Java API Client Use Cases

This chapter describes the most common Prime Cable Provisioning API use cases that are directly related to device provisioning and device management.

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 use the Prime Cable Provisioning admin UI to perform most of these activities. For more information on Prime Cable Provisioning admin UI, see the *Cisco Prime Cable Provisioning 7.4 User Guide*.

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 Prime Cable Provisioning installation directory:

- API Javadoc located at *BAC_61_Linux/docs/BAC_Javadoc_API_Provisioning* on a Linux machine.
- Sample API client code, located at *BPR_HOME/rdu/samples/provapi*.

BPR_HOME is the home directory in which you install Prime Cable Provisioning. 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 Prime Cable Provisioning API chapter.

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 Prime Cable Provisioning software.

- Adding a device record to the RDU—See Table 7-1.
- Modifying a device record in the RDU—See Table 7-2.
- Retrieving discovered device data from the RDU—See Table 7-3.
- Deleting device from the RDU—See Table 7-4.
- Retrieve Devices Matching Vendor Prefix—See Table 7-5.

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 or modifying a device record in the RDU with Activation mode = No_ACTIVATION.	

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

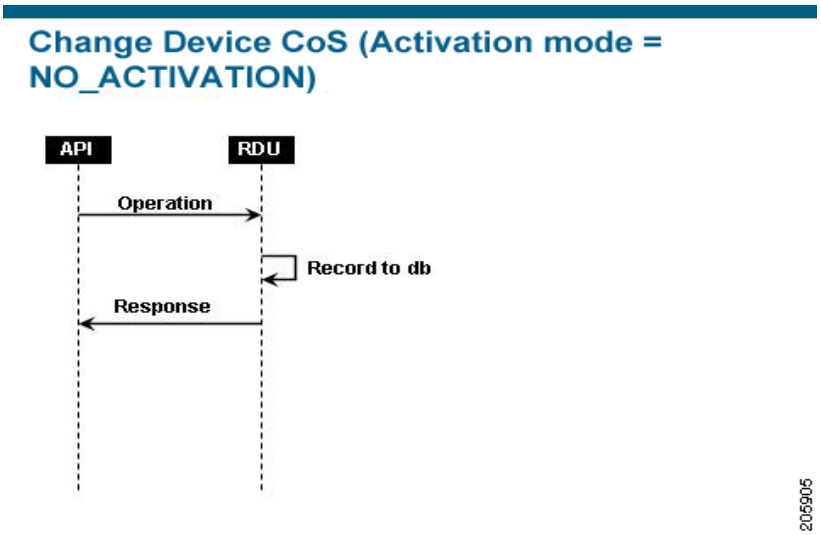


Table 7-2 Modifying a Device Record in the RDU

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 7-3 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 7-4 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.	

Table 7-5 Retrieve Devices Matching Vendor Prefix

Classfile	API
RetriveDevicesMatchingVendorPrefix.java	IPDevice.searchDevice()
Searches for devices that exist in the database. Uses the IPDevice.searchDevice() API to query a list of devices that exist in the database.	

Provisioning API Use Cases

This section presents a series of the most common provisioning application programming interface (API) use cases. See the Cisco Prime Cable Provisioning 7.2.1 API Javadoc for more details and sample Java code segments explaining individual API calls and features.

These use cases are directly related to device provisioning, service provisioning, or both. Many administrative operations, such as managing Class of Service, DHCP Criteria, and licenses are not addressed here. We recommend that you go through the API javadoc for more details on the related API calls. You can also use the administrator user interface to perform most of these activities.

This section describes:

- How to Create an API Client, page 7-3
- Use Cases, page 7-6

How to Create an API Client

Before going through the use cases, you must familiarize yourself with how to create an API client. Use the workflow described in this section to create the API client.

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;

// 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.
// - The maximum number of allowed sessions for the user
// has already been reached.

// However, the number of session validation for an user can be
// avoided by using the following overloaded method :
//
// PACEConnectionFactory.getInstance(
    // RDU host     rduHost,
    // RDU port     rduPort,
    // User name    userName,
    // Password     password
    // Is immediate authentication required authenticateImmediately
    // create a session forcefully          forceLogin
// )
try
{
    connection = PACEConnectionFactory.getInstance(

         // RDU host      rduHost,
         // RDU port      rduPort,
         // User name     userName,
         // Password       password);
}
catch (PACEConnectionException e)
{
// Handle connection error:

System.out.println("Failed to establish a PACEConnection to [" +
    userName + "@" + rduHost + ":" + rduPort + "]; " +
    e.getMessage());

System.exit(1);
}
```

Step 2 Create a new instance of a batch.

```
// To perform any operations in the Provisioning API, you must
// first create a batch. As you add commands to the batch,
// nothing gets executed 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 publishing to external database

    PublishingMode.NO_PUBLISHING);
```

Step 3 Register an API command with the batch. The example featured in this step uses the *getDetails(...)* call.

```
// Use the Provisioning API to get all of the information for
// the specified MAC address. Since methods aren't actually
// executed until the batch is posted, the results are not
// returned until after post() completes. The getCommandStatus()
// followed by getData() calls must be used to access the results
// once the batch is posted.

final DeviceID modemMACAddress = DeviceID.getInstance("1,6,00:11:22:33:44:55",
    KeyType.MAC_ADDRESS);

List options = new ArrayList();
options.add(DeviceDetailsOption.INCLUDE_LEASE_INFO);

myBatch.getDetails(modemMACAddress, options);
```

Step 4 Post the batch to the Regional Distribution Unit (RDU) server.

```
// Executes the batch against the RDU. All of the
// methods are executed in the order entered.

BatchStatus bStatus = null;
try
{
// Post batch in synchronous fashion without a timeout. This method will block until
// results are returned. Other API calls are available to submit a batch with timeout
// or in asynchronous (non-blocking) fashion.

bStatus = myBatch.post();
}
catch (ProvisioningException pe)
{
System.out.println("Failed to query for modem with MAC address [" +
    modemMACAddress + "]; " + pe.getMessage());

System.exit(2);
}
```

Step 5 Check the status of the batch.

```
// Check if any errors occurred during the execution of the
// batch. Exceptions occur during post() for truly exceptional
// situations such as failure of connectivity to RDU.
// Batch errors occur for inconsistencies such as no lease
// information for a device requiring activation. Command
// errors occur when a particular method has problems, such as
// trying to add a device that already exists.

//check batchStatus and commandStatus
//for any error

CommandStatus commandStatus = null;
if (batchStatus.getCommandCount() > 0)
{
     commandStatus = batchStatus.getCommandStatus(0);
}
if (batchStatus.isError()
    || commandStatus == null
    || commandStatus.isError())
{
        System.out.println("Failed to query for modem with MAC address [" +
           modemMACAddress + "]; " + bs.getStatusCode().toString() + ", "
           + bs.getErrorMessage());
        for (int i = 0; i < bs.getCommandCount(); i++)
        {
```

```
            CommandStatus cs = bs.getCommandStatus(i);
            System.out.println("Cmd " + i + ": status code "
                    + cs.getStatusCode().toString() + ", " + cs.getErrorMessage());
        }
}
```

If there is no error, the batch call returns a successful result.

```
// Successfully queried for device.

System.out.println("Queried for DOCSIS modem with MAC address ["+
    modemMACAddress + "]");

// Display the results of the command (TreeMap is sorted). The
// data returned from the batch call is stored on a per-command
// basis. In this example, there is only one command, but if
// you had multiple commands all possibly returning results, you
// could access each result by the index of when it was added.
// The first method added is always index 0. From the status of
// each command, you can then access the accompanying data by
// using the getData() call. Since methods can return data of
// different types, you will have to cast the response to the
// type indicated in the Provisioning API documentation.

Map deviceData = (Map)bStatus.getCommandStatus(0).getData();

// Created a sorted map view

Map<String, Object> deviceDetails = new TreeMap(deviceData);
    for(String key: deviceDetails.keySet())
{
    System.out.println(" " + key + "=" + deviceDetails.get(key));
}
```

Step 6 Release the connection.

```
// Once the last batch has been executed, the connection can
// be closed to the RDU. It is important to explicitly
// close connections since it helps ensure clean shutdown of
// the Java virtual machine.

connection.releaseConnection();
```

Use Cases

This section includes these use cases:

- Self-Provisioned Modem and Computer in Fixed Standard Mode, page 7-7
- Adding a New Computer in Fixed Standard Mode, page 7-10
- Disabling a Subscriber, page 7-13
- Preprovisioning Modems/Self-Provisioned Computers, page 7-15
- Modifying an Existing Modem, page 7-17
- Unregistering and Deleting a Subscriber's Devices, page 7-18
- Self-Provisioning First-Time Activation in Promiscuous Mode, page 7-22
- Bulk Provisioning 100 Modems in Promiscuous Mode, page 7-25

- Preprovisioning First-Time Activation in Promiscuous Mode, page 7-27
- Replacing an Existing Modem, page 7-29
- Adding a Second Computer in Promiscuous Mode, page 7-31
- Self-Provisioning First-Time Activation with NAT, page 7-31
- Adding a New Computer Behind a Modem with NAT, page 7-32
- Move Device to Another DHCP Scope, page 7-32
- Log Device Deletions Using Events, page 7-33
- Monitoring an RDU Connection Using Events, page 7-34
- Logging Batch Completions Using Events, page 7-35
- Getting Detailed Device Information, page 7-35
- Searching Using the Device Type, page 7-40
- Searching for Devices Using Vendor Prefix or Class of Service, page 7-41
- Preprovisioning PacketCable eMTA/eDVA, page 7-42
- SNMP Cloning on PacketCable eMTA/eDVA, page 7-44
- Incremental Provisioning of PacketCable eMTA/eDVA, page 7-45
- Preprovisioning DOCSIS Modems with Dynamic Configuration Files, page 7-48
- Optimistic Locking, page 7-49
- Temporarily Throttling a Subscriber's Bandwidth, page 7-51
- Preprovisioning CableHome WAN-MAN, page 7-53
- CableHome with Firewall Configuration, page 7-54
- Retrieving Device Capabilities for CableHome WAN-MAN, page 7-56
- Self-Provisioning CableHome WAN-MAN, page 7-57
- RBAC Administration and Operational Access Control, page 7-59
- Update Protection for Properties at Device Level, page 7-65
- Domain Administration and Instance Level Access Control, page 7-68

Self-Provisioned Modem and Computer in Fixed Standard Mode

The subscriber has a computer installed in a single-dwelling unit and has purchased a DOCSIS cable modem. The computer has a web browser installed.

Desired Outcome

Use this workflow to bring a new unprovisioned DOCSIS cable modem and computer online with the appropriate level of service.

Step 1 The subscriber purchases and installs a DOCSIS cable modem at home and connects a computer to it.

Step 2 The subscriber powers on the modem and the computer, and Prime Cable Provisioning gives the modem restricted access. The computer and modem are assigned IP addresses from restricted access pools.

Step 3 The subscriber starts a web browser, and a spoofing DNS server points the browser to a service provider's registration server (for example, an OSS user interface or a mediator).

Step 4 The subscriber uses the service provider's user interface to complete the steps required for registration, including selecting Class of Service.

Step 5 The service provider's user interface passes the subscriber's information, such as the selected Class of Service and computer IP address, to Prime Cable Provisioning, which then registers the subscriber's modem and computer.

```
// First we query the computer's information to find the
// modem's MAC Address. We use the computer IP Address (the web browser
// received this when the subscriber opened the service provider's
// web interface

PACEConnection connection = PACEConnectionFactory.getInstance(
            "localhost", 49187, "admin", "changeme");

// 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.
// NO_PUBLISHING is the publishing mode because we are not attempting
// to publish to external database.

Batch batch = connection.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);

// register getAllForIPAddress to the batch

batch.getAllForIPAddress("10.0.14.38");

BatchStatus batchStatus = null;

// post the batch to RDU server

try
{
batchStatus = batch.post();
}
catch(ProvisioningException e)
{
e.printStackTrace();
}
// Get the LeaseResults object after posting a batch.

CommandStatus commandStatus = batchStatus.getCommandStatus(0);

LeaseResults computerLease = (LeaseResults)commandStatus.getData();

// Derive the modem MAC address from computer's network
// information. The "1,6" is a standard prefix for an Ethernet
// device. The fully qualify MAC Address is required by PCP

StringBuffer modemMACAddress = new StringBuffer();
modemMACAddress.append("1,6,");
modemMACAddress.append(computerLease.getSingleLease().get("relay-agent-remote-id"));
```

```
// Create MacAddress object from the string

MACAddress modemMACAddressObject = new MACAddress(modemMACAddress.toString());

List<DeviceID> modemDeviceIDList = new ArrayList<DeviceID>();
    modemDeviceIDList.add(modemMACAddressObject);

// Create a new batch to add modem device

batch = connection.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);

// Register add API to the batch

batch.add(DeviceType.DOCSIS, modemDeviceIDList,
    null, null, "0123-45-6789", "silver", "provisioned-cm", null);

// post the batch to RDU server

// Derive computer MAC address from computer's network information.

String computerMACAddress =
    (String)computerLease.getSingleLease().get(DeviceDetailsKeys.MAC_ADDRESS);

// Create a map for computer property.

Map<String, Object> properties = new HashMap<String, Object>();
properties.put(IPDeviceKeys.MUST_BE_BEHIND_DEVICE, modemMACAddress.toString());

List<DeviceID> compDeviceIDList = new ArrayList<DeviceID>();
MACAddress computerMACAddressObject = new MACAddress(computerMACAddress);
compDeviceIDList.add(computerMACAddressObject);

// Register add API to the batch

batch.add(DeviceType.COMPUTER, compDeviceIDList,
    null, null, "0123-45-6789", null, "provisioned-cpe", properties);
try
{
    batchStatus = batch.post();
}
catch(ProvisioningException e)
{
    e.printStackTrace();
}
```

Step 6 The provisioning client calls *performOperation(DeviceOperation deviceOperation, DeviceID deviceID, Map<String,Object> parameters* to reboot the modem and gives the modem provisioned access.

