CHAPTER 14

Using the Device Credentials Repository

The Device Credentials Repository (DCR) is a common repository of devices, their attributes and their credentials. An DCR system consists of one or more DCR Servers that store and distribute device information, applications using DCR APIs to access this information, and the Device & Credentials Admin GUI accessible from the CiscoWorks Home Page.

When implemented, DCR provides:

- Easier, centralized access to device and credentials data.
- Secure device-data persistence, access and transport.
- Rationalized and controlled replication of device information, with less user-level reconciliation.
- Better integration with third-party and Cisco network-management applications.

Note DCR stores device information. It does not communicate with devices directly. The code that interacts directly with devices and fetches their data for storage in DCR is the responsibility of your application.

The following topics describe how to integrate DCR with your application:

- Understanding DCR
- Using DCR

For more information about DCR, see:

- *Device List and Credentials Repository Server Functional Specification, EDCS-283571*
- *DCR Deployment Scenarios, Use Cases and Guidelines for Applications, EDCS-283285*
- *Device List and Credentials Repository Master/Slave Functional Specification, EDCS-283284*
- *Device list and Credentials Repository Import Export Software Functional Specification, EDCS-285702*

Note CiscoWorks users know DCR by the acronym DCA, after the Device & Credentials Admin GUI they use to update DCR device lists. Some developers also use the DCA acronym to refer to DCR.

Understanding DCR

All network management applications need to store basic attributes of the devices they manage. This includes device credentials, such as SNMP community strings consisting of a user ID-and-password pair.

Devices and their credentials represent a special data management problem. They must be both:

- Shared: Multiple copies of device data are wasteful. If there is no single device data store that can be shared, users of multiple network management applications must waste resources reconciling the data, either via manual means or via automated tools they must build and maintain.
- Secure: Independent copies of credentials are insecure by definition, as they must be manually copied among applications, or transmitted using insecure means. Insecure credentials are dangerous, as anyone with access to credentials may reconfigure devices in destructive ways.

The DCR solution enables multiple applications to share device lists and credentials using a client-server mechanism, with secure storage and communications.

The following topics present basic information on DCR and how it works:

- About DCR Features and Benefits
- How DCR Works
- About the DCR Modes
- About the DCR Components
- How DCR Masters and Slaves Interact
- How DCR Secures Device and Credentials Data
- About DCR Data Storage
- Integrating DCR with OGS
- Integrating DCR with ACS
- About DCR Events

About DCR Features and Benefits

DCR offers the following capabilities:

- DCR stores device attributes and credentials, permits users to attach custom data to devices, and permits default grouping of devices.
- DCR supports proxy device attributes, storage of IPv6 and SNMP v3 device and credential information, and assigns a unique, internally generated DCR Device ID to every device.
- DCR supports "unreachable devices" — devices known to be "on the shelf" and not yet deployed, or "phone home" devices in transit to their location during initial deployment. DCR allows users to import these devices and retain credentials for them even though some critical information about them may not be available at first.
- DCR supports device pre-provisioning. Applications often do not know essential attributes of pre-provisioned devices, such as their host names or IP addresses. Using the DCR device_identity attribute, applications can uniquely identify pre-provisioned devices.
- Clients can add, modify, and delete DCR devices using the DCA GUI. DCR also permits users to populate the database via import from many sources, and to export device data for use with third-party products like HP OpenView and Netview.

