchStatusCodes.BATCH_FAILED_WRITE` • Command status code: `CommandStatusCodes.CMD_ERROR_DEVICEID_EXISTS` Any other errors that you receive indicates a validate error that is not related to retrying the original batch.
Delete objects in the RDU	For batches that contain commands to delete objects existing in the RDU, retrying is acceptable even if the original batch succeeds. You will get a command error code that the object is unknown. For example, if you try to delete an object that has already been deleted, the following batch and command status codes are returned: • Batch status code: `BatchStatusCodes.BATCH_FAILED_WRITE` • Command status code: `CommandStatusCodes.CMD_ERROR_DEVICEID_UNKNOWN` Any other errors that you receive indicate a validate error that is not related to retrying the original batch.
Manipulate objects in the RDU	For batches that contain commands that manipulate objects existing in the RDU, retrying does not make any difference. Any errors that you receive indicate a validate error that is not related to retrying the original batch.
Communicate with live devices	For batches that contain commands that perform operations on live devices, retrying depends on the operation. For example, if an operation adds a new object to the device, deletes an object from the device, or modifies an object from the device, retrying may cause a problem, similar to what an add device command does with the RDU.
When retrying a batch for which you created your own batch identifier, ensure that you use the identifier of the original batch. In case you receive a `Duplicate BatchID` error, wait until the original batch has finished execution (for example, using the batch join feature), then submit the batch, if required.	

Handling Errors

Troubleshooting integration issues involve handling errors and warnings. Integration errors may occur because of a:

- Failed client library connection to the RDU.
- Failed batch posted in the RDU.

When the connection between the client library and the RDU fails, the client library tries to reconnect to the RDU. When a batch fails, all database changes are rolled back; a batch status object is returned, indicating that an error occurred.

Batch warnings indicate that the batch succeeded and the changes were committed to the database.

Types of Errors

The two types of errors that occur while integrating the OSS and BSS components to Cisco Broadband Access Center are:

- Connection Errors, page 4-16.
- Batch and Command Errors, page 4-16.

Connection Errors

Connection errors are those that occur when the API client library tries to restore a broken connection with the RDU. In general, you can ignore connection errors because the client library tries to reconnect to the RDU until the connection is restored. After a connection is restored, processing continues as usual.

You must, however, explicitly address authentication connection errors, such as an `RDUAuthenticationException`. BAC does not automatically recover from an authentication error. As an administrator, you must confirm the authentication credentials of the user (username and password).

Batch and Command Errors

To check batch and command errors, see Step 5 in Creating an API Client, page 6-3.

The status objects, BatchStatus and CommandStatus, have methods to return the error code along with a detailed error message. See the API constants *BatchStatusCodes.java* and *CommandStatusCodes.java* in the API Javadocs in the installation directory of the product for the methods that return the error code along with the detailed error message.

Batch Warnings

A warning indicates that:

- The batch has succeeded and the changes have been committed.
- Something of interest has occurred.

The RDU may return warnings for successful batches in these instances:

- When the batch has altered high-level RDU objects, such as a Class of Service or a group. The devices related to these objects must have instructions regenerated (via the Instruction Generation Service).

  The warning indicates the need for instruction regeneration and that this activity will occur. The RDU automatically regenerates instructions for these devices.

- During the activation stage of a batch marked with the default no-confirmation batch flag, if an error (such as a connection request failure because the device is offline) occurs, the error appears as a warning, and the batch succeeds.
- When you execute a batch with the custom-confirmation flag and a warning occurs during activation, the batch results contain the warning. The batch proceeds, committing the database updates.

  However, if an error occurs during activation, and the batch results contain the error, the batch fails, and the database updates get rolled back.

CHAPTER 5

Events

This chapter provides an overview of the events that the RDU and DPEs provide and explains how to register and handle these events. This chapter contains the following sections:

Overview

Using the Cisco BAC client library, you can register for numerous types of events that are sourced from the RDU and the DPEs. The events that are sourced include:

- Device notification.
- TR-069 session events.
- Asynchronous operation notification.
- Batch status events.
- Custom extension events.
- Policy related events.

Event Registration

Events are registered by implementing the appropriate event listener interface. The resulting class is then registered through the PACE connection along with a qualifier.

The qualifier further filters the events that the client receives. If the client wants to receive all events, you can use the QualifyAll qualifier. If an object can be modified in the RDU, a corresponding event will be available in the API. For a complete list of available events, see the *package.com.cisco.provisioning.cpe.events* section in the API Javadocs.

Each event class has a specific qualifier with methods that allow you to refine the events that are to be delivered to the registered listener.