```
// Reset the computer
// Create a new batch
```

```
        batch = connection.newBatch(

            // No reset

            ActivationMode.AUTOMATIC,

            // No need to confirm activation

            ConfirmationMode.NO_CONFIRMATION);

        // Register performOperation command to the batch

        batch.performOperation(DeviceOperation.RESET, modemMACAddressObject, null);

        // Post the batch to RDU server

        try
        {
            batchStatus = batch.post();
        }
        catch(ProvisioningException e)
        {
            e.printStackTrace();
            }
        }
}
```

Step 7 The user interface prompts the subscriber to reboot the computer.

After rebooting, the computer receives a new IP address, and both cable modem and computer are now provisioned devices. The computer has access to the Internet through the service provider's network.

Adding a New Computer in Fixed Standard Mode

A Multiple System Operator (MSO) lets a subscriber have two computers behind a cable modem. The subscriber has one computer already registered and then brings home a laptop from work and wants access. The subscriber installs a hub and connects the laptop to it.

Desired Outcome

Use this workflow 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 powers on the new computer and Prime Cable Provisioning gives it restricted access.

Step 2 The subscriber starts a web browser on the new computer and a spoofing DNS server points it 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 to add a new computer.

Step 4 The service provider's user interface passes the subscriber's information, such as the selected Class of Service and computer IP address, to Prime Cable Provisioning, which then registers the subscriber's modem and computer.

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

```
PACEConnection connection = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "changeme");

// 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
// NO_PUBLISHING is the publishing mode because we are not attempting
// to publish to external database.

Batch batch = connection.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);

// register getAllForIPAddress to the batch

batch.getAllForIPAddress("10.0.14.39");
BatchStatus batchStatus = null;

// post the batch to RDU server

try
{
    batchStatus = batch.post();
}
catch(ProvisioningException e)
{
    e.printStackTrace();
}
// Get the LeaseResults object after posting a batch.

CommandStatus commandStatus = batchStatus.getCommandStatus(0);

LeaseResults computerLease = (LeaseResults)commandStatus.getData();

// derive the modem MAC address from computer's network
// information. The "1,6" is a standard prefix for an Ethernet
// device. The fully qualify MAC Address is required by PCP

StringBuffer modemMACAddress = new StringBuffer();
modemMACAddress.append("1,6,");
modemMACAddress.append(computerLease.getSingleLease().get("relay-agent-remote-id"));

// derive computer MAC address from computer's network information.

String computerMACAddress =
    (String)computerLease.getSingleLease().get(DeviceDetailsKeys.MAC_ADDRESS);

//Create a map for computer property.

Map<String, Object> properties = new HashMap<String, Object>();

// setting 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)
        // and hence fixed mode.

        properties.put(IPDeviceKeys.MUST_BE_BEHIND_DEVICE, modemMACAddress);

        // the IPDeviceKeys.MUST_BE_IN_PROV_GROUP ensures that the computer
        // will receive its provisioned access only when it is brought up in
        // the specified provisioning group. This prevents the computer
        // (and/or) the modem from moving from one locality to another
        // locality.

        properties.put(IPDeviceKeys.MUST_BE_IN_PROV_GROUP, "bostonProvGroup");

        List<DeviceID> compDeviceIDList = new ArrayList<DeviceID>();
        MACAddress computerMACAddressObject = new MACAddress(computerMACAddress);
        compDeviceIDList.add(computerMACAddressObject);

        batch = connection.newBatch(

            // No reset

            ActivationMode.NO_ACTIVATION,

            // No need to confirm activation

            ConfirmationMode.NO_CONFIRMATION,

            // No publishing to external database

            PublishingMode.NO_PUBLISHING);

        // register add API to the batch

        batch.add(
            DeviceType.COMPUTER,    // deviceType: Computer
            compDeviceIDList,       // compDeviceIDList: the list of DeviceIDs derived from
                                    // computerLease
            null,                   // hostName: not used in this example
            null,                   // domainName: not used in this example
            "0123-45-6789",         // ownerName
            null,                   // class of service: get the default COS
            "provisionedCPE",       // dhpcCriteria: Network Registrar uses this to
                                      // select a modem lease granting provisioned IP address
            properties                // device properties
        );


        // post the batch to RDU server

        try
        {
            batchStatus = batch.post();
        }
        catch(ProvisioningException e)
        {
            e.printStackTrace();
            }
        }
    }
```

Step 5 The user interface prompts the subscriber to reboot the new computer so that Prime Cable Provisioning can give the computer its registered service level.

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

Disabling a Subscriber

A service provider needs to disable a subscriber from accessing the Internet due to recurring nonpayment.

Desired Outcome

Use this workflow to disable an operational cable modem and computer, so that the devices temporarily restrict Internet access for the user. Additionally, this use case can redirect the user's browser to a special page that could announce:

```
You haven't paid your bill so your Internet access has been disabled.
```

Step 1 The service provider's application uses a provisioning client program to request a list of all of the subscriber's devices from Prime Cable Provisioning.

Step 2 The service provider's application then uses a provisioning client to individually disable or restrict each of the subscriber's devices.

```
PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");

//get all for owner ID

Batch batch = conn.newBatch();
batch.getAllForOwnerID("0123-45-6789");

BatchStatus batchStatus = null;
try
{
    batchStatus = batch.post();
}
catch(Exception e)
{
    e.printStackTrace();
}
CommandStatus commandStatus = batchStatus.getCommandStatus(0);


//batch success without error, retrieve the result

RecordSearchResults rcSearchResult = (RecordSearchResults)commandStatus.getData();
List<RecordData> resultList = rcSearchResult.getRecordData();

if (resultList != null)
{
// getting the data

for (int i=0; i<resultList.size(); i++)
{
    RecordData rd = resultList.get(i);
    Map<String, Object> detailMap = rd.getDetails();

    //get the deviceType from the detail map

    String deviceType =
        (String)detailMap.get(DeviceDetailsKeys.DEVICE_TYPE);
```

```
            Key primaryKey = rd.getPrimaryKey();

            //only interest in DOCSIS
            if (DeviceType.getDeviceType(deviceType)
                .equals(DeviceType.DOCSIS))
            {

            //change COS

            batch = conn.newBatch();
            batch.changeClassOfService((DeviceID)primaryKey, "DisabledCOS");

            //change DHCPCriteria

            batch.changeDHCPCriteria((DeviceID)primaryKey, "DisabledDHCPCriteria");

            batchStatus = null;
            try
            {
                 batchStatus = batch.post();
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }

        }
        //disable computer

        else if (DeviceType.getDeviceType(deviceType)
            .equals(DeviceType.COMPUTER))
        {
            //change DHCPCriteria

            batch = conn.newBatch();
            batch.changeClassOfService((DeviceID)primaryKey,
                "DisabledComputerCOS");
            batch.changeDHCPCriteria((DeviceID)primaryKey,
                "DisabledComputerDHCPCriteria");

            batchStatus = null;
            try
            {
                batchStatus = batch.post();
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }
        }
    }
```

Note You may need to consider the impact on the CPE behind the modem when defining the characteristics of DisabledCOS and resetting the modem. This is especially important if you have voice endpoints behind the modem, because disrupting the cable modem might affect the telephone conversation in progress at that time.

The subscriber is now disabled.

Preprovisioning Modems/Self-Provisioned Computers

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 its cable modems in bulk.

Desired Outcome

Use this workflow to bring a preprovisioned cable modem, and an unprovisioned computer, online in the roaming standard mode. This must be done so that both devices have the appropriate level of service and are registered.

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

Step 2 The service provider selects 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 Prime Cable Provisioning gives it a provisioned IP address.

Step 5 The technician turns on the computer and Prime Cable Provisioning 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 accesses the service provider's user interface to complete the steps required for registering the computer behind the provisioned cable modem.

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

PACEConnection connection = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "changeme");

// 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
// NO_PUBLISHING is the publishing mode because we are not attempting
// to publish to external database.

Batch batch = connection.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);

// register getAllForIPAddress to the batch

batch.getAllForIPAddress("10.0.14.38");

BatchStatus batchStatus = null;

// post the batch to RDU server
```

```
try
{
    batchStatus = batch.post();
}
catch(ProvisioningException e)
{
    e.printStackTrace();
}

// Get the LeaseResults object after posting a batch.

CommandStatus commandStatus = batchStatus.getCommandStatus(0);

LeaseResults computerLease = (LeaseResults)commandStatus.getData();

// derive computer MAC address from computer's network information.
String computerMACAddress =
    (String)computerLease.getSingleLease().get(DeviceDetailsKeys.MAC_ADDRESS);


    List<DeviceID> compDeviceIDList = new ArrayList<DeviceID>();
    MACAddress computerMACAddressObject = new MACAddress(computerMACAddress);
    compDeviceIDList.add(computerMACAddressObject);


// NO_ACTIVATION will generate new configuration for the computer,
// however it will not attempt to reset it.
// NO_CONFIRMATION is the confirmation mode because we are not
// attempting to reset the computer because this cannot be done.

batch = connection.newBatch(
    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);

// register add API to the batch

batch.add(
    DeviceType.COMPUTER,  // deviceType: Computer
    compDeviceIDList,     // compDeviceIDList: the list of DeviceIDs derived from
                          // computerLease
    null,                 // hostName: not used in this example
    null,                 // domainName: not used in this example
    "0123-45-6789",       // ownerName
    null,                 // class of service: get the default COS
    "provisionedCPE",     // dhpcCriteria: Network Registrar uses this to
                          // select a modem lease granting provisioned IP address
    null                  // properties: not used
);

// post the batch to RDU server

try
{
    batchStatus = batch.post();
```

```
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
}
```

Note The *IPDeviceKeys.MUST_BE_BEHIND_DEVICE* property is not set on the computer and this allows roaming from behind one cable modem to another.

Step 8 The technician restarts the computer and the computer receives a new provisioned 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.

Modifying an Existing Modem

A service provider's subscriber currently has a level of service known as **Silver** and has decided to upgrade to **Gold** service. The subscriber has a computer installed at 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

Use this workflow to 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** to **Gold**.

Step 2 The service provider's application makes these API calls in Prime Cable Provisioning:

```
// 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
// NO_PUBLISHING is the publishing mode because we are not attempting
// to publish to external database.

Batch batch = connection.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);

// replace changeClassOfService to this. Make sure the comment
// on top of this line is still there.
```

```
    batch.changeClassOfService(new MACAddress("1,6,00:11:22:33:44:55")

        // the MACAddress object

        , "Gold");


    // post the batch to the RDU

    BatchStatus batchStatus = null;
    try
    {
        batchStatus = batch.post();
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
}
```

The subscriber can now access the service provider's network with the *Gold* service.

Unregistering and Deleting a Subscriber's Devices

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

Desired Outcome

Use this workflow to permanently remove all 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 This step describes how to unregister a subscriber's device and delete a subscriber's device. Deleting a device is optional because some service providers prefer to keep the cable modem in the database unless it is broken. Note however that if you unregister a device using Step 2-a, you cannot delete the device using Step 2-b.

a. To unregister a device, the service provider's application uses a provisioning client program to request a list of all the subscriber's devices from Prime Cable Provisioning, and unregisters and resets each device so that it is brought down to the default (unprovisioned) service level.

Note If the device specified as the parameter to the "unregister" API is already in unregistered state then the status code from the API call will be set to `CommandStatusCodes.CMD_ERROR_DEVICE_UNRE- GISTER_UNREGISTERED_ERROR`. This is normal/expected behavior.

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

// Create a new connection

PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");
```

```
// 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
// NO_PUBLISHING is the publishing mode because we are not attempting
// to publish to external database.

Batch batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);

batch.getAllForOwnerID("0123-45-6789"

//  query all the devices for this account number
);

BatchStatus batchStatus = null;
try
{
    batchStatus = batch.post();
}
catch(Exception e)
{
    e.printStackTrace();
}
CommandStatus commandStatus = batchStatus.getCommandStatus(0);


//batch success without error, retrieve the result

RecordSearchResults rcSearchResult = (RecordSearchResults)commandStatus.getData();
List<RecordData> resultList = rcSearchResult.getRecordData();

//  We need to unregister all the devices behind each modem(s) or else the
//  unregister call for that modem will fail.

if (resultList != null)
{
  //Unregister the COMPUTER
  for (int i=0; i<resultList.size(); i++)
{
    RecordData rd = resultList.get(i);
    Map<String, Object> detailMap = rd.getDetails();

    //get the deviceType from the detail map

    String deviceType = (String)detailMap.get(DeviceDetailsKeys.DEVICE_TYPE);


    //only interest in DOCSIS

    if (DeviceType.getDeviceType(deviceType) .equals(DeviceType.COMPUTER))
    {
        Key primaryKey = rd.getPrimaryKey();
```

```
            batch = conn.newBatch();
            batch.unregister((DeviceID)primaryKey);

            batchStatus = null;
            try
            {
               batchStatus = batch.post();
            }
            catch(ProvisioningException e)
            {
               e.printStackTrace();
            }
         }
    }
}
    //  for each modem in the retrieved list:

    for (int i=0; i<resultList.size(); i++)
    {
    RecordData rd = resultList.get(i);
    Map<String, Object> detailMap = rd.getDetails();

    //get the deviceType from the detail map

    String deviceType = (String)detailMap.get(DeviceDetailsKeys.DEVICE_TYPE);


    //only interest in DOCSIS

    if (DeviceType.getDeviceType(deviceType) .equals(DeviceType.DOCSIS))
    {
        Key primaryKey = rd.getPrimaryKey();
        batch = conn.newBatch();
        batch.unregister((DeviceID)primaryKey);
        batchStatus = null;
        try
        {
            batchStatus = batch.post();
        }
        catch(ProvisioningException e)
        {
            e.printStackTrace();
        }
    }
}
```

b. To delete a device, the service provider's application uses a provisioning client program to delete each of the subscriber's remaining devices individually from the database.

```
// Create a new connection

PACEConnection conn =
    PACEConnectionFactory.getInstance("localhost", 49187, "admin", "password1");


// 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
// NO_PUBLISHING is the publishing mode because we are not attempting
// to publish to external database.