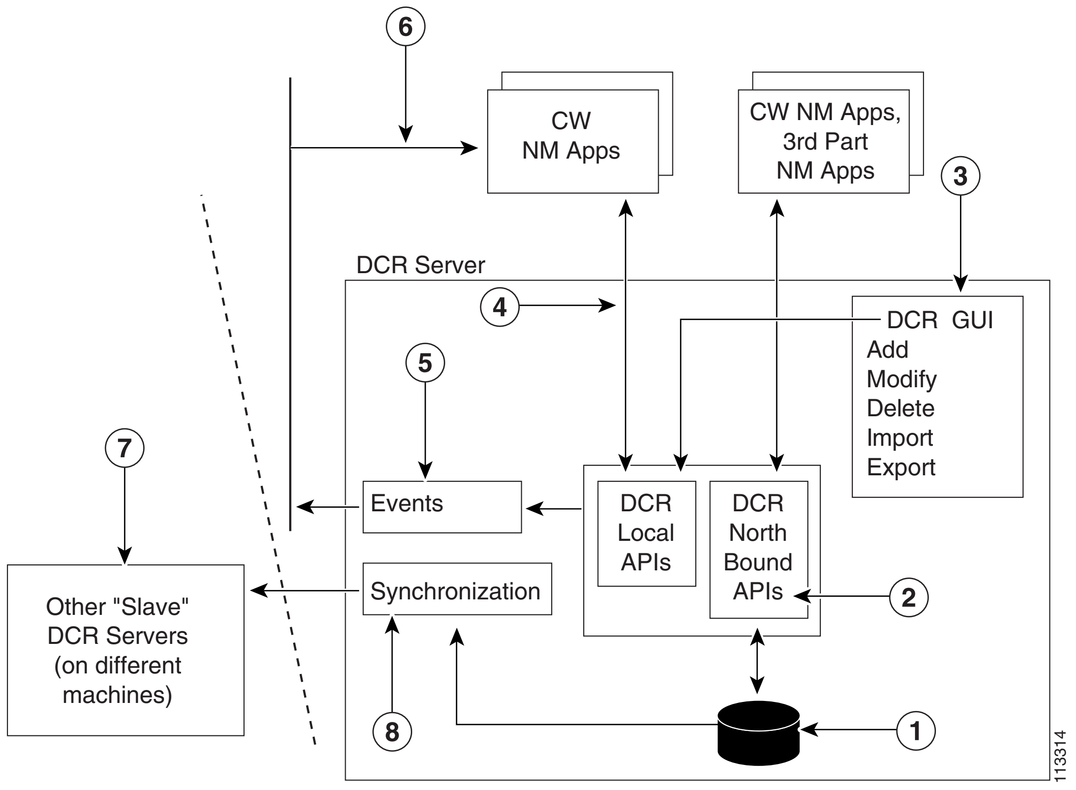
- DCR uses several attributes to detect and stop the creation of duplicate devices. If a new device has (or an update causes an existing device to change to) a display_name or fully-qualified DNS name (i.e., host_name + domain_name) that is the same as an existing device, the operation will fail.
- DCR data distribution is flexible, scalable, reliable, using a network of Master/Slave servers. All server-to-server communications take place using CSTM (see Chapter 31, "Using the Common Services Transport Mechanism").
- DCR data is secure. DCR encrypts credential data using a static encryption scheme that must be hard-coded into the methods used for encryption (considered acceptable by the security team).
- DCR communications are secure. Access to device data via API calls or the DCA GUI is performed only by secured channel (HTTPS) and authorized clients. DCR sends events to applications using ESS (see Chapter 19, "Using Event Services Software").
- DCR supports a full range of administrative features, including the DCA GUI, change logs and reports, and full compatibility with CWCS backup and restore.

How DCR Works

Figure 14-1shows the DCR system flow:

1. DCR provides a database structure to store device lists, the attributes for each device, and device credentials.
2. DCR provides north-bound APIs to allow remote applications to add, update and manage the data. It also provides Java APIs for local CiscoWorks network management applications.
3. A DCR Server in Master or Standalone mode (see the "About the DCR Modes" section on page 14-4) provides the DCA GUI to administer device data. Users can use this GUI to add or import devices and credentials, modify device credential data, or delete devices. Note that only the Master and Standalone modes provide access to DCA. CiscoWorks applications using a DCR Server in Slave mode cannot use DCA.
4. Cisco Works applications residing on the same machine as DCR can access or modify the data using local Java APIs (most CiscoWorks applications) or north-bound APIs (e.g., C++ CiscoWorks applications). Third-party network management applications can also access or modify the data using the secure north-bound APIs.
5. Changes in DCR data (whether the DCR Server is in Master or Slave mode) are broadcast to network management applications on that server as Event Services Software (ESS) events. Third-party applications who register with DCR can also receive HTTP- based broadcasts of these events.
6. DCR clients (any Cisco Works application) may choose to listen to these events. Upon receiving these events, the application can ignore any events that are of no interest to it (e.g, applications may want to filter out updates about devices they do not support). Upon filtering the data, the application can get the device details from DCR through APIs.
7. DCRs in Slave mode reside on other CWCS Servers installed on different machines.
8. A DCR in slave mode synchronizes its data with that of its master automatically every 45 seconds (the default) or whenever there is a change in the master data (the slave must be set to listen for and receive sync events from the master). During synchronization, the slave fetches all newly updated data from the Master and updates its local database accordingly.