Note You can use only the qualifiers that the client library provides. Cisco BAC does not support implementing your own qualifiers.

For example, to handle all asynchronous operation events that are fired when an on-connect device operation completes:

Step 1 Create the listener class using:

```
public class AsyncEventHandler implements AsyncOperationListener
{
    private boolean m_isOneShot;

    /**
     * The method invoked when a {@link AsyncOperationEvent
     * AsyncOperationEvent} arrives as a result of an async
     * operation completing.
     *
     * <P>
     * @param ev The object containing the {@link AsyncOperationEvent
     * AsyncOperationEvent} data.
     */
    public void completed(final AsyncOperationEvent ev)
    {
        // handle the incoming event
    }

    /**
     * Gets oneShot mode value, specifying whether or not the listener
     * is registered for just one occurrence of the Event.
     *
     * <P>
     * @return <TT>true</TT> if oneShot mode has been set.
     */
    public boolean getOneShot()
    {
        return m_isOneShot;
    }

    /**
     * Sets oneShot mode, specifying that the registration request is
     * for just one occurrence of the Event.
     *
     * <P>
     * @param flg <TT>true</TT> if oneShot mode is being set.
     */
    public void setOneShot(final boolean flg)
    {
        m_isOneShot = flg;
    }
}
```

Step 2 Register the created listener class using the PACE connection:

```
final AsyncEventHandler handler = new AsyncEventHandler();
// use a qualifier that filters all events
final Qualifier qualifier = new QualifyAll();

// register the listener, this will contact the RDU
// and from now on we will start receiving events
connection.addAsyncOperationLister(handler, qualifier);
```

Note If the connection breaks after the listener is registered, you do not have to reregister the listener. The client library automatically registers the listener again.

Step 3 Receive the events.

The listener class will be executed when the event arrives.

Step 4 Remove the listener that is created.

You can use any of the following methods to remove a listener:

- Where the implementing class can specify if the listener is one shot. This means that the listener will receive only the first qualified event and is removed after receiving its first event.
- By using the PACE connection with an explicit remove listener call.

To explicitly remove the event listener that was created in Step 1:

```
// unregister the listener
// note we must use the same references for the handler
// and the qualifier from the addAsyncOperationListener
// method call
connection.removeAsyncOperationLister(handler, qualifier);
```

Event Handling

When an event is delivered to your registered listener, you must execute any logic that is required. However, because the thread delivering the event does so from the Cisco BAC client library, you must exercise caution.

When running any logic for handling events:

- Avoid any complex logic for your registered listener that uses a Cisco BAC client library thread. If the thread is busy processing the listener, the thread may not be able to deliver events to other listeners or batch results to threads that have completed synchronous posting.
- Re-accessing the PACE connection can cause a deadlock. For example, if you receive an event and then try to submit a new batch while handling the event with the current thread, a deadlock can occur in the client library.

To avoid these issues, we recommend that you:

- Keep the logic in your listener short.
- Avoid re-accessing the PACE connection. If you require a more complex logic, you can notify any one of your threads for the processing.

Event Reliability

The client library receives events when it maintains a connection with the RDU. If the connection is lost (for example, because of a network crash), events may be lost. You cannot retrieve missed events.

You may also lose events that are generated from the DPE, such as a CWMP inform event. For example, an interruption in the connection from the DPE to the RDU makes it impossible for the DPE to forward the events to the RDU, and from there, to the client.

For more information on how the client library communicates with the RDU, see:

- Use Cases
- Getting Started with the BAC API

CHAPTER 6

Getting Started with the BAC API

This chapter describes the startup process involving system configuration and API execution.

This chapter contains the following sections:

Startup Process for API Client

The startup process for an API client interaction involves:

Configuring the System

Before executing a simple client, ensure that you have completed the tasks listed in this section.

Note These tasks are part of an initial configuration workflow that you must complete before executing a simple client for the first time. Thereafter, you can execute any number of simple clients.

Table 6-1 System Configuration Workflow

Task		Refer to
1.	Install Java Development Kit version 1.6.	Sun Microsystems support site
2.	Ensure that files bpr.jar and bacbase.jar are available in the classpath. These .jar files are located in the <*BPR_HOME*>/lib directory.	—
3.	Access the Cisco BAC administrator user interface and ensure that the password that you set for the default **bacadmin** username matches the password that you set on the RDU. The default password is **changeme**.	*Cisco Broadband Access Center 3.8 Administrator Guide*

Table 6-1 System Configuration Workflow (continued)

Task	Refer to
4. Add a valid license for each technology that you provision, specifically for the CPE WAN Management Protocol (CWMP) technology and for the DPE component.	*Cisco Broadband Access Center 3.8 Administrator Guide*
5. From the administrator user interface, ensure if the DPE is registered with the RDU. To verify if the DPE is registered, check the DPE status from **Servers > DPEs> View Device Provisioning Engines Details** page.	*Cisco Broadband Access Center 3.8 Administrator Guide*

Executing the API Client

To execute a simple API client:

Note This procedure uses the *AddDeviceExample.java* classfile as an example.