Batch batch = conn.newBatch(

    // No reset
```

```
ActivationMode.NO_ACTIVATION,

// No need to confirm activation

ConfirmationMode.NO_CONFIRMATION,

// No publisining to external database

PublishingMode.NO_PUBLISHING);

batch.getAllForOwnerID("0123-45-6789"   //  query all the devices for this account
                                        // number
);

BatchStatus batchStatus = null;
try
{
    batchStatus = batch.post();
}
catch(Exception e)
{
    e.printStackTrace();
}

CommandStatus commandStatus = batchStatus.getCommandStatus(0);


//batch success without error, retrieve the result

RecordSearchResults rcSearchResult = (RecordSearchResults)commandStatus.getData();
List<RecordData> resultList = rcSearchResult.getRecordData();

if (resultList != null)
{

//  for each modem in the retrieved list, delete it

for (int i=0; i<resultList.size(); i++)
{
    RecordData rd = resultList.get(i);
    Map<String, Object> detailMap = rd.getDetails();

    //get the deviceType from the detail map

    String deviceType = (String)detailMap.get(DeviceDetailsKeys.DEVICE_TYPE);


    //only interest in DOCSIS

    if (DeviceType.getDeviceType(deviceType) .equals(DeviceType.DOCSIS))
    {
         Key primaryKey = rd.getPrimaryKey();

         //change COS

         batch = conn.newBatch();
         batch.delete((DeviceID)primaryKey, true);

         batchStatus = null;
         try
        {
             batchStatus = batch.post();
        }
```

```
                catch(ProvisioningException e)
               {
                    e.printStackTrace();
                }
            }
    }
}
```

Self-Provisioning First-Time Activation in Promiscuous Mode

The 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 to 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 at home.

Step 2 The subscriber powers on the modem, and Prime Cable Provisioning gives it restricted access.

Step 3 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 4 The subscriber uses the service provider's user interface to complete the steps required for registration, including selecting a Class of Service.

The service provider's user interface passes the subscriber's information to Prime Cable Provisioning, including the selected Class of Service and computer IP address. The subscriber's cable modem and computer are then registered with Prime Cable Provisioning.

Step 5 The user interface prompts the subscriber to reboot the computer.

```
// Create a new connection
PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");


Batch batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);


// 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
// modem's MAC address.
// We use the computer's IP address (the web browser
```

```
// received this when the subscriber opened the service
// provider's web interface).
// We also assume that "bostonProvGroup"
// is the provisioning group used in that locality.

List<String> provGroupList = new ArrayList<String>();

provGroupList.add("bostonProvGroup");


batch.getAllForIPAddress("10.0.14.38",

    // ipAddress: restricted access computer lease
    provGroupList
    // provGroups: List containing provgroup
    );

BatchStatus batchStatus = null;

// post the batch to RDU server

try
{
    batchStatus = batch.post();
}
catch(ProvisioningException e)
{
    e.printStackTrace();
}

// Get the LeaseResults object after posting a batch.

CommandStatus commandStatus = batchStatus.getCommandStatus(0);

LeaseResults computerLease = (LeaseResults)commandStatus.getData();

// 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 PCP

StringBuffer modemMACAddress = new StringBuffer();
modemMACAddress.append("1,6,");
modemMACAddress.append(computerLease.getSingleLease().get("relay-agent-remote-id"));


//create MacAddress object from the string

MACAddress modemMACAddressObject = new MACAddress(modemMACAddress.toString());

List<DeviceID> modemDeviceIDList = new ArrayList<DeviceID>();
modemDeviceIDList.add(modemMACAddressObject);


// 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
// NO_PUBLISHING is the publishing mode because we are not attempting
// to publish to external database.

batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,
```

```
    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);

Map<String, Object> properties = new HashMap<String, Object>();

// Set the property PolicyKeys.COMPUTER_PROMISCUOUS_MODE_ENABLED
// to enable promiscuous mode on modem

properties.put(PolicyKeys.COMPUTER_PROMISCUOUS_MODE_ENABLED, Boolean.TRUE);

properties.put(PolicyKeys.COMPUTER_DHCP_CRITERIA, "provisionedCPE");


// enable promiscuous mode by changing the technology default

batch.changeDefaults(DeviceType.DOCSIS,
    properties, null);


// post the batch to RDU server

try
{
    batchStatus = batch.post();
}
catch(ProvisioningException e)
{
    e.printStackTrace();
}
batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);


batch.add(
    DeviceType.DOCSIS,    // deviceType: DOCSIS
    modemDeviceIDList,    // macAddress: derived from computer lease
    null,                 // hostName: not used in this example
    null,                 // domainName: 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                  // properties:
);
```

Step 6 The provisioning client calls *performOperation(…)* to reboot the modem and gives the modem provisioned access.

```
    // Reset the computer
    // create a new batch
    batch = conn.newBatch(

        // No reset

        ActivationMode.AUTOMATIC,

        // No need to confirm activation

        ConfirmationMode.NO_CONFIRMATION);

    // register performOperation command to the batch

    batch.performOperation(DeviceOperation.RESET,
        modemMACAddressObject, null);

    // post the batch to RDU server

    try
    {
        batchStatus = batch.post();
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
}
```

Step 7 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 Prime Cable Provisioning, 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 Provisioning 100 Modems in Promiscuous Mode

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

Desired Outcome

Use this workflow to distribute modem data for all modems to new subscribers. The customer service representative has a list of modems available for assignment.

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

Step 2 Modems that are assigned to a particular kiosk are bulk-loaded into Prime Cable Provisioning and are flagged with the identifier for that kiosk.

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

```
    // Create a new connection
```

```
PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");


Batch batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);



// 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 yet. A configuration can't 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 PolicyKeys.COMPUTER_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(PolicyKeys.COMPUTER_PROMISCUOUS_MODE_ENABLED, Boolean.TRUE);

// The PolicyKeys.CPE_DHCP_CRITERIA is used to specify the DHCP
// Criteria to be used while selecting IP address scopes for
// CPE behind this modem in the promiscuous mode.

properties.put(PolicyKeys.COMPUTER_DHCP_CRITERIA, "provisionedCPE");

// enable promiscuous mode by changing the technology default

batch.changeDefaults(DeviceType.DOCSIS,properties, null);

BatchStatus batchStatus = null;

// post the batch to RDU server

try
{
    batchStatus = batch.post();
}
catch(ProvisioningException e)
{
    e.printStackTrace();
        }

// for each modem MAC-address in list:
```

```
    ModemLoop:
    {
        batch = conn.newBatch(

           // No reset

           ActivationMode.NO_ACTIVATION,

           // No need to confirm activation

           ConfirmationMode.NO_CONFIRMATION,

           // No publishing to external database

           PublishingMode.NO_PUBLISHING);


    batch.add(
        DeviceType.DOCSIS,   // deviceType: DOCSIS
        modemMACAddressList, // modemMACAddressList: the list of deviceID
        null,                // hostName: not used in this example
        null,                // domainName: 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
        properties           // properties:
    );

    try
    {
        batchStatus = batch.post();
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
    // end ModemLoop.
}
```

Preprovisioning First-Time Activation in Promiscuous Mode

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

Desired Outcome

Use this workflow to bring a new unprovisioned cable modem and computer online with 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 its own user interface.

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

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.

// Create a new connection
PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");


Batch batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);


// 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 cannot 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<String, Object> properties = new HashMap<String, Object>();

// Set the property PolicyKeys.COMPUTER_PROMISCUOUS_MODE_ENABLED
// to enable promiscuous mode on modem

properties.put(PolicyKeys.COMPUTER_PROMISCUOUS_MODE_ENABLED, Boolean.TRUE);

properties.put(PolicyKeys.COMPUTER_DHCP_CRITERIA, "provisionedCPE");


// enable promiscuous mode by changing the technology default

batch.changeDefaults(DeviceType.DOCSIS, properties, null);

BatchStatus batchStatus = null;

// post the batch to RDU server

try
{
    batchStatus = batch.post();
```

```
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
    batch = conn.newBatch(

        // No reset

        ActivationMode.NO_ACTIVATION,

        // No need to confirm activation

        ConfirmationMode.NO_CONFIRMATION,

        // No publishing to external database

        PublishingMode.NO_PUBLISHING);


    MACAddress macAddressObject = new MACAddress("1,6,00:11:22:33:44:55");
    List<DeviceID> modemDeviceIDList = new ArrayList<DeviceID>();
    modemDeviceIDList.add(macAddressObject);

    batch.add(
        DeviceType.DOCSIS,      // deviceType: DOCSIS
        modemDeviceIDList,      // macAddress: derived from computer lease
        null,                   // hostName: not used in this example
        null,                   // domainName: 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                 // properties:
    );


    // post the batch to RDU server

    try
    {
        batchStatus = batch.post();
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
}
```

Step 6 The technician powers on the cable modem and Prime Cable Provisioning gives it provisioned access.

Step 7 The technician powers on the computer and Prime Cable Provisioning 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.

Replacing an Existing Modem

A service provider wants to replace a broken modem.

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

Desired Outcome

Use this workflow to physically replace an existing cable 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.

```
    // Create a new connection
    PACEConnection conn = PACEConnectionFactory.getInstance(
        "localhost", 49187, "admin", "password1");

    batch = conn.newBatch(

        // No reset

        ActivationMode.NO_ACTIVATION,

        // No need to confirm activation

        ConfirmationMode.NO_CONFIRMATION,

        // No publishing to external database

        PublishingMode.NO_PUBLISHING);


    MACAddress macAddressObject = new MACAddress("1,6,00:11:22:33:44:55");
    List<DeviceID> modemDeviceIDList = new ArrayList<DeviceID>();
    modemDeviceIDList.add(macAddressObject);

    //  old macAddress: unique identifier for the old modem
    MACAddress oldMacAddress = new MACAddress("1,6,00:11:22:33:44:55");

    // new macAddress: unique identifier for the new modem
    MACAddress newMacAddress = new MACAddress("1,6,00:11:22:33:44:66");
    List<DeviceID> newDeviceIDs = new ArrayList<DeviceID>();
    newDeviceIDs.add(newMacAddress);

    batch.changeDeviceID(oldMacAddress, newDeviceIDs);

    // post the batch to RDU server

    try
    {
        batchStatus = batch.post();
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
}
```

Step 2 The service provider replaces the cable modem and turns it on.

Step 3 The computer must also be turned on.

The cable modem is now a fully provisioned device with the appropriate level of service, as is the computer behind the cable modem.

Adding a Second Computer in Promiscuous Mode

A subscriber wants to connect a second computer behind an installed cable modem. This case does not require calls to the provisioning API.

Desired Outcome

Use this workflow to ensure that the subscriber's selected service permits the connection of multiple sets of CPE, and that the subscriber has network access from both connected computers.

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

Step 2 The subscriber turns on the computer.

If the subscriber's selected service permits connecting multiple sets of CPE, Prime Cable Provisioning gives the second computer access to the Internet.

Self-Provisioning First-Time Activation with NAT

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 to bring a new unprovisioned cable modem (with NAT) and the computers behind it online with 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 Prime Cable Provisioning 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 cable modem 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. For details, see Self-Provisioned Modem and Computer in Fixed Standard Mode, page 7-7.

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 must also be registered similarly to registering a computer.

Adding a New Computer Behind a Modem with NAT

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.

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

Desired Outcome

Use this workflow 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.

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

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

Move Device to Another DHCP Scope

A service provider is renumbering its network causing a registered cable modem to require an IP address from a different Network Registrar scope.

Desired Outcome

A provisioning client changes the DHCP Criteria, and the cable modem receives an IP address from the corresponding DHCP scope.

Step 1 Change the DOCSIS modem's DHCP Criteria to "newmodemCriteria".

```
// Create a new connection
PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");

Batch batch = conn.newBatch(

    // No reset

    ActivationMode.AUTOMATIC,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);

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

```
    batch.changeDHCPCriteria(
        new MACAddress("1,6,ff:00:ee:11:dd:22"),  // Modem's MAC address or FQDN
              "newmodemCriteria"
    );

    // post the batch to RDU server

    BatchStatus batchStatus = null;
    try
    {
        batchStatus = batch.post();
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
}
```

Step 2 The modem gets an IP address from the scope targeted by "newmodemCriteria."

Log Device Deletions Using Events

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
        extends DeviceAdapter

        //Extend the DeviceAdapter class.
{
    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();

        // Modem's MAC address or FQDN "newmodemCriteria"

DeviceEventQualifier qualifier = new DeviceEventQualifier();

// We are interested only in device deletion.
```

```
qualifier.setDeletedDevice ();

// Add device listener using PACEConnection

connection.addDeviceListener(deviceDeletionLogger, qualifier
);
```

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

Monitoring an RDU Connection Using Events

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

Use this workflow to have the event interface notify 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
    extends MessagingAdapter

    //Extend the 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.setConnectionDown();
MessagingNotifier messagingNotifier = new MessagingNotifier();
connnection.addMessagingListener(messagingNotifier, qualifier
);
```

Step 3 If a connection breaks, the event is generated, and the listener is notified. Whenever connectivity is interrupted, the PACE Connection automatically reconnects to the RDU.

Logging Batch Completions Using Events

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

Desired Outcome

When any provisioning client completes 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
    extends BatchAdapter

    //Extend the BatchAdapterclass.
{
    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();
BatchEventQualifier qualifier = new BatchEventQualifier();
connection.addBatchListener(batchCompletionLogger , qualifier
);
```

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

Getting Detailed Device Information

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 the MAC address for the device being queried into the service provider's administrator user interface.