Figure 14-1 DCR System Flow



About the DCR Modes

Any instance of DCR can run in Master, Slave, or Standalone mode:

- **In Master mode**, DCR hosts the authoritative, or master, list of all devices and their credentials. All other DCRs in the same management domain which are running in Slave mode will normally share this list. There can be only one DCR running in Master mode in a single management domain. All Slave DCRs update the Master DCR before updating themselves, and changes to the Master DCR device data are propagated to Slave DCRs using CWCS CSTM. The Master always contains the most up-to-date device data in the management domain.
- **In Slave mode**, DCR maintains an exact replica of the data managed by the Master DCR for the management domain. There can be multiple Slave DCRs per management domain. When a client application updates device data in a Slave DCR, that Slave DCR will first send the update to the Master DCR, and then update itself from the Master. Since it is possible for a Slave to miss some updates, Slaves can also initiate synchronization with the Master DCR.
- **In Standalone mode**, DCR maintains an independent repository of device list and credential data. It does not participate in a management domain and its data is not shared with any other DCR. It does not communicate with or contain registration information about any other Master, Slave, or Standalone DCR.

When considering DCR modes, remember that:

- DCR is a shared component, so there is only one DCR Server per CWCS Server. All client applications installed on that CWCS Server use the local DCR Server running on that machine, regardless of mode.
- The DCR Server mode is transparent to applications.
- DCR is not fault tolerant. It does not ensure that client applications receive update notifications, and does not validate the repository data. It relies on DCR clients to update repository data.
- DCR currently supports only one Master per management domain, with multiple Slaves. It does not support multiple Master DCR Servers in the same domain.
- All newly-installed DCR Servers start in Standalone mode by default. Users set it to Master or Slave mode using the DCA GUI or via the DCR CLI (see "Using the DCR Command-Line Interface" section on page 14-46).

About the DCR Components

An instance of DCR consists of the following components:

- **Repository:** Stores and maintains device and credentials data in the CWCS database. It includes all relevant database server functions, including the APIs providing access to the data, duplicate-device detection, and maintenance of time stamps.