Step 1 Compile the API classfile using the following code:

```
javac -classpath .:bpr.jar:bacbase.jar class_file
```

For example:

```
javac -classpath .:bpr.jar:bacbase.jar AddDeviceExample.java
```

Note This example assumes that the *bpr.jar* and *bacbase.jar* files exist in the local directory.

Step 2 Execute the API classfile using the following code:

```
java -cp .:bpr.jar:bacbase.jar class_file
```

For example:

```
java -cp .:bpr.jar:bacbase.jar AddDeviceExample.java
```

Step 3 Verify the results.

For example, the AddDeviceExample will print success or failure messages. If there is no error, the following message appears:

```
Successfully provisioned device with identifier [OUI-serial-12345]
```

You can also verify the results for the device record from the administrator user interface from the **Devices > Manage Device** page. For more information, see Cisco Broadband Access Center 3.8 Administrator Guide.

Creating an API Client

This section describes how you can connect to the RDU, create a batch, post the batch to the RDU, and verify the result.

Note This procedure uses the *AddDeviceExample.java* classfile as an example.

Step 1 Create a connection to the Provisioning API Command Engine (PACE).

```
// The PACE connection to use throughout the example.  When
// executing multiple batches in a single process, it is advisable
// to use a single PACE connection that is retrieved at the start
// of the application.  When done with the connection, YOU MUST
// explicitly close the connection with the releaseConnection()
// method call.
PACEConnection connection = null;

// ---------------------------------------------------------------------
//
// 1) Connect to the Regional Distribution Unit (RDU).
//
//    The parameters defined at the beginning of this class are
//    used here to establish the connection.  Connections are
//    maintained until releaseConnection() is called.  If
//    multiple calls to getInstance() are called with the same
//    arguments, you must still call releaseConnection() on each
//    connection you received.
//
//    The call can fail for one of the following reasons:
//    - The hostname / port is incorrect.
//    - The authentication credentials are invalid.
//
// ---------------------------------------------------------------------
try
{
    connection = PACEConnectionFactory.getInstance(
        // RDU host
        rduHost,
        // RDU port
        rduPort,
        // User name
        userName,
        // Password
        password);
}
catch (PACEConnectionException pce)
{
    // failed to get a connection
    System.out.println("Failed to establish a PACEConnection to ["
            + userName + "@" + rduHost + ":" + rduPort + "]; " +
            pce.getMessage());
    throw new RuntimeException(pce.getMessage());
}
catch (RDUAuthenticationException bae)
{
    // failed to get a connection
    System.out.println("Failed to establish a PACEConnection to ["
            + userName + "@" + rduHost + ":" + rduPort + "]; " +
              bae.getMessage());
```

```
       throw new RuntimeException(bae.getMessage());
}
// ------------------------------------------------------------------
```

Step 2 Get a new batch instance.

```
// ------------------------------------------------------------------
//
// 2) Get a new batch instance.
//
//    To perform any operations in the Provisioning API, you must
//    first start a batch.  As you make commands against the batch,
//    nothing will actually start until you post the batch.
//    Multiple batches can be started concurrently against a
//    single connection to the RDU.
//
// ------------------------------------------------------------------
Batch myBatch = connection.newBatch(
                        // No reset
                        ActivationMode.NO_ACTIVATION,
                        // No need to confirm activation
                        ConfirmationMode.NO_CONFIRMATION,
                        // No publisining to external database
                        PublishingMode.NO_PUBLISHING);
// ------------------------------------------------------------------
```