Step 2 Prime Cable Provisioning queries the embedded database for the device details.

```
    // The host name or IP address of the RDU. It is
    // recommended that you normally use a fully-qualified domain name
    // since it lends itself to the greatest flexibility going forward.
    // For example, you could change the host running RDU without
    // having to reassign IPs. For that reason, having an alias for
    // the machine is better than a specific name.

    final String rduHost = "localhost";
```

```
// The port number of RDU on the server.

final int rduPort = 49187;

// The user name for connecting to RDU.

final String userName = "admin";

// The password to use with the username.

final String password = "changeme";

// ------------------------------------------------------------------
// DEVICE PARAMETERS, see IPDevice.getDetails()
// ------------------------------------------------------------------

// The MAC address of the modem to be queried. MAC addresses in BAC
// must follow the simple "1,6,XX:XX:XX:XX:XX:XX" format.

final DeviceID modemMACAddress = DeviceID.getInstance("1,6,00:11:22:33:44:55",
    KeyType.MAC_ADDRESS);

// 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.
// - The maximum number of allowed sessions for the user
// has already been reached.
try
{
connection = PACEConnectionFactory.getInstance(
// RDU host   rduHost,
// RDU port   rduPort,
// User name  userName,
// Password   password
);
}
catch (PACEConnectionException e)
{
// failed to get a connection

System.out.println("Failed to establish a PACEConnection to [" +
    userName + "@" + rduHost + ":" + rduPort + "]; " +
    e.getMessage());

System.exit(1);
}
```

```
// 2) Create 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 publishing to external database

    PublishingMode.NO_PUBLISHING);

// 3) Register the getDetails(...) with the batch.

// Use the Provisioning API to get all of the information for
// the specified MAC address. Since methods aren't actually
// executed until the batch is posted, the results are not
// returned until after post() completes. The getCommandStatus()
// followed by getData() calls must be used to access the results
// once the batch is posted.

final DeviceID modemMACAddress = DeviceID.getInstance("1,6,00:11:22:33:44:55",
        KeyType.MAC_ADDRESS);

List options = new ArrayList();
     options.add(DeviceDetailsOption.INCLUDE_LEASE_INFO);

myBatch.getDetails(modemMACAddress, options);

// 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 bStatus = null;
try
{
    bStatus = myBatch.post();
}
catch (ProvisioningException pe)
{
System.out.println("Failed to query for modem with MAC address [" +
    modemMACAddress + "]; " + pe.getMessage());

System.exit(2);
}
// 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 activation. Command
```

```
// errors occur when a particular method has problems, such as
// trying to add a device that already exists.

if (bStatus.isError())
{
// Batch error occurred.

System.out.println("Failed to query for modem with MAC address [" +
    modemMACAddress + "]; " + bStatus.getErrorMessage());

System.exit(3);
}
```

Step 3 The service provider's application presents a page of device data details, which can display everything that is known about the requested device. If the device was connected to the service provider's network, this data includes lease information (for example, IP address and relay agent identifier). The data indicates whether the device was provisioned, and if it was, the data also includes the device type.

```
// Successfully queried for device.
System.out.println("Queried for DOCSIS modem with MAC address ["+
    modemMACAddress + "]");

// Display the results of the command (TreeMap is sorted). The
// data returned from the batch call is stored on a per-command
// basis. In this example, there is only one command, but if
// you had multiple commands all possibly returning results, you
// could access each result by the index of when it was added.
// The first method added is always index 0. From the status of
// each command, you can then access the accompanying data by
// using the getData() call. Since methods can return data of
// different types, you will have to cast the response to the
// type indicated in the Provisioning API documentation.

Map<String, Object> deviceDetails = new HashMap<String,
    Object>((Map)bStatus.getCommandStatus(0).getData());

String deviceType = (String)deviceDetails.get(DeviceDetailsKeys.DEVICE_TYPE);
String macAddress = (String)deviceDetails.get(DeviceDetailsKeys.MAC_ADDRESS);
String fqdn = (String)deviceDetails.get(DeviceDetailsKeys.FQDN);
String duid = (String)deviceDetails.get(DeviceDetailsKeys.DUID);
String host = (String)deviceDetails.get(DeviceDetailsKeys.HOST);
String domain = (String)deviceDetails.get(DeviceDetailsKeys.DOMAIN);

// if the device is DocsisModem, get the COS

String cos   = (String)deviceDetails.get(DeviceDetailsKeys.CLASS_OF_SERVICE);
String dhcpCriteria   = (String)deviceDetails.get(DeviceDetailsKeys.DHCP_CRITERIA);
String provGroup = (String)deviceDetails.get(DeviceDetailsKeys.PROV_GROUP);
Boolean isProvisioned = (Boolean)deviceDetails.get(DeviceDetailsKeys.IS_PROVISIONED);
String ownerID = (String)deviceDetails.get(DeviceDetailsKeys.OWNER_ID);
Boolean isRegistered = (Boolean)deviceDetails.get(DeviceDetailsKeys.IS_REGISTERED);
String oidNumber = (String)deviceDetails.get(GenericObjectKeys.OID_REVISION_NUMBER);

// if the device is a modem, get the device behind

String relayAgentMacAddress =
    (String)deviceDetails.get(DeviceDetailsKeys.RELAY_AGENT_MAC);
String relayAgentDUID = (String)deviceDetails.get(DeviceDetailsKeys.RELAY_AGENT_DUID);

// get the map of Device property

Map deviceProperties = (Map)deviceDetails.get(DeviceDetailsKeys.PROPERTIES);
```

```
// get the map of discovery data v4

Map dhcpdiscovermapv4 =
    (Map)deviceDetails.get(DeviceDetailsKeys.DISCOVERED_DATA_DHCPV4);

// if discovery data is not null, get the inform, response, request and environment
// map from discovery data map

Map dhcpInformMap = (Map)dhcpdiscovermapv4.get("INFORM");
Map dhcpRespMap = (Map)dhcpdiscovermapv4.get("RESPONSE");
Map dhcpReqMap = (Map)dhcpdiscovermapv4.get("REQUEST");
Map dhcpEnvMap = (Map)dhcpdiscovermapv4.get("ENVIRONMENT");

// get the map of lease query v4

Map leasemapv4 = (Map)deviceDetails.get(DeviceDetailsKeys.LEASE_QUERY_DATA_DHCPV4);
String leaseTime = (String)leasemapv4.get(CNRNames.DHCP_LEASE_TIME.toString());
String rebindingTime =
    (String)leasemapv4.get(CNRNames.DHCP_REBINDING_TIME.toString());

String clientLastTransTime =
    (String)leasemapv4.get(CNRNames.CLIENT_LAST_TRANSACTION_TIME.toString());
String clientIPAddress= (String)leasemapv4.get(CNRNames.CLIENT_IPADDRESS.toString());
String relayAgentRemoteID=
    (String)leasemapv4.get(CNRNames.RELAY_AGENT_REMOTE_ID.toString());
String relayAgentCircuitID=
    (String)leasemapv4.get(CNRNames.RELAY_AGENT_CIRCUIT_ID.toString());

// get the map of discovery DHCP v6

Map dhcpdiscovermapv6 =
    (Map)deviceDetails.get(DeviceDetailsKeys.DISCOVERED_DATA_DHCPV6);

// if discovery data is not null , get the inform, response, request and environment
// map from discovery data map

Map dhcpv6InformMap = (Map)dhcpdiscovermapv6.get("INFORM");

Map dhcpv6RespMap = (Map)dhcpdiscovermapv6.get("RESPONSE");

Map dhcpv6ReqMap = (Map)dhcpdiscovermapv6.get("REQUEST");

Map dhcpv6RelReqMap = (Map)dhcpdiscovermapv6.get("RELAY_REQUEST");

Map dhcpv6EnvMap = (Map)dhcpdiscovermapv6.get("ENVIRONMENT");

// get the map of lease query V6

Map leasemapv6 = (Map)deviceDetails.get(DeviceDetailsKeys.LEASE_QUERY_DATA_DHCPV6);

String iaprefixkey = (String)leasemapv6.get(CNRNames.IAPREFIX.toString());
String iaaddrkey = (String)leasemapv6.get(CNRNames.IAADDR.toString());
String leasetimev6 = (String)leasemapv6.get(CNRNames.VALID_LIFETIME.toString());
String renewaltimev6 = (String)leasemapv6.get(CNRNames.PREFERRED_LIFETIME.toString());
String dhcplasttranstimev6 =
    (String)leasemapv6.get(CNRNames.CLIENT_LAST_TRANSACTION_TIME);
String clientIpAddressv6 = (String)leasemapv6.get(CNRNames.CLIENT_IPADDRESS);
String relayagentremoteidv6 = (String)leasemapv6.get(CNRNames.RELAY_AGENT_REMOTE_ID);
String relayagentcircuitidv6 =
    (String)leasemapv6.get(CNRNames.RELAY_AGENT_CIRCUIT_ID);
```

Searching Using the Device Type

A service provider wants to allow an administrator to view data for all DOCSIS modems.

Desired Outcome

The service provider's administrative application returns a list of DOCSIS devices.

Step 1 The administrator selects the search option in the service provider's administrator user interface.

Step 2 Prime Cable Provisioning queries the embedded database for a list of all MAC addresses for the DOCSIS modems.

```
public static void getAllDevicesByDeviceType() throws Exception  {
      DeviceSearchType dst = DeviceSearchType.getByDeviceType(
          DeviceType.getDeviceType(DeviceTypeValues.DOCSIS_MODEM),
          ReturnParameters.ALL);

      RecordSearchResults rs = null;

      SearchBookmark sb = null;

      rs = searchDevice(dst, sb);
      sb = rs.getSearchBookmark();

      while (sb != null)
      {
          // print out the data in the record search result.
          sb = printRecordSearchResults(rs);

          // call the search routine again
          rs = searchDevice(dst, sb);
      }
 }
 private static RecordSearchResults searchDevice(DeviceSearchType dst,
                SearchBookmark sb) throws Exception  {
      RecordSearchResults rs = null;
      final Batch batch = s_conn.newBatch();
      final int numberOfRecordReturn = 10;

      //calling the search API
      batch.searchDevice(dst, sb, numberOfRecordReturn);

      // Call the RDU.
      BatchStatus batchStatus = batch.post();

      // Check for success.
      CommandStatus commandStatus = null;

      if (0 < batchStatus.getCommandCount())
      {
          commandStatus = batchStatus.getCommandStatus(0);
      }
      //check to see if there is an error
      if (batchStatus.isError()
      || batchStatus.isWarning()
      || commandStatus == null
      || commandStatus.isError())
      {
          System.out.println("report batch error.");
          return null;
      }
```

```
        //batch success without error, retrieve the result
        //this is a list of devices
        rs = (RecordSearchResults)commandStatus.getData();
        return rs;
    }


    private static SearchBookmark printRecordSearchResults(RecordSearchResults rs)  throws
Exception  {

        SearchBookmark sb = rs.getSearchBookmark();

        List<RecordData> rdlist = rs.getRecordData();
        Iterator<RecordData> iter = rdlist.iterator();

        while (iter.hasNext())
        {
            RecordData rdObj = iter.next();
            Key keyObj = rdObj.getPrimaryKey();

             System.out.println("DeviceOID: " + ((DeviceID)keyObj).getDeviceId());

            //this is for secondary keys.
            List<Key> deviceList = rdObj.getSecondaryKeys();

            if (deviceList != null && !deviceList.isEmpty())
            {
                for (int i=0; i<deviceList.size(); i++)
                {
                    Key key = deviceList.get(i);
                    System.out.println("DeviceID : " + key.toString());
                }
            }
        }
        return sb;

    }
```

Searching for Devices Using Vendor Prefix or Class of Service

A service provider wants to allow an administrator to search for all devices matching a particular vendor prefix or a particular Class of Service.

Desired Outcome

The service provider's administrative application returns a list of devices matching the requested vendor prefix or the Class of Service.

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

Step 2 Prime Cable Provisioning queries the embedded database for a list of all MAC addresses for the devices that match the requested vendor prefix or Class of Service. This example illustrates how you can build the search query to retrieve devices using the MAC address. Also see Searching Using the Device Type, page 7-40.

```
DeviceIDPattern pattern = new MACAddressPattern("1,6,22:49:*");
```

```
DeviceSearchType dst = DeviceSearchType.getDevices(pattern, ReturnParameters.ALL);

// To set up search for class of service:

DeviceSearchType searchType = DeviceSearchType.getByClassOfService(

new ClassOfServiceName(name), AssociationType

.valueOf(association), ReturnParameters.ALL);
```

Step 3 The service provider's application requests details on these devices from Prime Cable Provisioning, and presents a page of device data. For each device, the code displays the device type, MAC address, client class, and provisioned status of the device. One device is identified per line.

```
// calling the search procedure

rs = searchDevice(connection, dst, sb);
sb = processRecordSearchResults(rs);

if (rs != null)
{
    while (sb != null)
         {
            // The search returns a search bookmark, which can be used to make
            // the next search call that would return next set of results

            rs = searchDevice(connection, dst, sb);
            sb = processRecordSearchResults(rs);
        }
    }
}
}
```

Preprovisioning PacketCable eMTA/eDVA

A new customer contacts a service provider to order PacketCable voice service. The customer expects to receive a provisioned embedded MTA or embedded DVA.

Desired Outcome

Use this workflow to preprovision an embedded MTA or embedded DVA so that the modem MTA/DVA component has the appropriate level of service when brought online.

Note This use case skips the call agent provisioning that is required for making telephone calls from eMTA/eDVAs.

Step 1 Choose a subscriber username and password for the billing system.

Step 2 Choose the appropriate Class of Service and DHCP Criteria for the modem component and add it to Prime Cable Provisioning.

```
// Create a new connection

PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");
```

```
Batch batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION);

// Let's provision the modem and the MTA component 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.
// To add a DOCSIS modem:

List<DeviceID> modemDeviceIDList = new ArrayList<DeviceID>();
modemDeviceIDList.add(new MACAddress("1,6,01:02:03:04:05:06"));
batch.add(
    DeviceType.DOCSIS,     // deviceType: DOCSIS
    modemDeviceIDList,     // macAddress: scanned from the label
    null,                  // hostName: not used in this example
    null,                  // domainName: 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                   // properties: not used
);
```

Step 3 Choose the appropriate Class of Service and DHCP Criteria for the MTA/DVA component and add it to Prime Cable Provisioning.

```
List<DeviceID> packetcableMTADeviceIDList = new ArrayList<DeviceID>();
packetcableMTADeviceIDList.add(new MACAddress("1,6,01:02:03:04:05:07"));

// Continuation of the batch in Step2
// To add the MTA component:

batch.add(
    DeviceType.PACKET_CABLE_MTA,  // deviceType: PACKET_CABLE_MTA
    packetcableMTADeviceIDList,   // macAddress: scanned from the label
    null,                         // hostName: not used in this example, will be auto
                                  // generated
    null,                         // domainName: not used in this example, will be
                                  // auto generated. The FqdnKeys.AUTO_FQDN_DOMAIN
                                  // property must be set somewhere in the property
                                  // hierarchy.
    "0123-45-6789",               // ownerID: here, account number from billing system
    "Silver",                     // ClassOfService
    "provisionedMTA",             // DHCP Criteria: Network Registrar uses this to
                                  // select an MTA lease granting provisioned IP
                                  // address
    null                          // properties: not used
);

BatchStatus batchStatus = null;

// post the batch to RDU server

try
{
    batchStatus = batch.post();
```

```
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
}
```

Step 4 The embedded MTA or embedded DVA is shipped to the customer.

Step 5 The customer brings the embedded MTA or embedded DVA online and makes telephone calls using it.

SNMP Cloning on PacketCable eMTA/eDVA

An administrator wants to grant SNMP Element Manager access to a PacketCable eMTA/eDVA.

Desired Outcome

An external Element Manager is granted secure SNMPv3 access to the PacketCable eMTA/eDVA.