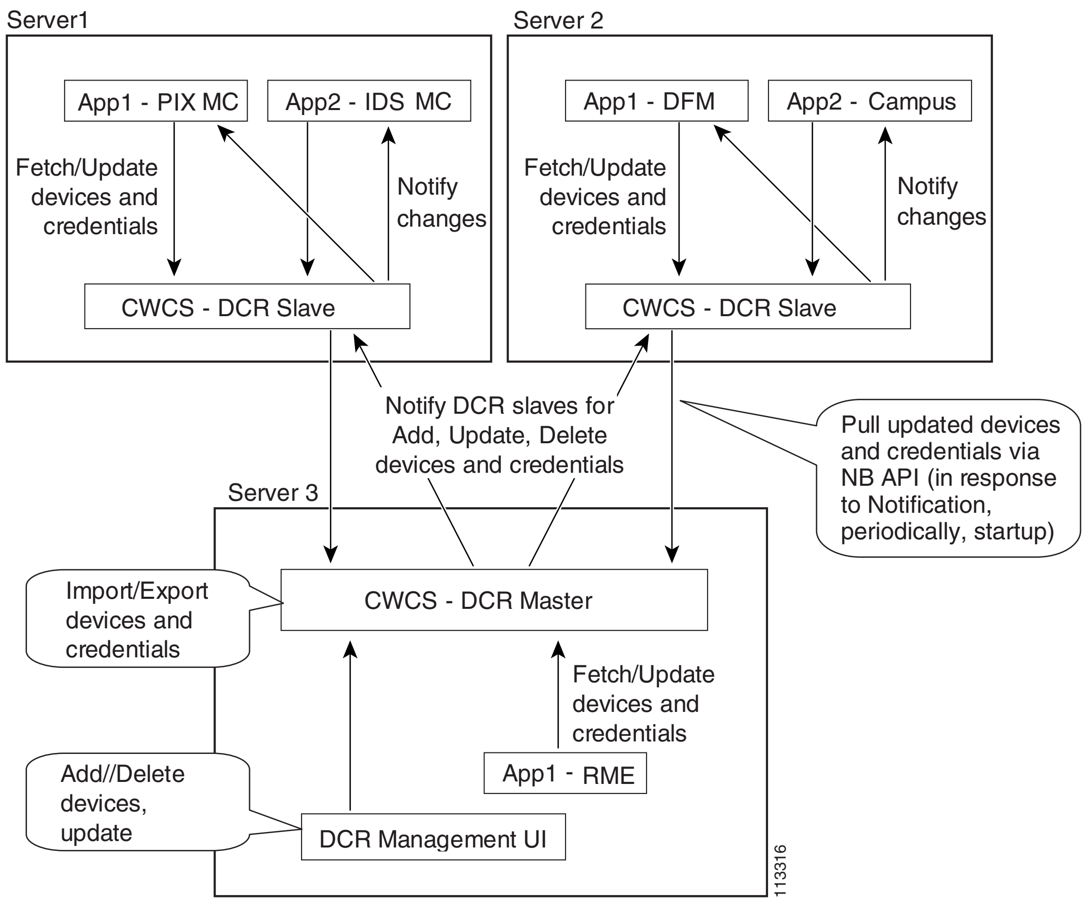
  The Repository also includes DCR-specific functions. These include generating the unique DCR Device IDs used to identify each device on both Masters and Slaves, information about the DCR Server's current mode, and synchronization information.
- **Registry**: Maintains the information needed to contact corresponding DCR Servers. For a Master DCR Server, it stores the URL and port number of each of its Slaves; for a DCR Slave, it stores the URL and port of the Master.
- **Notifier**: Active only on Master DCR Servers, it notifies Slaves of changes in Repository data whenever such changes occur. The Notifier uses the Registry to get the contact details for all of the Slaves. DCR uses CSTM (see Chapter 31, "Using the Common Services Transport Mechanism") to send the notifications.
- **Receiver**: Active only on Slaves, it processes notifications from the Master about Repository data changes. To keep Slave data tightly coupled with the Master, Slaves generate events for these update notifications and send the events to applications using the Slaves (note that notifications are only exchanged among servers; applications only receive events). Applications are responsible for filtering out events for devices in which they are not interested, using the DCR Device ID (and other details, such as the sysObjectID) contained in the event.
- **Synchronizer**: Lets Slave DCRs get updates from their respective Masters on a regular basis, and in response to update notifications from the Master.
- **Event Generator**: Generates events for the applications installed on and sharing a local DCR on a single CWCS server. The Event Generator runs regardless of the DCR mode (see the "How DCR Masters and Slaves Interact" section on page 14-6). The generated events notify subscribing applications that there is modified, new or deleted device or credentials information in the repository. Each event carries enough information for the subscribing application to determine if it does or does not want the new information, and if so, to formulate a query to retrieve it. Applications are responsible for subscribing to these events, and can receive them either via ESS or HTTP (see the "About DCR Events" section on page 14-15).

How DCR Masters and Slaves Interact

Figure 14-2 shows an example of a DCR Master/Slave setup. In this example, several CiscoWorks applications are installed on different CWCS Servers. Each server has one local DCR Server. The applications installed on each CWCS Server interact with the local DCR Server only. For example: PIX MC and IDS MC on Server 1 interact with the DCR Server in Slave mode on Server 1; DFM and Campus Manager interact with the DCR Server in Slave mode on Server 2; RME interacts with the DCR Server in Master mode on Server 3.