Step 3 Register the `AddDeviceExample( )` call with the batch.

```
// ------------------------------------------------------------------
//
// 3) Register the add(...) call with the batch.
//
//    Add to the batch the add(...) call.  This will make
//    the batch add the device during the post() operation.  If
//    multiple methods are added to a batch, they will be executed
//    in the order they are registered.  For example, you could
//    add a device and then modify it successfully in a batch.
//
//    The host name and domain name only needs to be specified if the
//    device should have an explicit name assigned to it -- and this is
//    only really useful if you have dynamic DNS enabled in DHCP/CNR.
//    Properties can be used to store additional information that
//    should be maintained by BAC.  This data will be returned as a
//    response to a query for device details.
//
// ------------------------------------------------------------------

// A CWMP device requires the following properties to
// be populated.
//
Map<String, Object> propMap = new HashMap<String, Object>();
propMap.put(IPDeviceKeys.HOME_PROV_GROUP, provisioningGroup);


myBatch.add(
            // Device type
            DeviceType.CWMP,
            // Device identifier
            new CWMPDeviceID(deviceId),
            // Host name - Not used in this example
            null,
            // Domain Name - Not used in this example
            null,
            // ownerID
            ownerId,
```

```
            // classOfService - Use default COS
            null,
            // properties
            propMap);

// ------------------------------------------------------------------
```

Step 4 Post a batch to the RDU.

```
//
// 4) Post the batch to the server.
//
//    Executes the batch against the RDU.  All of the
//    methods are executed in the order entered and the data
//    changes are applied against the embedded database in RDU.
//
// ------------------------------------------------------------------
BatchStatus batchStatus = null;
try
{
    batchStatus = myBatch.post();
}
catch (ProvisioningException pe)
{
    System.out.println("Failed to provision device with identifer ["
        + deviceId + "]; " + pe.getMessage());

    throw new RuntimeException(pe.getMessage());
}

// ------------------------------------------------------------------
```

Step 5 Verify the result of the connection.

```
//
// 5) Check to see if the batch was successfully posted.
//
//    Verify if any errors occurred during the execution of the
//    batch.  Exceptions occur during post() for truly exception
//    situations such as failure of connectivity to RDU.
//    Batch errors occur for inconsistencies such as no lease
//    information for a device requiring activiation.  Command
//    errors occur when a particular method has problems, such as
//    trying to add a device that already exists.
//
// ------------------------------------------------------------------
if (batchStatus.isError())
{
    // Batch error occurred.
    // we need to determine if it was a batch error or a
    // command error that caused this failure

    if (batchStatus.getFailedCommandIndex() == -1)
    {
        // this is a batch only error
        // get the error code and get the error message
        final StringBuilder msg = new StringBuilder(128);
        msg.append("Batch with ID [");
        msg.append(batchStatus.getBatchID());
        msg.append("] failed with error code [");
        msg.append(batchStatus.getStatusCode());
        msg.append("]. [");
        msg.append(batchStatus.getErrorMessage());
        msg.append("].");
```

```
        // throw an exception or log the message
        System.out.println("Failed to add device with identifier ["
                + deviceId + "]; " +  msg.toString());
    }
    else
    {
        // this is a batch error caused by a command
        final CommandStatus commandStatus =
            batchStatus.getFailedCommandStatus();

        // get the error code and get the error message
        final StringBuilder msg = new StringBuilder(128);
        msg.append("Batch with ID [");
        msg.append(batchStatus.getBatchID());
        msg.append("] failed with command error code [");
        msg.append(commandStatus.getStatusCode());
        msg.append("]. [");
        msg.append(commandStatus.getErrorMessage());
        msg.append("].");

        // throw an exception or log the message
        System.out.println("Failed to add device with identifier ["
                + deviceId + "]; " + msg.toString());
    }
}
else
{
    // Successfully added device
    System.out.println("Successfully added device with identifier ["
        + deviceId + "]");
}
```

Step 6 Release the connection to the RDU.

```
// ------------------------------------------------------------------
//
// 6) Release the connection to the RDU.
//
//    Once the last batch has been executed, the connection can
//    be closed to the RDU.  It is important to explictly
//    close connections since it helps ensure clean shutdown of
//    the Java virtual machine.
//
// ------------------------------------------------------------------
connection.releaseConnection();
```

CHAPTER 7

Use Cases

This chapter describes the most common Cisco Broadband Access Center (BAC) API use cases. These use cases are directly related to device provisioning and device management provisioning.

Many system configuration and management operations, such as managing Class of Service, DHCP Criteria, and licenses, are not addressed here because these operations do not require integration with BSS and OSS. You can also use the Cisco BAC administrator user interface to perform most of these activities. See the *Cisco Broadband Access Center 3.8 Administrator Guide,* for details.

For more details on related API calls and sample API client code segments explaining individual API calls and features, refer to these resources that are available in the Cisco BAC installation directory:

- API Javadocs, located at <*BPR_HOME*>/docs/nb-api/javadoc.
- Sample API client code, located at <*BPR_HOME*>/rdu/samples/nb-api.

<*BPR_HOME*> is the home directory in which you install Cisco BAC. The default home directory is /opt/CSCObac.

This chapter lists various API constants and their functions. To execute any API, you must follow the steps described in the Getting Started with the BAC API chapter.