Note Changes made to RW MIB variables are not permanent and are not updated in the Prime Cable Provisioning configuration for the eMTA/eDVA. The information written into the eMTA/eDVA MIB is lost the next time the MTA powers down or resets.

Step 1 Call the provisioning API method, *performOperation(...)*, passing in the MAC address of the MTA/DVA and the username of the new user to create on the MTA/DVA. This will be the username used in subsequent SNMP calls by the Element Manager.

```
// Create a new connection
PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");

Batch batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION);

// NO_ACTIVATION is the activation mode because we don't want to
// reset the device.
// NO_CONFIRMATION is the confirmation mode because we are
// not attempting to reset the device.
// The goal here is to create a new user on the MTA indicated
// by the MAC address. The other parameter needed here is the new
// user name, which is passed in the Map.
// Create a map that contains one element - the name of
// the new user to be created on the MTA

HashMap<String, Object> map = new HashMap<String, Object>();
map.put( SNMPPropertyKeys.CLONING_USERNAME, "newUser" );

// The first param is the actual device operation to perform.

batch.performOperation(
```

```
        DeviceOperation.ENABLE_SNMPV3_ACCESS,    // deviceOperation : ENABLE_SNMPV3_ACCESS
        new MACAddress("1,6,00:00:00:00:00:99"), // macORFqdn : MAC Address of the modem
        map                                      // parameters: operation specific
                                                 // parameters
    );

    BatchStatus batchStatus = null;

    // post the batch to RDU server

    try
    {
        batchStatus = batch.post();
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
}
```

Step 2 The provisioning API attempts to perform an SNMPv3 cloning operation to create an entry on the MTA/DVA for the new user passed in Step 1. The keys used in the new user entry row are a function of two passwords defined within Prime Cable Provisioning. These passwords will be made available to the customer and the RDU command passes these passwords (the auth and priv password) through a key localization algorithm to create an auth and priv key. These are stored, along with the new user, in the eMTA's/eDVA's user table.

Note The auth and priv passwords mentioned in this step may be changed by setting `SNMPPropertyKeys.CLONING_AUTH_PASSWORD` (*/snmp/cloning/auth/password*) and `SNMPPropertyKeys.CLONING_PRIV_PASSWORD` (*/snmp/cloning/priv/password*) properties, respectively, in the *rdu.properties* configuration file.

Step 3 The customer issues SNMPv3 requests using the specified username, passwords, and key localization algorithm to allow for secure communication with the MTA/DVA.

Incremental Provisioning of PacketCable eMTA/eDVA

A customer has a PacketCable eMTA/eDVA in service with its first line (endpoint) enabled. The customer wants to enable the second telephone line (endpoint) on the eMTA/eDVA and connect a telephone to it.

Desired Outcome

The customer should be able to connect a telephone to the second line (endpoint) on the eMTA/eDVA and successfully make phone calls from it without any service interruption.

Note In order to use the second line on the eMTA/eDVA, the Call Agent needs to be configured accordingly. This use case does not address provisioning of call agents.

Step 1 The service provider's application invokes the Prime Cable Provisioning API to change the Class of Service of the eMTA/eDVA. The new Class of Service supports two endpoints on the eMTA/eDVA. This change in Class of Service does not take effect until the eMTA/eDVA is reset. Disrupting the eMTA/eDVA is not desirable; therefore, incremental provisioning is undertaken in the next step.

```
PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");

Batch batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION);

// NO_ACTIVATION is the activation mode because we don't want to
// reset the device.
// NO_CONFIRMATION is the Confirmation mode because we are not
// disrupting the device.

batch.changeClassOfService(
    new MACAddress("1,6,ff:00:ee:11:dd:22"),  / eMTA's MAC address or FQDN
    "twoLineEnabledCOS"                      // This COS supports two lines.
);
BatchStatus batchStatus = null;

// post the batch to RDU server

try
{
    batchStatus = batch.post();
}
catch(ProvisioningException e)
{
    e.printStackTrace();
    }
```

Step 2 The service provider's application uses the Prime Cable Provisioning incremental update feature to set SNMP objects on the eMTA/eDVA and thereby enabling the service without disrupting the eMTA/eDVA.

```
// The goal here is to enable a second phone line, assuming one
// phone line is currently enabled.   We will be adding a new
// row to the pktcNcsEndPntConfigTable.

batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION);

// NO_ACTIVATION is the activation mode because we don't want to
// reset the device.
// NO_CONFIRMATION is the confirmation mode because we are
// not attempting to reset the device.
// Create a map containing one element - the list of SNMP
// variables to set on the MTA
```

```
    HashMap<String, Object> map = new HashMap<String, Object>();

    // Create an SnmpVarList to hold SNMP varbinds

    SnmpVarList list = new SnmpVarList();

    // An SnmpVariable represents an oid/value/type triple.
    // pktcNcsEndPntConfigTable is indexed by the IfNumber, which in this case we will
    // assume is interface number 12 (this is the last number in each of the oids below).
    // The first variable represents the creation of a new row in
    // pktcNcsEndPntConfigTable we are setting the RowStatus
    // column (column number 26). The value of 4 indicates that
    // a new row is to be created in the active state.

    SnmpVariable variable = new SnmpVariable( ".1.3.6.1.4.1.4491.2.2.2.1.2.1.1.26.12",
        "4", SnmpType.INTEGER );
    list.add( variable );

    // The next variable represents the call agent id for this new
    // interface, which we'll assume is 'test.com'

    variable = new SnmpVariable( ".1.3.6.1.4.1.4491.2.2.2.1.2.1.1.1.12", "test.com",
        SnmpType.STRING );
    list.add( variable );

    // The final variable represents the call agent port

    variable = new SnmpVariable( ".1.3.6.1.4.1.4491.2.2.2.1.2.1.1.2.12", "2728",
        SnmpType.INTEGER );
    list.add( variable );

    // Add the SNMP variable list to the Map to use in the API call

    map.put( SNMPPropertyKeys.SNMPVAR_LIST, list );

    // Invoke the PCP API to do incremental update on the eMTA.

    batch.performOperation(
        DeviceOperation.INCREMENTAL_UPDATE,      // device operation
        new MACAddress("1,6,00:00:00:00:00:99"), // MAC Address
        map                                      // Parameters for the operation
    );

    // post the batch to RDU server

    try
    {
        batchStatus = batch.post();
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
}
```

Step 3 The eMTA/eDVA is enabled to use the second telephone line. The eMTA/eDVA continues to receive the same service, after being reset, because the Class of Service was changed in Step 1.

Preprovisioning DOCSIS Modems with Dynamic Configuration Files

A new customer contacts a service provider to order a DOCSIS modem with high-speed *Gold* data service for two sets of CPE behind it.

Desired Outcome

Use this workflow to preprovision a DOCSIS modem with a Class of Service that uses DOCSIS templates. The dynamic configuration file generated from the templates is used while the modem comes online.

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

Step 2 The service provider chooses Gold Class of Service, and the appropriate DHCP Criteria, and then adds the cable modem to Prime Cable Provisioning.

```
PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");

Batch batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION);


Map<String, Object> properties = new HashMap<String, Object>();

// Set the property PolicyKeys.COMPUTER_PROMISCUOUS_MODE_ENABLED to enable
// promiscuous mode on modem

properties.put(PolicyKeys.COMPUTER_PROMISCUOUS_MODE_ENABLED, Boolean.TRUE);

// enable promiscuous mode by changing the technology default

batch.changeDefaults(DeviceType.DOCSIS,properties, null);

BatchStatus batchStatus = null;

// post the batch to RDU server

try
{
    batchStatus = batch.post();
}
catch(ProvisioningException e)
{
    e.printStackTrace();
}
// No CPE DHCP Criteria is specified.
// The CPE behind the modem will use the default provisioned
// promiscuous CPE DHCP criteria specified in the system defaults.
// This custom property corresponds to a macro variable in the
// DOCSIS template for "gold" class of service indicating the
// maximum number of CPE allowed behind this modem. We set it
// to two sets of CPE from this customer.

properties = new HashMap<String, Object>();
```

```
    properties.put("docsis-max-cpes", "2");

    batch = conn.newBatch(

        // No reset

        ActivationMode.NO_ACTIVATION,

        // No need to confirm activation

        ConfirmationMode.NO_CONFIRMATION,

        // No publishing to external database

        PublishingMode.NO_PUBLISHING);

    // To add a DOCSIS modem:

    List<DeviceID> deviceIDList = new ArrayList<DeviceID>();
    deviceIDList.add(new MACAddress("1,6,01:02:03:04:05:06"));
    batch.add(
        DeviceType.DOCSIS,     // deviceType: DOCSIS
        deviceIDList,          // macAddress: scanned from the label
        null,                  // hostName: not used in this example
        null,                  // domainName: not used in this example
        "0123-45-6789",        // ownerID: here, account number from billing system
        "gold",                // classOfService:
        "provisionedCM",       // DHCP Criteria: Network Registrar uses this to
                               // select a modem lease granting provisioned IP address
        properties             // properties:
    );

    try
    {
        batchStatus = batch.post();
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
}
```

Step 3 The cable modem is shipped to the customer.

Step 4 The customer brings the cable modem online and connects the computers behind it.

Optimistic Locking

An instance of the service provider application needs to ensure that it is not overwriting the changes made by another instance of the same application.

Desired Outcome

Use this workflow to demonstrate the optimistic locking capabilities provided by the Prime Cable Provisioning API.

Note Locking of objects is done in multiuser systems to preserve integrity of changes, so that one person's changes do not accidentally get overwritten by another. With optimistic locking, you write your program assuming that any commit has a chance to fail if at least one of the objects being committed was changed by someone else since you began the transaction.

Step 1 The service representative selects the search option in the service provider's user interface and enters the cable modem's MAC address.

Step 2 Prime Cable Provisioning queries the embedded database, gets the details of the device, and the MSO user interface displays the information.

```
PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");

Batch batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);


final DeviceID modemMACAddress = DeviceID.getInstance("1,6,00:11:22:33:44:55",
        KeyType.MAC_ADDRESS);
List<DeviceDetailsOption> options = new ArrayList<DeviceDetailsOption>();

options.add(DeviceDetailsOption.INCLUDE_LEASE_INFO);

// 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.

batch.getDetails(modemMACAddress, options);

// post the batch to RDU server

BatchStatus batchStatus = null;

try
{
    batchStatus = batch.post();
}
catch(ProvisioningException e)
{
    e.printStackTrace();
}
```

Step 3 The service representative attempts to change the Class of Service and the DHCP Criteria of the modem using the user interface. This in turn invokes the Prime Cable Provisioning API.

```
Map<String, Object> deviceDetails = new TreeMap((Map<String,
    Object>)batchStatus.getCommandStatus(0).getData());
```

```
    //  extract device detail data from the map

    String deviceType = (String)deviceDetails.get(DeviceDetailsKeys.DEVICE_TYPE);
    String macAddress = (String)deviceDetails.get(DeviceDetailsKeys.MAC_ADDRESS);
    String relayAgentID = (String)deviceDetails.get(DeviceDetailsKeys.RELAY_AGENT_MAC);
    Boolean isProvisioned = (Boolean)deviceDetails.get(DeviceDetailsKeys.IS_PROVISIONED);

    //  Let's save the OID_REVISION_NUMBER property so that we can set it in
    //  step 3.

    String oidRevisionNumber =
        (String)deviceDetails.get(GenericObjectKeys.OID_REVISION_NUMBER);

    // We need a reference to Batch instance so that ensureConsistency()
    // method can be invoked on it.

    batch = conn.newBatch() ;
    List<String> oidList = new ArrayList<String>();

    // Add the oid-rev number saved from step 2 to the list

    oidList.add(oidRevisionNumber);

    // Sends a list of OID revision numbers to validate before processing the
    // batch. This ensures that the objects specified have not been modified
    // since they were last retrieved.

    batch.ensureConsistency(oidList);
    batch.changeClassOfService (
        new MACAddress("1,6,00:11:22:33:44:55"),  // macORFqdn: unique identifier for the
                                                  // device.
        "gold"                                    // newCOSName : Class of service name.
    );

    batch.changeDHCPCriteria (
        new MACAddress("1,6,00:11:22:33:44:55"),  // macORFqdn: unique identifier for the
                                                  // device.
        "specialDHCPCriteria"                     // newDHCPCriteria : New DHCP Criteria.
    );

    // This batch fails with BatchStatusCodes.BATCH_NOT_CONSISTENT,
    // in case if the device is updated by another client in the meantime.
    // If a conflict occurs, then the service provider client
    // is responsible for resolving the conflict by querying the database
    // again and then applying changes appropriately.
    }
}
```

Step 4 The user is ready to receive Gold Class of Service with appropriate DHCP Criteria.

Temporarily Throttling a Subscriber's Bandwidth

An MSO has a service that allows a subscriber to download only 10 MB of data a month. Once the subscriber reaches that limit, their downstream bandwidth is turned down from 10 MB to 56 K. When the month is over they are moved back up to 10 MB.

Note You may want to consider changing upstream bandwidth as well, because peer-to-peer users and users who run websites tend to have heavy upload bandwidth.

Desired Outcome

Use this workflow to move subscribers up and down in bandwidth according to their terms of agreement.

Step 1 The MSO has a rate tracking system, such as *NetFlow*, which keeps track of each customer's usage by MAC address. Initially a customer is provisioned at the *Gold* Class of Service level with 1 MB downstream.

Step 2 When the rate tracking software determines that a subscriber has reached the 10-MB limit it notifies the OSS. The OSS then makes a call into the Prime Cable Provisioning API to change the subscriber's Class of Service from *Gold* to *Gold-throttled*.

```
PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");

Batch batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);


// AUTOMATIC is the activation mode because we are
// attempting to reset the modem so that it
// receives low bandwidth service.
// 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, then it will
// be disabled when it is turned back on.
// Let's change the COS of the device so that it restricts
// bandwidth usage of the modem.

batch.changeClassOfService(
    new MACAddress("1,6,00:11:22:33:44:55"),  // macAddress: unique identifier for
                                              // this modem
    "Gold-throttled"                          // newClassOfService: restricts
                                              // bandwidth usage to 56k
);

BatchStatus batchStatus = null;

// post the batch to RDU server

try
{
    batchStatus = batch.post();
}
catch(ProvisioningException e)
{
```

```
            e.printStackTrace();
        }
    }
```

Step 3 At the end of the billing period, the OSS calls the Prime Cable Provisioning API to change the subscriber's Class of Service back to *Gold*.