The Master sends notifications of any changes in its repository to both Slaves. The Slaves in turn make synchronize with the Master to get its device attribute and credential updates. Both the Master and the Slaves also broadcast events to applications via ESS (they also publish events via HTTP to subscribed third-party applications).

Figure 14-2 Example DCR Master/Slave Setup


Server1
App1 - PIX MC
App2 - IDS MC
Fetch/Update devices and credentials
Notify changes
CWCS - DCR Slave
Server 2
App1 - DFM
App2 - Campus
Fetch/Update devices and credentials
Notify changes
CWCS - DCR Slave
Notify DCR slaves for Add, Update, Delete devices and credentials
Pull updated devices and credentials via NB API (in response to Notification, periodically, startup)
Server 3
Import/Export devices and credentials
CWCS - DCR Master
Fetch/Update devices and credentials
App1 - RME
Add//Delete devices, update
DCR Management UI
113316

The interactions among this Master and its Slaves vary depending on the kinds of operations being performed and the DCR Server mode. The following topics explain what happens in each case:

- How DCR Adds Devices
- How DCR Modifies Devices
- How DCR Deletes Devices

How DCR Adds Devices

Users whose application resides on the same CWCS server with a DCR Master can use the local DCR APIs or the DCA GUI to add a new device directly to that Master. When this happens:

1. The DCR Master checks whether the device is a duplicate or not. If it is a duplicate, the operation fails.
2. If the new device is not a duplicate,tThe DCR Master adds the device to the repository.
3. The DCR Master then:
   a. Sends an update notification to all of its DCR Slaves.
   b. Sends “New Device Added” events to all applications using the Master.
4. Each DCR Slave receiving update notification from its Master then:
   - Synchronizes with its Master to get the device details.
   - Sends “New Device Added” events to all applications using that DCR Slave.

Users whose application resides on the same CWCS Server with a Slave can use the application UI or the DCR local API to add a new device. When this happens:

1. The Slave calls the Master to add the device to the Master’s Repository.
2. If the DCR Master is down, the new device is a duplicate, or there is another problem, the Master returns an error to the Slave. The Slave returns an error to the application, and no updates take place on the Master or the Slave.
3. If there are no problems, the DCR Master adds the device to its repository.
4. The DCR Master then:
   a. Sends an update notification to its DCR Slaves.
   b. Sends “New Device Added” events to the applications using that DCRMaster.
5. Each DCR Slave receiving update notification from its Master then:
   - Synchronizes with its Master to get the device details.
   - Sends “New Device Added” events to all applications using that DCR Slave.

How DCR Modifies Devices

Users whose application resides on the same CWCS server with a DCR Master can use the local DCR APIs or the DCA GUI to update devices on that Master. When this happens::

1. The DCR Master checks whether the modifications will turn the modified device into a duplicate of another device in the repository. If it is a duplicate, the operation fails.
2. If the device update is not a duplicate, the DCR Master udpates the device in the repository.
3. The DCR Master then:
   a. Sends an update notification to its DCR Slaves.
   b. Sends “Device Updated” events to all applications using the Master.
4. Each DCR Slave receiving the update notification from its Master then:
   - Synchronizes with its Master to get the device details.
   - Sends “Device Updated” events to all applications using that DCR Slave.

Users whose application resides on the same CWCS Server with a Slave can use the application UI or the DCR local API to update an existing device. When this happens:

1. The Slave calls the Master to modify the device in the Master's Repository.
2. If the DCR Master is down, the updated device is a duplicate, or there is another problem, the Master returns an error to the Slave. The Slave returns an error to the application, and no updates take place on the Master or the Slave.
3. If there are no problems, the DCR Master adds the device to its repository.
4. The DCR Master then:
   a. Sends an update notification to its DCR Slaves.
   b. Sends "Device Updated" events to all applications using the Master.
5. Each DCR Slave receiving update notification from its Master will:
   – Synchronize with its Master and get the device details.
   – Send "Device Updated" events to all applications using that DCR Slave.