This chapter contains the following sections:

Provisioning Operations

This section describes the following provisioning operation use cases:

Note The classfiles referenced in these use cases; for example, the AddDeviceExample.java classfile that illustrates how you can add a device record to the RDU, are only samples that are bundled with the Cisco BAC software.

- Adding a device record to the RDU—See Table 7-1.
- Searching device records in the RDU—See Table 7-2.
- Associating a device record with a Class of Service in the RDU—See Table 7-3.
- Associating a device record with an owner ID in the RDU—See Table 7-4.
- Modifying a device record in the RDU—See Table 7-5.

- Retrieving device faults cached in BAC servers—See Table 7-6.
- Retrieving discovered device data from the RDU—See Table 7-7.
- Retrieving device data with the property hierarchy—See Table 7-8.
- Retrieving device operation history from the RDU—See Table 7-9.
- Deleting device from the RDU— See Table 7-10

Table 7-1 Adding a Device Record to the RDU

Classfile	API
`AddDeviceExample.java`	IPDevice.add()
Adds a new device record to the RDU database. Uses the IPDevice.add() API and submits the batch synchronously with the NO_ACTIVATION flag. This operation causes the RDU to generate instructions for the device, which are then cached in the DPE. Figure 7-1 explains adding/modifying a device record in the RDU with Activation mode = No_ACTIVATION.	

Table 7-2 Searching Device Records in the RDU

Classfile	API
`SearchDeviceExample.java`	IPDevice.searchDevice()
Searches for a device record in the RDU database. Uses the IPDevice.searchDevice() API and submits the batch synchronously with the NO_ACTIVATION flag.	

Table 7-3 Associating a Device Record with a Class of Service in the RDU

Classfile	API
`ChangeDeviceCoSExample.java`	IPDevice.changeClassOfService()
Associates a device to the specified class of service. Uses the IPDevice.changeClassOfService() API and submits the batch synchronously with the NO_ACTIVATION flag. This operation causes the RDU to generate instructions for the device, which are then cached in the DPE.	

Figure 7-1 Change Device Class of Service (Activation mode= NO_ACTIVATION)

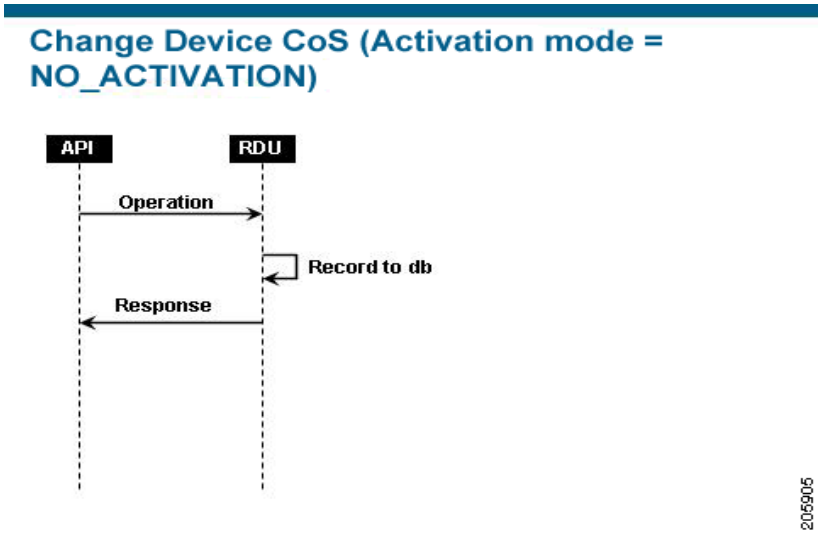


Table 7-4 Associating a Device Record with an Owner in the RDU

Classfile	API
`ChangeDeviceOwnerIdExample.java`	IPDevice.changeOwnerID()
Associates the device record with the owner ID in the RDU. Uses the IPDevice.changeOwnerID() API and submits the batch synchronously with the NO_ACTIVATION flag. This operation causes the RDU to generate instructions for the device, which are then cached in the DPE.	

Table 7-5 Modifying a Device Record in the RDU

Classfile	API
`ModifyDeviceExample.java`	IPDevice.changeProperties()
Changes the properties of a device record stored in the RDU. Uses the IPDevice.changeProperties() API and submits the batch synchronously with the NO_ACTIVATION flag. This operation causes the RDU to generate instructions for the device, which are then cached in the DPE.	

Table 7-6 Retrieving Device Faults Cached in BAC Servers

Classfile	API
`RetrieveFaultsExample.java`	IPDevice.getDetails()
Retrieves the device faults that are stored in the RDU and DPEs. Uses the IPDevice.getDetails() API and submits the batch synchronously with the NO_ACTIVATION flag.	