Preprovisioning CableHome WAN-MAN

A new customer contacts a service provider to order home networking service. The customer expects a provisioned CableHome device.

Desired Outcome

Use this workflow to preprovision a CableHome device so that the cable modem and WAN-MAN components on it will have the appropriate level of service when brought online.

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

Step 2 The service provider chooses the appropriate Class of Service and the DHCP Criteria for the modem component, then adds it to Prime Cable Provisioning.

```
PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");

Batch batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation
    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);

// Let's provision the modem and the WAN-Man component in the same
// batch.
// To add a DOCSIS modem:

List<DeviceID> docisDeviceIDList = new ArrayList<DeviceID>();
docisDeviceIDList.add(new MACAddress("1,6,01:02:03:04:05:06"));
batch.add(
    DeviceType.DOCSIS,      // deviceType: DOCSIS
    docisDeviceIDList,      // macAddress: scanned from the label
    null,                   // hostName: not used in this example
    null,                   // domainName: 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                    // properties: not used
);
```

Step 3 The service provider chooses the appropriate Class of Service and DHCP Criteria for the WAN-MAN component, then adds it to Prime Cable Provisioning.

```
    List<DeviceID> wanManDeviceIDList = new ArrayList<DeviceID>();
    wanManDeviceIDList.add(new MACAddress("1,6,01:02:03:04:05:07"));
    batch.add(
        DeviceType.CABLEHOME_WAN_MAN,   // deviceType: CABLEHOME_WAN_MAN
        wanManDeviceIDList,             // macAddress: scanned from the label
        null,                           // hostName: not used in this example
        null,                           // domainName: not used in this example
        "0123-45-6789",                 // ownerID: here, account number from billing
                                        // system
        "silverWanMan",                 // classOfService
        "provisionedWanMan",            // DHCP Criteria: Network Registrar uses this to
                                        // select a modem lease granting provisioned IP
                                        // address
        null                            // properties: not used
    );
}
```

Step 4 The CableHome device is shipped to the customer.

Step 5 The customer brings the CableHome device online.

CableHome with Firewall Configuration

A customer contacts a service provider to order a home networking service with the firewall feature enabled. The customer expects to receive a provisioned CableHome device.

Desired Outcome

Use this workflow to preprovision a CableHome device so that the cable modem and the WAN-MAN components on it have the appropriate level of service when brought online.

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

Step 2 The service provider chooses the appropriate Class of Service and DHCP Criteria for the cable modem component, then adds it to Prime Cable Provisioning.

```
PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");

Batch batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);

// Let's provision the modem and the WAN-Man component in the same
// batch.
// To add a DOCSIS modem:

List<DeviceID> docisDeviceIDList = new ArrayList<DeviceID>();
docisDeviceIDList.add(new MACAddress("1,6,01:02:03:04:05:06"));
```

```
    batch.add(
        DeviceType.DOCSIS,      // deviceType: DOCSIS
        docisDeviceIDList,      // macAddress: scanned from the label
        null,                   // hostName: not used in this example
        null,                   // domainName: 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                    // properties: not used
    );
```

Step 3 The service provider chooses the appropriate Class of Service and DHCP Criteria for the WAN-MAN component and adds it to Prime Cable Provisioning.

```
    //      Continuation of the batch in Step 2
    // To add the WAN-Man component:
    // Create a Map to contain WanMan's properties

    Map<String, Object> properties = new HashMap<String, Object>();

    // The fire wall configuration for the Wan Man component is specified
    // using the CableHomeKeys.CABLEHOME_WAN_MAN_FIREWALL_FILE property.
    // This use case assumes that the firewall configuration file named
    // "firewall_file.cfg" is already present in the RDU database and the
    // firewall configuration is enabled in the Wan Man configuration file
    // specified with the corresponding class of service.

    properties.put(CableHomeKeys.CABLEHOME_WAN_MAN_FIREWALL_FILE, "firewall_file.cfg");

    List<DeviceID> wanManDeviceIDList = new ArrayList<DeviceID>();
    wanManDeviceIDList.add(new MACAddress("1,6,01:02:03:04:05:07"));
    batch.add(
        DeviceType.CABLEHOME_WAN_MAN,    // deviceType: CABLEHOME_WAN_MAN
        wanManDeviceIDList,              // macAddress: scanned from the label
        null,                         // hostName: not used in this example
        null,                         // domainName: not used in this example
        "0123-45-6789",               // ownerID: here, account number from billing system
        "silverWanMan",               // classOfService
        "provisionedWanMan",          // DHCP Criteria: Network Registrar uses this to
                                      // select a modem lease granting provisioned IP
                                      // address
        null                          // properties: not used
    );

    BatchStatus batchStatus = null;

    // post the batch to RDU server

    try
    {
        batchStatus = batch.post();
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
}
```

Step 4 The CableHome device is shipped to the customer.

Step 5 The customer brings the CableHome device online and the cable modem and the WAN-MAN component get provisioned IP addresses and proper configuration files.

Retrieving Device Capabilities for CableHome WAN-MAN

A service provider wants to allow an administrator to view capabilities information for a CableHome WAN-MAN device.

Desired Outcome

The service provider's administrative application displays all known details about a given CableHome WAN-MAN component, including MAC address, lease information, provisioned status, and the device capabilities information.

Step 1 The administrator enters the MAC address of the WAN-MAN being queried into the service provider's user interface.

Step 2 Prime Cable Provisioning queries the embedded database for details of the device identified using the MAC address entered.

```
PACEConnection conn = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "password1");


Batch batch = conn.newBatch(

    // No reset

    ActivationMode.NO_ACTIVATION,

    // No need to confirm activation

    ConfirmationMode.NO_CONFIRMATION,

    // No publishing to external database

    PublishingMode.NO_PUBLISHING);


final DeviceID modemMACAddress = DeviceID.getInstance("1,6,00:11:22:33:44:55",
        KeyType.MAC_ADDRESS);

List<DeviceDetailsOption> options = new ArrayList<DeviceDetailsOption>();
options.add(DeviceDetailsOption.INCLUDE_LEASE_INFO);

// 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.

batch.getDetails(modemMACAddress, options);

// post the batch to RDU server

BatchStatus batchStatus = null;
try
{
    batchStatus = batch.post();
}
```

```
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
}
```

Step 3 The service provider's application then presents a page of device data details, which can display everything that is known about the requested device. If the device was connected to the service provider's network, this data includes lease information, such as the IP address or the relay agent identifier. This data indicates whether the device is provisioned. If it is provisioned, the data also includes the device type and device capabilities information.

```
    Map<String, Object> deviceDetails = new TreeMap((Map<String,
        Object>)batchStatus.getCommandStatus(0).getData());

    //  extract device detail data from the map

    String deviceType = (String)deviceDetails.get(DeviceDetailsKeys.DEVICE_TYPE);
    String macAddress = (String)deviceDetails.get(DeviceDetailsKeys.MAC_ADDRESS);
    String relayAgentID = (String)deviceDetails.get(DeviceDetailsKeys.RELAY_AGENT_MAC);
    Boolean isProvisioned = (Boolean)deviceDetails.get(DeviceDetailsKeys.IS_PROVISIONED);

    String deviceID = (String) deviceDetails.get(CNRNames.DEVICE_ID.toString());
    String serNum = (String) deviceDetails.get(CNRNames.DEVICE_SERIAL_NUMBER.toString());
    String hwVer = (String)
        deviceDetails.get(CNRNames.HARDWARE_VERSION_NUMBER.toString());
    String swVer = (String)
        deviceDetails.get(CNRNames.SOFTWARE_VERSION_NUMBER.toString());
    String brVer = (String) deviceDetails.get(CNRNames.BOOT_ROM_VERSION.toString());
    String vendorOui = (String) deviceDetails.get(CNRNames.VENDOR_OUI.toString());
    String modelNum = (String) deviceDetails.get(CNRNames.MODEL_NUMBER.toString());
    String vendorNum = (String) deviceDetails.get(CNRNames.VENDOR_NAME.toString());

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

Self-Provisioning CableHome WAN-MAN

A subscriber has a computer with a browser application installed in a single-dwelling unit and has purchased an embedded CableHome device.

Desired Outcome

Use this workflow to bring a new unprovisioned embedded CableHome device online with the appropriate level of service, and give the subscriber internet access from computers connected to the embedded CableHome device.

Step 1 The subscriber purchases an embedded CableHome device and installs it at home.

Step 2 The subscriber powers on the embedded CableHome device. Prime Cable Provisioning gives the embedded cable modem restricted access, allowing two sets of CPE: one for the CableHome WAN-MAN and the other for the computer.

Note This use case assumes an unprovisioned DOCSIS modem allows two sets of CPE behind it. Until configured to do otherwise, Prime Cable Provisioning supports only a single device behind an unprovisioned DOCSIS modem. You can change this behavior by defining an appropriate Class of Service that supports two sets of CPE and then using it as the default Class of Service for DOCSIS devices.

Step 3 Prime Cable Provisioning configures the CableHome WAN-MAN, including IP connectivity and downloading the default CableHome boot file. The default CableHome boot file configures the CableHome device in passthrough mode. The CableHome device is still unprovisioned.

Step 4 The subscriber connects the computer to the CableHome device. The computer gets an unprovisioned (restricted) IP address. The subscriber starts a browser application on the computer. 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 cable modem registration, including selecting a Class of Service. The subscriber also selects a CableHome Class of Service.

Step 6 The service provider's user interface passes the subscriber's information to Prime Cable Provisioning, including the selected Class of Service for cable modem and CableHome, and computer IP address. The subscriber is then registered with Prime Cable Provisioning.

Step 7 The user interface prompts the subscriber to reboot the computer.

Step 8 The provisioning client calls *performOperation(…)* to reboot the modem and gives the modem provisioned access.

```
    // create a new batch

    batch = conn.newBatch(

        // No reset

        ActivationMode.AUTOMATIC,

        // No need to confirm activation

        ConfirmationMode.NO_CONFIRMATION);


    // register performOperation command to the batch

    batch.performOperation(DeviceOperation.RESET, modemMACAddressObject, null);

    // post the batch to RDU server

    try
    {
        batchStatus = batch.post();
    }
    catch(ProvisioningException e)
    {
        e.printStackTrace();
    }
}
```

Step 9 When the computer is rebooted, it receives a new IP address from the CableHome device's DHCP server. The cable modem and the CableHome device are now both provisioned. Now the subscriber can connect a number of computers to the Ethernet ports of the CableHome device and they have access to the Internet.

Note If the configuration file supplied to the WAN-MAN component enables the WAN-Data component on the box, it will be provisioned in the promiscuous mode. This assumes that the promiscuous mode is enabled at the technology defaults level for the DeviceType.CABLEHOME_WAN_DATA device type.

RBAC Administration and Operational Access Control

Roles and privileges provide fine grain authorization to manage Prime Cable Provisioning. A privilege represents an authority granted for an operation that can be performed and roles are composed of these privileges.

This section includes these use cases:

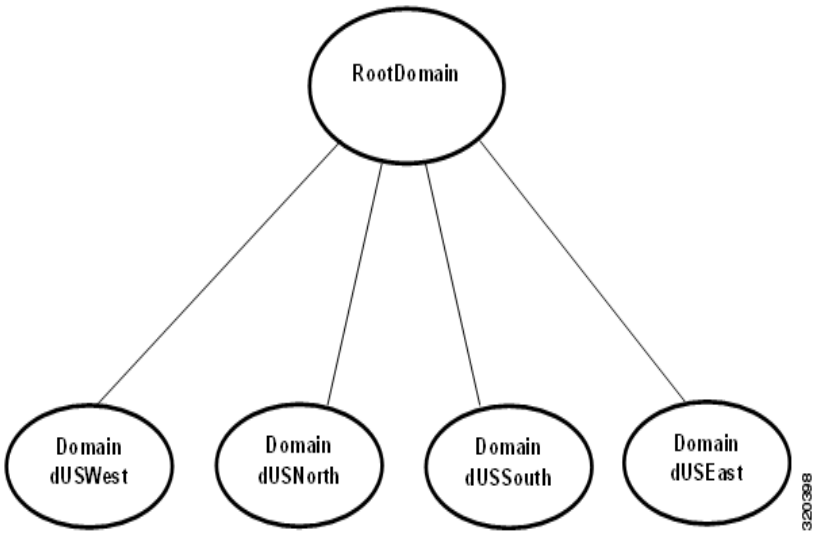
- Role Based Access Control Administration, page 7-59
- Operational Access Control, page 7-63

Role Based Access Control Administration

An administrator wants to use role based access control (RBAC) in the RDU server. The administrator has to first create various configurations.

Figure 7-2 represents the domain hierarchy for RBAC.

Figure 7-2 RBAC Domain Hierarchy



Desired Outcome

Use this work flow to create various configurations required for Role Based Access Control (Domain, Role, User Group, User).

Step 1 Create a domain dUS under the out of the box domain RootDomain. RootDomain is the top most domain defined in the RDU server.

```
//Create a new connection
PACEConnection connection = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "changeit");
Batch batch = connection.newBatch();
//Create a new domain dUS under RootDomain
bsatch.addDomain("dUS", "US", DomainTypeValues.ROOT_DOMAIN, null);
BatchStatus bStatus = null;
//Post the batch to RDU Server
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

Step 2 Create two other domains dUSEast and dUSWest with dUS as their parent.

```
//Create a new batch

batch = connection.newBatch();

//Create two child domains dUSEast and dUSWest under the parent domain dUS

batch.addDomain("dUSEast", "US East", "dUS", null);
batch.addDomain("dUSWest", "US West", "dUS", null);
bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

Step 3 Create a role rAllDevice with all device related privileges associated with it. Also, associate different properties which can be modified by a user with this role.

```
//Create a new batch

batch = connection.newBatch();

//Add privileges to the role

List <String> allDevicePrivileges=new ArrayList<String>();
allDevicePrivileges.add(PrivilegeTypeValues.PRIV_DEVICE_CREATE);
allDevicePrivileges.add(PrivilegeTypeValues.PRIV_DEVICE_DELETE);
allDevicePrivileges.add(PrivilegeTypeValues.PRIV_DEVICE_OPERATION);
allDevicePrivileges.add(PrivilegeTypeValues.PRIV_DEVICE_READ);
allDevicePrivileges.add(PrivilegeTypeValues.PRIV_DEVICE_REGEN);
allDevicePrivileges.add(PrivilegeTypeValues.PRIV_DEVICE_UPDATE);

//Add properties to the role
```

```
Map<String, Object> properties = new HashMap<String, Object>();

//Add device related properties which can be administered by the role.