How DCR Deletes Devices

Applications cannot delete devices directly. Only administrators using the DCA GUI or the DCR CLI on a DCR Server in Master or Standalone mode can delete devices. When this happens, the Master:

1. Deletes the device from the Master repository.
2. Sends "Device Deleted" notifications to its DCR Slaves .
3. Sends "Device Deleted" events to all applications using that DCR Master.
4. Each DCR Slave receiving the "Device Deleted" notifications from its Master will:
   – Synchronize with its Master.
   – Send "Device Deleted" events to all applications using that DCR Slave.

How DCR Secures Device and Credentials Data

DCR allows access to device and credentials data only by authenticated, authorized users. DCR provides a basic level of security by storing device credentials as encrypted data (other attribute data is in plain text), and by using secure communications protocols.

DCR also uses the following security keys:

- Username: The user name of the accessing user.
- Password: The accessing user's password.
- Secret Key: A password supplied by the administrator during CWCS installation, or specified using the **Security > Server >System Identity Setup** option on the CiscoWorks Home Page. Note that there is also a Secret User associated with this Secret Key.

How DCR uses these security keys varies with the kind of communication taking place:

- **Master Notifications to Slaves**: These notifications receive no extra security. A DCR Master calls Slave APIs only to notify its Slaves that changes have occurred in its repository. The notifications do not contain credentials data or attributes other than the internal DCR Device IDs. Slaves must call the Master to get the updated attribute or credentials data.

- **Slave Calls to Master**: These occur during synchronization, in response to a notification from a Master, and when applications use a Slave to add or update devices and credentials. All such Slave requests must pass either the Secret Key or Password to authenticate, and the User Name to authorize, access to the Master. Note that third-party applications calling a DCR Master remotely via northbound API may pass the Secret Key only.
- **DCR Events to Applications**: These events receive no extra security. DCR sends events to applications only to notify them that devices or credentials data in its repository has been updated. The events do not contain credentials, or anything but basic device identifiers. Applications must call DCR to receive the updated device or credentials data.
- **Application Calls to DCR**: Applications on the same server as the called DCR use the DCR local APIs. DCR assumes the client was successfully authenticated before the call, and requires only the user name to authorize access. Applications residing on a remote server can call DCR via northbound APIs, but must provide the user name and password. DCR authenticates and authorizes using the user name and password.

About DCR Data Storage

The following topics explain how DCR stores device lists, their attributes and credentials:

- About the DCR Device ID
- How DCR Stores Attributes
- How DCR Stores Credentials
- How DCR Stores Proxy Device Data
- How DCR Stores Enable-Mode Passwords
- About User-Defined Fields

Note DCR and its APIs do not communicate with devices directly. The code that interacts directly with devices and fetches their data for storage in DCR is the responsibility of the application.

About the DCR Device ID

For every device, DCR maintains an internally generated, unique, sequential number called the DCR Device ID. This ID identifies the device's record in the DCR database. Your application must use this ID when communicating with DCR about a specific device and its data. DCR does not re-use deleted DCR Device IDs. Whenever an administrator deletes a DCR device using the DCA GUI or the DCR CLI, all information for that device, including the DCR Device ID, is removed from DCR.

Be sure not to confuse the DCR Device ID with other attributes used to identify the device in the network, such as the host_name or sysObjectID (for a complete list of these other attributes, see the "How DCR Stores Attributes" section on page 14-10).

If your application maintains its own set of device identifiers, you will need to create and maintain tables that map your application's device identifiers to the DCR Device IDs (see the "Guidelines for DCR Application Development" section on page 14-38).

How DCR Stores Attributes

In DCR, an attribute is all device data other than credentials (see the "How DCR Stores Credentials" section on page 14-10). Attribute data consists of displayable strings of printable ASCII characters. Attributes are considered unique to each device (e.g., host_name and management_ip_address).