Table 7-7 Retrieving Discovered Device Data in the RDU

Classfile	API
QueryDeviceExample.java	IPDevice.getDetails()
Retrieves the discovered data of a device that is stored in the RDU. Uses the IPDevice.getDetails() API and submits the batches synchronously using the on-connect mode with the NO_ACTIVATION flag.	

Table 7-8 Retrieving Device Data with the property hierarchy

Classfile	API
QueryDeviceExample.java GetDetails.java DeviceDetailsKeys.java DTE.java ObjectDetails.java	IPDevice.getDetails()
Retrieves the device data along with the property hierarchy of the device, that is stored in the RDU. Uses the IPDevice.getDetails() API and submits the batches using the immediate and on-connect mode.	

Table 7-9 Retrieving Device Operation History from the RDU

Classfile	API
GetDeviceHistoryExample.java	IPDevice.getDeviceHistory()
Retrieves the history of a device that is stored in the RDU. Uses the IPDevice.getDeviceHistory() API and submits the batch synchronously with the NO_ACTIVATION flag.	

Table 7-10 Delete Device from the RDU

Classfile	API
DeleteDeviceExample.java	IPDevice.delete()
Deletes a device from the RDU. Uses the IPDevice.delete() API and submits the batch synchronously with the NO_ACTIVATION flag.	

Device Management Operations

This section describes the following device management operation use cases:

Note The classfiles referenced in these use cases; for example, the GetDeviceLiveDataExample.java classfile that illustrates how you can retrieve live data from a device, are only samples that are bundled with the Cisco BAC software.

- Retrieving live data, such as statistics, from a device—See Table 7-11.

- Executing diagnostics on a device—See Table 7-12.
- Rebooting a device—See Table 7-13.
- Executing diagnostics on a device on its next connection—See Table 7-14.
- Executing proxy operations on a device in asynchronous mode—See Table 7-15.
- Executing the java script based DPE extensions—See Table 7-16.

Table 7-11 Retrieving Live Data from a Device

Classfile	API
`GetDeviceLiveDataImmediateExample.java`	IPDevice.performOperation()
Retrieves live data directly from a device. Uses the IPDevice.performOperation() API to perform the TR-069 RPC GetParameterValues operation on the device and submits the batch synchronously using the immediate operation mode with the AUTOMATIC_ACTIVATION flag (to trigger a session with the device).	
Figure 7-2 explains retrieving live data from devices.	

Figure 7-2 Retrieving Live Data from a Device

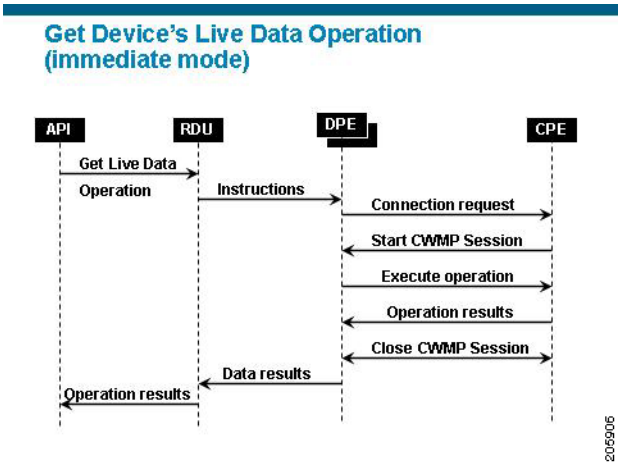


Table 7-12 Executing Diagnostics on a Device

Classfile	API
`CwmpDiagnosticImmediateExample.java`	IPDevice.performOperation()
Executes ping diagnostics on a device. Uses the IPDevice.performOperation() API to perform the TR-069 RPC SetParameterValue and then GetParameterValues operation on the device. Submits the batches synchronously using the immediate operation mode with the AUTOMATIC_ACTIVATION flag (to trigger the sessions with the device).	

Table 7-13 Rebooting a Device

Classfile	API
`RebootDeviceImmediateExample.java`	IPDevice.performOperation()
Reboots a device using the TR-069 RPC Reboot. Uses the IPDevice.performOperation() API in the batch. Submits the batch synchronously using the immediate connection mode with the AUTOMATIC_ACTIVATION flag (to trigger a provisioning session with the device).	