List<String> devicesProperties = new ArrayList<String>();

devicesProperties.add(FqdnKeys.AUTO_FQDN_ENABLE);

devicesProperties.add(FqdnKeys.AUTO_FQDN_DOMAIN);

properties.put(RoleDetailsKeys.MODIFIABLE_DEVICE_PROPERTIES,
     devicesProperties);

//Create the role

batch.addRole("rAllDevice", "Role with all device privileges",
    allDevicePrivileges, properties);

bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

Step 4 Create a new user group gDeviceAdmin and associate it with the newly created role rAllDevice and other out of the box roles (for example, File Admin Role) defined in the RDU server.

```
//Create a new batch

batch = connection.newBatch();

List<String> roleList = new ArrayList<String>();

//Associate the role "rAllDevice" to the user group

roleList.add("rAllDevice");

//Associate the out of the box roles already defined in the system to the user group

roleList.add(RoleTypeValues.FILE_ADMIN_ROLE);

//add the user group

batch.addUserGroup("gDeviceAdmin", "Device admin user group", roleList);

bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

Step 5 Create another user group gCosDhcpAdmin which is associated with the out of the box roles defined in the RDU server.

```
//Create a new batch
```

```
batch = connection.newBatch();

roleList = new ArrayList<String>();

//Associate the out of the box roles already defined in the RDU Server

roleList.add(RoleTypeValues.COS_ADMIN_ROLE);
roleList.add(RoleTypeValues.DHCP_ADMIN_ROLE);

//Add a user group

batch.addUserGroup("gCosDhcpAdmin", "COS and DHCP Criteria admin user group", roleList);

bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

Step 6 Create a new user uOperator and assign roles, user groups and domains to the user.

```
//Create a new batch

batch = connection.newBatch();

Map<String, Object> userProperties = new HashMap<String, Object>();

//Associate the role rAllDevice as well as other out of the box roles with the user

List<String> userRoleList = new ArrayList<String>();
userRoleList.add("rAllDevice");
userRoleList.add(RoleTypeValues.FILE_ADMIN_ROLE);

//Associate the user group gCosDhcpAdmin with the user

List<String> userGroupList = new ArrayList<String>();
userGroupList.add("gCosDhcpAdmin");

//Associate the user with the domains dUSEast and dUSWest.

List<String> userDomainList = new ArrayList<String>();
userDomainList.add("dUSEast");
userDomainList.add("dUSWest");

userProperties.put(UserDetailsKeys.ROLES_ASSIGNED, userRoleList);
userProperties.put(UserDetailsKeys.GROUPS_ASSIGNED, userGroupList);
userProperties.put(UserDetailsKeys.DOMAINS_ASSIGNED, userDomainList);

//Add the user

batch.addUser("uOperator", "changeit", userProperties);

bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
```

```
{
    System.exit(1);
}
```

Step 7 Create a mapping between a user group defined in an external AAA server and a user group defined in the RDU server.

```
//Create a new batch

batch = connection.newBatch();

Map<String, Object> rduDefaultsProperties =
    new HashMap<String, Object>();

Map<String, String> externalToInternalUserGroupMapping = new HashMap<String, String>();

//Map the user group gDeviceAdmin defined in RDU to the user group externalDeviceAdmin
//defined in external AAA server

externalToInternalUserGroupMapping.put("externalDeviceAdmin",  "gDeviceAdmin");

//Map the user group gCosDhcpAdmin defined in RDU to the user group externalCosDhcpAdmin
//defined in external AAA server

externalToInternalUserGroupMapping.put("externalCosDhcpAdmin", "gCosDhcpAdmin");

rduDefaultsProperties.put(UserAuthenticationKeys.USER_GROUP_MAPPING,
    externalToInternalUserGroupMapping);

//Save the mapping information at the RDU Defaults property level
batch.changeRDUDefaults(rduDefaultsProperties, null);

bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

Operational Access Control

A service provider has created configurations for administering Role Based Access Control. The service provider wants to use the operational level access control.

Desired Outcome

The operator should not be able to perform an operation unless the appropriate privileges necessary for the operation are assigned to the operator.

Step 1 Create a user uCoSAdmin with class of service admin role.

```
//Create a new connection

PACEConnection connection = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "changeit");

//Create a new batch
```

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

//Create a user with class of service admin role

Map<String, Object> userProperties = new HashMap<String, Object>();

//Add Class of Service Admin Role to the user's property

List<String> userRoleList = new ArrayList<String>();
userRoleList.add(RoleTypeValues.COS_ADMIN_ROLE);

userProperties.put(UserDetailsKeys.ROLES_ASSIGNED, userRoleList);

//Add the user

batch.addUser("uCoSAdmin", "changeit", userProperties);

BatchStatus bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
   System.exit(1);
}
```

Step 2 Since the user uCoSAdmin has class of service admin privilege, the user is able to create a class of service successfully.

```
//Create a new PACE Connection with the user uCoSAdmin

PACEConnection connectionUCoSAdmin = PACEConnectionFactory.getInstance(
    "localhost", 49187, "uCoSAdmin", "changeit");

//Create a new batch

batch = connectionUCoSAdmin.newBatch();

//Create a class of service

Map<String, Object> cosProperties = new HashMap<String, Object>();
cosProperties.put(ClassOfServiceKeys.COS_DOCSIS_FILE, "gold.cm");
cosProperties.put(ClassOfServiceKeys.COS_ASSIGNED_DOMAIN,
    DomainTypeValues.ROOT_DOMAIN);

batch.addClassOfService(DeviceType.DOCSIS, "sampleCoS", cosProperties);

//The operation should be successful

bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

Step 3 The user uCoSAdmin tries to set the newly created class of service sampleCoS as the default class of service for DOCSIS technology. But, since the user does not have necessary privilege(s), the operation fails with command status CMD_ERROR_USER_DOES_NOT_HAVE_PRIVILEGE.

```
//User uCoSAdmin creates a new batch

batch = connectionUCoSAdmin.newBatch();

//Try to set sampleCoS  as the default class of service for DOCSIS

Map<String, Object> docsisDefaultsProperties = new HashMap<String, Object>();
docsisDefaultsProperties.put(TechnologyDefaultsKeys.DEFAULT_CLASS_OF_SERVICE,
"sampleCoS");

batch.changeDefaults(DeviceType.DOCSIS, docsisDefaultsProperties, null);

bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}

CommandStatusCode commandStatus = (CommandStatusCode)
bStatus.getCommandStatus(0).getStatusCode();

//Since the user does not have privilege for changing the technology defaults, the
operation fails

if (bStatus.isError() &&
CommandStatusCode.CMD_ERROR_USER_DOES_NOT_HAVE_PRIVILEGE.equals(commandStatus))  {
    // Batch error occurred.

    System.out.println("Failed to change the Docsis technology defaults with the user
sampleCoS"
        + "since the user does not have sufficient privilges");
    System.exit(1);
}
```

Update Protection for Properties at Device Level

A service provider wants to control its operator from updating properties of devices.

Desired Outcome

A role can be associated with a set of device related properties (including custom properties). An operator assigned with that role, is able to administer only those properties to a device.

Step 1 Create a custom property at the RDU Server. This property can be assigned to a device.

```
//Create a PACE Connection

PACEConnection connection = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "changeit");

//Create a new batch
```

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

//Add a custom property to the RDU Server

batch.addCustomPropertyDefinition("/role/owner", DataType.STRING);

BatchStatus bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    //System.exit(1);
}
```

Step 2 Create a role rDevAdmin with different device related properties and custom properties. An operator with the role is able to administer only those properties to a device.

```
//Create a new batch

batch = connection.newBatch();

//Assign all the device related privileges to the role

List <String> allPrivileges = new ArrayList<String>();

allPrivileges.add(PrivilegeTypeValues.PRIV_DEVICE_CREATE);
allPrivileges.add(PrivilegeTypeValues.PRIV_DEVICE_DELETE);
allPrivileges.add(PrivilegeTypeValues.PRIV_DEVICE_OPERATION);
allPrivileges.add(PrivilegeTypeValues.PRIV_DEVICE_READ);
allPrivileges.add(PrivilegeTypeValues.PRIV_DEVICE_REGEN);
allPrivileges.add(PrivilegeTypeValues.PRIV_DEVICE_UPDATE);

Map<String, Object> properties = new HashMap<String, Object>();

List<String> devicesProperties = new ArrayList<String>();

//Add device related properties to the role

//Add Class Of Service as a modifiable property

devicesProperties.add(DeviceDetailsKeys.CLASS_OF_SERVICE);

//Add the following properties as modifiable properties

devicesProperties.add(FqdnKeys.AUTO_FQDN_PREFIX);
devicesProperties.add(FqdnKeys.AUTO_FQDN_SUFFIX);
devicesProperties.add(FqdnKeys.AUTO_FQDN_ENABLE);
devicesProperties.add(FqdnKeys.AUTO_FQDN_DOMAIN);

//Add the custom property "/role/owner" as a modifiable property
devicesProperties.add("/role/owner");

//Add modifiable device related properties to the role

properties.put(RoleDetailsKeys.MODIFIABLE_DEVICE_PROPERTIES,
     devicesProperties);

//Add the role

batch.addRole("rDevAdmin", "Role with all device privileges",
    allPrivileges, properties);
```

```
bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    //System.exit(1);
}
```

Step 3 Create an operator with rDevAdmin role.

```
//Create a new batch

batch = connection.newBatch();

Map<String, Object> userProperties = new HashMap<String, Object>();

//Associate the operator with the role rDevAdmin

List<String> userRoleList = new ArrayList<String>();
userRoleList.add("rDevAdmin");

userProperties.put(UserDetailsKeys.ROLES_ASSIGNED, userRoleList);

//Create the user

batch.addUser("uDeviceAdminOperator", "changeit", userProperties);

bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    //System.exit(1);
}
```

Step 4 Operator uDeviceAdminOperator tries to add a device with few properties all of which are associated with its role as modifiable device related properties.

The device is added successfully.

```
PACEConnection connectionUser = PACEConnectionFactory.getInstance(
    "localhost", 49187, "uDeviceAdminOperator", "changeit");

batch = connectionUser.newBatch();

DeviceID modemMacAddress = DeviceID.getInstance("1,6,00:11:00:00:00:92",
KeyType.MAC_ADDRESS);
List<DeviceID> deviceIDs = new ArrayList<DeviceID>();
deviceIDs.add(modemMacAddress);

Map<String, Object> prop = new HashMap<String, Object>();
prop.put(FqdnKeys.AUTO_FQDN_DOMAIN, "false");
prop.put("/role/owner", "abcd");

batch.add(DeviceType.DOCSIS, deviceIDs, null, null, null, "sample-bronze-docsis", null,
prop);

bStatus = null;
try
```

```
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

Step 5 Operator uDeviceAdminOperator tries to modify the DHCP Criteria of the device.

The operation fails as this role does not have DHCP Criteria as modifiable property of a device.

```
//Create a new batch

batch = connectionUser.newBatch();

//Change the DHCP Criteria of the device

batch.changeDHCPCriteria(modemMacAddress, "sample-provisioned");

bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}

CommandStatus commandStatus = bStatus.getCommandStatus(0);
StatusCode statusCode = commandStatus.getStatusCode();

System.out.println(bStatus);

// Batch error occurred.

if (bStatus.isError())  {
    System.out.println("Failed to modify the DHCP Criteria of the device with MAC Address
" + modemMacAddress);

//The operation should fail with CMD_ERROR_USER_CAN_NOT_UPDATE_PROPERTY command
//status code.

    if (CommandStatusCode. CMD_ERROR_USER_CAN_NOT_UPDATE_PROPERTY.equals(statusCode)) {
        System.out.println("The user uDeviceAdminOperator can't update the DHCP Criteria
of a device");
    }
    System.exit(1);
}
```

Domain Administration and Instance Level Access Control

While operational level access control defines what actions a user can perform, instance level access control determines whether or not those actions can be performance on a specific instances. Domain and instance level authorization are independent of roles and privileges.

This section includes these use cases:

- Assigning Existing Configuration Objects to a Domain, page 7-69

- Instance Level Authorization, page 7-71

Assigning Existing Configuration Objects to a Domain

An administrator wants to assign the existing resources such as devices, Classes of Service, DHCP Criterion, files, DPEs, CPNRs, and Provisioning Groups to a domain.

Desired Outcome

The resources are being assigned to the specified domain. The service provider's administrative application displays the changed domain information for each of the resources.

1. **Use Case 1**: Assign the existing classes of service sample-bronze-docsis, sample-gold-docsis, and unprovisioned-stb to the domain dUS.

```
//Create a new connection

PACEConnection connection = PACEConnectionFactory.getInstance("localhost", 49187, "admin",
"changeit");

//Create a new batch

Batch batch = connection.newBatch();

//List of classes of service to be assigned to the domain dUS

List<String> cosList = new ArrayList<String>();
cosList.add("sample-bronze-docsis");
cosList.add("sample-gold-docsis");
cosList.add("unprovisioned-stb");

//Assign the classes of service to the domain

batch.changeDomainProperties("dUS", null, cosList, ResourceType.CLASS_OF_SERVICE);

BatchStatus bStatus = null;

//Post the batch to RDU Server
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

2. **Use Case 2:** Assign the existing DHCP criterion sample-provisioned and unprovisioned-cablehome-wan-data to the domain dUSEast.

```
//Create a new batch

batch = connection.newBatch();

//List of DHCP Criterion to be assigned to the domain dUSEast

List<String> dhcpList = new ArrayList<String>();
dhcpList.add("sample-provisioned");
dhcpList.add("unprovisioned-cablehome-wan-data");

// Assign the DHCP Criterion to the domain

batch.changeDomainProperties("dUSEast", null, dhcpList, ResourceType.DHCP_CRITERIA);
```

```
bStatus = null;

//Post the batch to RDU Server

try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

3. **Use Case 3:** Assign the existing files bronze.cm, gold.cm, unprov.cm to the domain dUSWest.