Table 7-14 Executing Diagnostics on a Device on its Next Connection

Classfile	API
`CwmpDiagnosticOnConnectExample.java`	IPDevice.performOperation()
Executes ping diagnostics on a device. Uses the IPDevice.performOperation()API to perform the TR-069 RPC SetParameterValue and then GetParameterValues on the device. Submits the batches in synchronous mode using the on connect mode with the NO_ACTIVATION flag. Figure 7-3 describes the workflow when submitting a batch to set the ping diagnostic parameters in the on-connect mode.	

Figure 7-3 Executing Diagnostics on a Device on its Next Connection

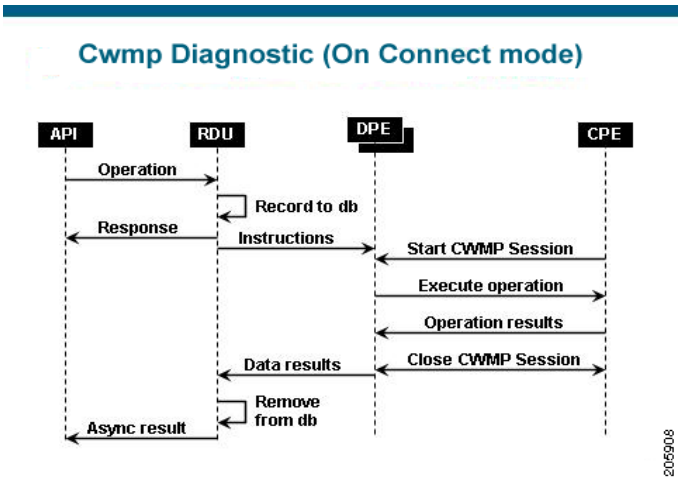


Table 7-15 Executing Diagnostics on a Device in Asynchronous Mode

Classfile	API
`GetLiveDataAsyncImmediateProxyOperationExample.java`	IPDevice.performOperation()

Executes proxy operations on a device. Uses the IPDevice.performOperation()API to run the proxy TR-069 operations like GetParamNames and GetParamValues RPC, on the device.

Asynchronous operation facilitates RDU to execute the asynchronous proxy operation batches received from API client, without reserving a PACE thread till proxy operation result is returned from DPE.

The asynchronous operation submits the batches in asynchronous mode using the activation mode as ASYNC_AUTOMATIC. Figure 7-4 describes the workflow when the API client submits a TR-069 operation request to RDU for running a proxy operation on the device.

Figure 7-4 Executing Diagnostics on a Device in Asynchronous Mode

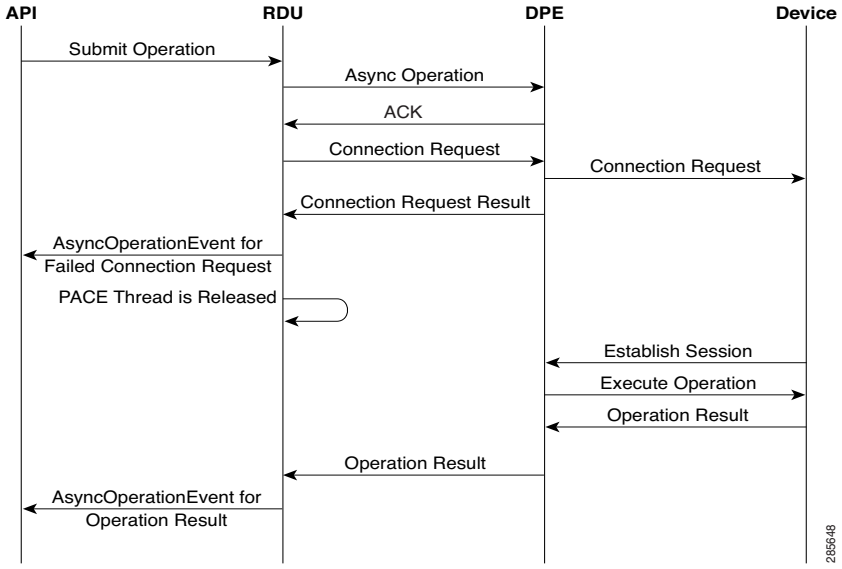


Table 7-16 Executing the Java Script Based DPE Extensions

Script Types	Extensions
DPE Extension Scripts Helper Scripts	soapRequestSender soapResponseSender incomingEventViewer outgoingEventViewer

Scriptable extension service facilitates executing the java scripts based DPE extensions. When the CPE boots and attempts to establish connection with the DPE, the extension scripts will be executed at the following extension points:

- IPDevice/extensions/soapRequestSender
- IPDevice/extensions/soapResponseSender
- IPDevice/extensions/incomingEventViewer
- IPDevice/extensions/outgoingEventViewer

These extensions can be configured in the device property hierarchy at Class of Service (CoS), Device, Group, Provisioning group level and CWMP defaults.

For un-registered devices, the extensions can be configured from the admin UI at CWMP default level.

The extension scripts are added in the RDU database from the admin UI or API client. The following types of extension scripts are added in the RDU database:

- DPE Extension scripts – which have the main provisioning logic
- Helper scripts – which have the utility functions and can be shared by other scripts

Once the extension scripts are added in the RDU database, the RDU pushes the scripts to the DPE cache. When the device boots, and attempts to establish connection with the DPE, the DPE invokes the scriptable extension service and executes the extensions configured on the device property hierarchy.

GLOSSARY

A

alert — A syslog or SNMP message notifying an operator or administrator of a problem.

API — Application programming interface. Specification of function-call conventions that defines an interface to a service.

audit logs — A log file containing a summary of the major changes in the RDU database. This includes the changes to system defaults, technology defaults, and classes of service.

auto configuration server (ACS) — A server that provisions a device or a collection of devices. In BAC, ACS refers to the BAC server, and in some instances, the DPE.

B

broadband — Transmission system that multiplexes multiple independent signals onto one cable. In Telecommunications terminology; any channel having a bandwidth greater than a voice-grade channel (4 kHz). In LAN terminology; a co-axial cable on which analog signaling is used.

Broadband Access Center (BAC) — An integrated solution for managing and provisioning broadband home networks. BAC is a scalable product capable of supporting millions of devices.

Business Support Systems(BSS) — Components that service providers use to run business operations. The roles of a BSS in a service provider network include managing products, customers, revenue, and orders.

C

caching — Form of replication in which information learned during a previous transaction is used to process later transactions.

cipher suites — A set of cryptographic algorithms that the SSL module requires to perform key exchange, authentication, and Message Authentication Code.

customer premises equipment (CPE) — Terminating equipments, such as telephones, computers, and modems, supplied and installed at a customer location.

CPE WAN Management Protocol (CWMP) — A standard defined in the TR-069 specification by the Broadband Forum. CWMP integrates the capabilities defined in TR-069 to increase operator efficiency and reduce network management problems.

D

device provisioning engine (DPE) Distributed servers that cache device instructions and perform CWMP services. DPEs automatically synchronize with the RDU to obtain the latest instructions, and provide BAC scalability and redundancy.

F

fully qualified domain name (FQDN) 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.

H

HTTPS See Secure Sockets Layer and Transport Layer Security.

I

instruction generation service (IGS) The process of generating instructions at the RDU, for devices defined by a search criteria, and distributing these instructions to the DPE, which then caches the instructions. The instructions inform the DPE the actions to be performed on the CPE, which may include configuration, firmware upgrade, or other operations.

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.

N

network address translation (NAT) 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.

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 Time Protocol (NTP) A protocol designed to synchronize server clocks over a network.

O

Operations Support Systems(OSS) Computer systems used by telecommunication providers, dealing with telecom network, customers and support processes.

P

provisioning API A series of BAC functions that programs can use to make the operating system perform various functions.

provisioning groups Groupings of devices with a defined set of associated DPE servers, based on either network topology or geography.

publishing Publishing provides provisioning information to an external datastore in real time. Publishing plug-ins must be developed to write data to a datastore.

PACE Provisioning API Command Engine.

R

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.

regional distribution unit (RDU) The RDU is the primary server in the BAC provisioning system. It manages generation of device instructions, processes all API requests, and manages the BAC system.

S

Secure Sockets Layer (SSL) A protocol for transmitting private documents via the Internet. SSL uses a cryptographic system that uses two keys to encrypt data: a public key known to everyone and a private or secret key known only to the recipient of the message. URLs that require an SSL connection start with https: instead of http:. BAC supports SSLv3.

See Transport Layer Security.

shared secret A character string used to provide secure communication between two servers or devices.

T

template files XML files that contain configuration or firmware rules for devices.

Transport Layer Security (TLS) A protocol that guarantees privacy and data integrity between client/server applications communicating over the Internet. BAC supports TLSv1.

See Secure Sockets Layer.

TR-069 A standard which defines the CPE WAN Management Protocol (CWMP). TR-069 enables communication between CPE and an autoconfiguration server.

V

Voice over IP (VoIP) Mechanism to make telephone calls and send faxes over IP-based data networks with a suitable quality of service (QoS) and superior cost savings.

W

watchdog agent A watchdog agent is a daemon process that is used to monitor, stop, start, and restart BAC component processes such as the RDU, JRun, and SNMP agent.

INDEX

T

W