```
//Create a new batch

batch = connection.newBatch();

//List of files to be assigned to the domain dUSWest

List<String> fileList = new ArrayList<String>();
fileList.add("bronze.cm");
fileList.add("gold.cm");
fileList.add("unprov.cm");

// Assign the files to the domain

batch.changeDomainProperties("dUSWest", null, fileList, ResourceType.FILE);

bStatus = null;

//Post the batch to RDU Server

try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

4. **Use Case 4**: Assign the devices already existing in the RDU server to the domain dUSWest.

```
//Create a new batch

batch = connection.newBatch();

// List of devices to be assigned to the domain dUSWest

List<DeviceID> deviceList = new ArrayList<DeviceID>();
deviceList.add(DeviceID.getInstance("1,6,00:11:00:00:00:01", KeyType.MAC_ADDRESS));
deviceList.add(DeviceID.getInstance("1,6,00:11:00:00:00:02", KeyType.MAC_ADDRESS));
deviceList.add(DeviceID.getInstance("03:00:00:01:00:00", KeyType.DUID));

// Assign the devices to the domain

batch.changeDeviceDomainProperties("dUSWest", null, deviceList);

bStatus = null;

//Post the batch to RDU Server
```

```
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

5. **Use Case 5**: Assign a DPE already in ready state to the domain dUS.

```
//Create a new batch

batch = connection.newBatch();

// List of DPEs to be assigned to the domain dUS

List<String> dpeList = new ArrayList<String>();
dpeList.add("cpc-rhel-test.cisco.com");

//Assign the DPEs to the domains

batch.changeDomainProperties("dUS", null, dpeList, ResourceType.DPE);

bStatus = null;

//Post the batch to RDU Server

try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

Instance Level Authorization

A service provider has created configurations for administering Role Based Access Control. The service provider wants to use the instance level authorization for various resources such as Device, Class of service, DHCP Criteria, File, DPE, CPNR, and Provisioning Group.

Desired Outcome

Only those operators who have access to a specific resource must be allowed to operate on it.

Step 1 Enable instance level authorization feature at the RDU Server.

```
//Create a new connection by admin user.

PACEConnection connection = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "changeit");

//Create a new batch

Batch batch = connection.newBatch();

//Enable instance level authorization at the RDU Server.

Map<String, Object> properties = new HashMap<String, Object>();
```

```
properties.put(ServerDefaultsKeys.INSTANCE_LEVEL_AUTH_ENABLE, true);
batch.changeRDUDefaults(properties, null);

BatchStatus bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

Step 2 Create a user uDUS with administrator role and associate the user to the domain dUS and not to the domain RootDomain.

Step 3 Create a class of service cosDUSEast associated with the domain dUSEast and another class of service cosRootDomain associated with the domain RootDomain.

Since instance level authorization is enabled, the user uDUS is be able to access only the resources associated to the domain dUS and its children (dUSEast, dUSWest). But, the user is not authorized to access resources associated to RootDomain.

```
//Create a batch

batch = connection.newBatch();

Map<String, Object> userProperties = new HashMap<String, Object>();

//Assign the administrator role to the user

List<String> userRoleList = new ArrayList<String>();
userRoleList.add(RoleTypeValues.SUPER_ADMIN_ROLE);

//Associate the user to the domain dUS

List<String> userDomainList = new ArrayList<String>();
userDomainList.add("dUS");

userProperties.put(UserDetailsKeys.ROLES_ASSIGNED, userRoleList);
userProperties.put(UserDetailsKeys.DOMAINS_ASSIGNED, userDomainList);

//add the user

batch.addUser("uDUS", "changeit", userProperties);

//Create a class of service and associate it to the domain dUSEast

Map<String, Object> cosProperties = new HashMap<String, Object>();
cosProperties.put(ClassOfServiceKeys.COS_ASSIGNED_DOMAIN, "dUSEast");
cosProperties.put(ClassOfServiceKeys.COS_DOCSIS_FILE, "gold.cm");

batch.addClassOfService(DeviceType.DOCSIS, "cosDUSEast", cosProperties);

//Create another class of service and associate it to the domain RootDomain.

cosProperties = new HashMap<String, Object>();
cosProperties.put(ClassOfServiceKeys.COS_ASSIGNED_DOMAIN,
DomainTypeValues.ROOT_DOMAIN);
cosProperties.put(ClassOfServiceKeys.COS_DOCSIS_FILE, "gold.cm");

batch.addClassOfService(DeviceType.DOCSIS, "cosRootDomain", cosProperties);
```

```
bStatus = null;
try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

Step 4 The user uDUS tries to get the details of the class of service cosDUSEast belonging to the domain dUSEast. Since the user has access to the domain dUS and its children (dUSEast, dUSWest), user can successfully get the class of service details.

```
//Create a PACEConnection by the user uDUS

PACEConnection connectionByDUS = PACEConnectionFactory.getInstance(
    "localhost", 49187, "uDUS", "changeit");

//Create a new batch

batch = connectionByDUS.newBatch();

//User uDUS tries to get the details of class of service cosDUSEast

batch.getClassOfServiceProperties("cosDUSEast");

bStatus = null;

//The batch should successfully return the class of service details.

try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
    System.exit(1);
}
```

Step 5 The user uDUS tries to get the details of the class of service cosRootDomain which is associated with the domain RootDomain. Since the user does not have access to RootDomain, the user is not authorized to get details of the class of service.

```
//Create a new batch from the PACEConnection connectionByDUS

batch = connectionByDUS.newBatch();

//User uDUS tries to get the details of class of service cosRootDomain

batch.getClassOfServiceProperties("cosRootDomain");

bStatus = null;

//Post the batch

try
{
    bStatus = batch.post();
}
catch (ProvisioningException pe)
{
```

```
    System.exit(1);
}

CommandStatus commandStatus = bStatus.getCommandStatus(0);
StatusCode statusCode = commandStatus.getStatusCode();

//The batch should fail with Command Status code CMD_ERROR_USER_NOT_AUTHORIZED
//as the user uDUS does not have access to the class of service cosRootDomain

if (bStatus.isError() &&
    CommandStatusCode.CMD_ERROR_USER_NOT_AUTHORIZED.equals(statusCode))  {
    // Batch error occurred.
System.out.println("User uDUS failed to get the details of the class of service
cosRootDomain");
    System.exit(1);
}
```

CRS Management

Prime Cable Provisioning provides greater control on CRS, such as you can enable, disable, pause, and resume CRS without restarting RDU. You can also view, filter, and delete any request queued by CRS.

Desired Outcome

The operator must be able to control CRS without restating RDU.

Step 1 Enable CRS.

```
//Create a new connection by admin user.

PACEConnection connection = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "changeit");

Batch batch = connection.newBatch();
final Map<String, Object> map = new HashMap<String, Object>();
//Enabling CRS using SERVER_CRS_ENABLE
map.put(ServerDefaultsKeys.SERVER_CRS_ENABLE, Boolean.TRUE);
batch.changeRDUDefaults(map, null);
BatchStatus bStatus = null;
//Post the batch to RDU Server
try
{
bStatus = batch.post();
}
catch (ProvisioningException pe)
{
System.exit(1);
}
```

Step 2 Change CoS to add CRS request to the queue.

```
//Create a new connection by admin user.

PACEConnection connection = PACEConnectionFactory.getInstance(
```

```
    "localhost", 49187, "admin", "changeit");

Batch batch = connection.newBatch();
Map propAdd1 = new HashMap();
propAdd1.put(ClassOfServiceKeys.COS_DOCSIS_FILE, "gold.cm");
//Adding request to the CRS queue
batch.changeClassOfServiceProperties("bac1", propAdd1, null);
BatchStatus bStatus = null;
//Post the batch to RDU Server
try
{
bStatus = batch.post();
}
catch (ProvisioningException pe)
{
System.exit(1);
}
```

Step 3 Fetch CRS Statistics using getRDUDetails() API.

```
//Create a new connection by admin user.

PACEConnection connection = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "changeit");

Batch batch = connection.newBatch();
//Fetch CRS Statistics using getRDUDetails
batch.getRDUDetails("localhost");
BatchStatus bStatus = null;
//Post the batch to RDU Server
try
{
bStatus = batch.post();
}
catch (ProvisioningException pe)
{
System.exit(1);
}
```

Step 4 Get details of all the requests in the queue.

```
private static SearchBookmark printRecordSearchResults(RecordSearchResults rs)
    throws Exception
    {
        SearchBookmark sb = rs.getSearchBookmark();
        List<RecordData> rdlist = rs.getRecordData();
        Iterator<RecordData> iter = rdlist.iterator();
```

```
        while (iter.hasNext())
        {
            RecordData rdObj = iter.next();
            Key keyObj = rdObj.getPrimaryKey();
           System.out.println("CrsRequestId: " + ((CrsRequestId)keyObj).getRequestId());
            Map crsDetails = rdObj.getDetails();
        }
        return sb;
    }

    public static void getAllCRSRequestsQueuedBasedOnDHCPCriteria() throws Exception
    {
        ResourceNamePattern pattern = new ResourceNamePattern("*",
CRSRequestType.DHCP_CRITERIA);
        CRSSearchType cst = new CRSSearchType(ReturnParameters.ALL);
        RecordSearchResults rs = null;
        SearchBookmark sb = null;

        rs = searchCRSRequest(cst, sb);
        sb = rs.getSearchBookmark();

        while (sb != null)
        {
            // print out the data in the record search result.
            sb = printRecordSearchResults(rs);
            // call the search routine again
            rs = searchCRSRequest(cst, sb);
        }
    }

   private static RecordSearchResults searchCRSRequest(CRSSearchType cst, SearchBookmark
sb) throws Exception
    {
        RecordSearchResults rs = null;
        PACEConnection s_conn = null;
        final Batch batch = s_conn.newBatch();
        final int numberOfRecordReturn = 10;
```

```
        //calling the search API
        batch.searchCRSRequest(cst, sb, numberOfRecordReturn);

        // Call the RDU.
        BatchStatus batchStatus = batch.post();

        // Check for success.
        CommandStatus commandStatus = null;
        if (0 < batchStatus.getCommandCount())
        {
            commandStatus = batchStatus.getCommandStatus(0);
        }
        //check to see if there is an error
        if (batchStatus.isError()
                || batchStatus.isWarning()
                || commandStatus == null
                || commandStatus.isError())
        {
            System.out.println("report batch error.");
            return null;
        }

        //batch success without error, retrieve the result
        //this is a list of CRS Requests
        rs = (RecordSearchResults)commandStatus.getData();
        return rs;
    }
```

Step 5 Pause CRS. When CRS is paused, execution of CRS requests is paused.

```
//Create a new connection by admin user.

PACEConnection connection = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "changeit");

Batch batch = connection.newBatch();
final Map<String, Object> map = new HashMap<String, Object>();
//Pause CRS using SERVER_CRS_PAUSED
map.put(ServerDefaultsKeys.SERVER_CRS_PAUSED,  Boolean.TRUE);
batch.changeRDUDefaults(map, null);
BatchStatus bStatus = null;
//Post the batch to RDU Server
```

```
try
{
bStatus = batch.post();
}
catch (ProvisioningException pe)
{
System.exit(1);
}
```

Step 6 Delete the CRS requests.

```
//Create a new connection by admin user.

PACEConnection connection = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "changeit");

Batch batch = connection.newBatch();
//Delete CRS using deleteCRSRequests
List crsRequestIdList = new ArrayList();
crsRequestIdList.add("325f44454641554c545f434c4153535f4f465f534552564943455f444f4353495
34d4f44454d3030");
  crsRequestIdList.add("3273616d706c652d676f6c642d646f637369733230");
batch.deleteCRSRequests(crsRequestIdList);


BatchStatus bStatus = null;
//Post the batch to RDU Server
try
{
bStatus = batch.post();
}
catch (ProvisioningException pe)
{
System.exit(1);
}
```

Step 7 Resume CRS. When CRS is resumed, the execution of CRS requests is resumed.

```
//Create a new connection by admin user.

PACEConnection connection = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "changeit");

Batch batch = connection.newBatch();
final Map<String, Object> map = new HashMap<String, Object>();
//Resume CRS using SERVER_CRS_PAUSED
map.put(ServerDefaultsKeys.SERVER_CRS_PAUSED,  Boolean.FALSE);
batch.changeRDUDefaults(map, null);
BatchStatus bStatus = null;
//Post the batch to RDU Server
try
```

```
{
bStatus = batch.post();
}
catch (ProvisioningException pe)
{
System.exit(1);
}
```

Step 8 Disable CRS. When CRS is disabled the entire CRS service is stopped and existing requests in the queue are cleared automatically.

```
//Create a new connection by admin user.

PACEConnection connection = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "changeit");

Batch batch = connection.newBatch();
final Map<String, Object> map = new HashMap<String, Object>();
//Disable CRS using SERVER_CRS_ENABLE
map.put(ServerDefaultsKeys.SERVER_CRS_ENABLE, Boolean.FALSE);
batch.changeRDUDefaults(map, null);
BatchStatus bStatus = null;
//Post the batch to RDU Server
try
{
bStatus = batch.post();
}
catch (ProvisioningException pe)
{
System.exit(1);
}
```

Step 9 Set the policies for CRS pause on exceeding failure threshold.

When SERVER_CRS_PAUSE_ON_FAILURE_THRESHOLD is set to true and once the percentage of failed devices exceeds SERVER_CRS_FAILURE_THRESHOLD_PERCENTAGE, CRS automatically pauses. By default SERVER_CRS_PAUSE_ON_FAILURE_THRESHOLD is set to false.

```
//Create a new connection by admin user.

PACEConnection connection = PACEConnectionFactory.getInstance(
    "localhost", 49187, "admin", "changeit");

Batch batch = connection.newBatch();
final Map<String, Object> map = new HashMap<String, Object>();
//Set the policies for CRS using SERVER_CRS_PAUSE_ON_FAILURE_THRESHOLD and
SERVER_CRS_FAILURE_THRESHOLD_PERCENTAGE
map.put(ServerDefaultsKeys.SERVER_CRS_PAUSE_ON_FAILURE_THRESHOLD, Boolean.FALSE);
map.put(ServerDefaultsKeys.SERVER_CRS_FAILURE_THRESHOLD_PERCENTAGE,
Float.parseFloat("10"));
batch.changeRDUDefaults(map, null);
```

```
BatchStatus bStatus = null;
//Post the batch to RDU Server
try
{
bStatus = batch.post();
}
catch (ProvisioningException pe)
{
System.exit(1);
}
```