DCR stores values for the attributes shown in Table 14-1 for all standard devices. Some proxy devices store additional attributes (see the "Integrating DCR with OGS" section on page 14-13)

Table 14-1 DCR Device Standard Attributes

Attribute	Description
host_name	Device host name.
domain_name	Domain name of the device.
management_ip_address	The IP address used to access the device. Both IPv4 and IPv6 address types are supported. Required if host_name is not specified.
display_name	The name the user wants the device to have in reports or graphical displays. This value can be derived from host_name or management_ip_address. Required.
sysObjectID[1]	A string with the sysObjectID value. This attribute is required, but DCR will fill in the value automatically if the mdf_type is specified.
mdf_type[1]	A normative name for the device type, taken from Cisco's Meta Data Framework (MDF) database. This attribute is required, but DCR will fill in the value automatically if the sysObjectID is specified.
dcr_device_type[2]	A name for the device type to be used in DCR. It has meaning only in the DCR context. This is not the actual network device attribute.

1. You cannot leave both of these attributes blank, but you can specify either one or both of them as "unknown". DCR will determine the sysObjectID value automatically if only the mdf_type is specified, and vice versa. If you enter both, DCR will *not* check that they map properly.
2. You must specify a value for dcr_device_type when adding a new device. The dcr_device_type and DCR Device Category are different terms for the same construct (e.g., the value of a device's dcr_device_type is set when you call the SetDCRDeviceCategory method on that device).

If for any of the attributes, the value is modified to an empty string, the attribute itself will not be disassociated from the corresponding device. Instead, the attribute will be stored with an empty value. `If a device is added in DCR without an IP address, then DCR returns null for the attribute, management_ip_address. But if IP address is removed ( ie, set to empty ) from an existing DCR device, then DCR returns empty string.`

How DCR Stores Credentials

DCR credentials are the attribute values that applications use to access and operate on devices. For example, if the device has SNMP enabled, the SNMP read-only and read-write community strings used to access that device are considered DCR credentials. All credentials are encrypted and not displayable in encrypted form.

DCR associates the credentials shown in Table 14-2 with all standard devices, DSBU Clusters and DSBU Cluster Members. Some proxy devices use a different mix of credentials (see the "How DCR Stores Proxy Device Data" section on page 14-11).

Table 14-2 DCR Device Standard Credentials

Credential	Description
primary_username	The primary user name used to access the device.
primary_password	The password for the primary_username.
primary_enable_password	The device's primary "enable password" or "enable secret" password. See the "How DCR Stores Enable-Mode Passwords" section on page 14-13.
snmp_v2_ro_comm_string	The device's SNMP V2 read-only community string.
snmp_v2_rw_comm_string	The device's SNMP V2 read/write community string.
snmp_v3_user_id	The device's SNMP V3 user ID.
snmp_v3_password	The device's SNMP V3 password.
snmp_v3_engine_ID	The device's SNMP V3 engine ID.
snmp_v3_auth_algorithm	The SNMP V3 authorization algorithm used on the device (i.e., MD5 or SHA-1).
rxboot_mode_username	The device's diagnostic user ID.
rxboot_mode_password	The device's diagnostic password.

The following credentials and attributes are added as part of Common Services 3.0 Service Pack 1 (CS3.0 SP1)

Table 14-3 Credential and Attributes Added in CS 3.0 SP1

Credential/Attribute	Description
http_username	HTTP Username
http_password	HTTP password
http_port	HTTP Port
https_port	HTTPS Port
http_mode	Current transfer mode (http or https)
cert_common_name	Common name attribute value in the server's certificate

How DCR Stores Proxy Device Data

For standard network devices, DCR stores the attributes and credentials listed in the "How DCR Stores Attributes" section on page 14-10 and in the "How DCR Stores Credentials" section on page 14-10. Some proxy devices use different attribute and credential schemes. DCR supports these devices with attributes and credentials appropriate to them, as follows:

- **DSBU Clusters:** Each DSBU cluster has its own database record, which represents a "logical" DSBU cluster. This record does not point to any one physical device in the cluster, and stores the credentials of the DSBU Commander. For DSBU Clusters, DCR stores the attributes and credentials used for standard devices.

- **DSBU Cluster Members:** Each DSBU Cluster Member has its own host_name, sysObjectID, and mdf_type value, and uses the same credentials as the DSBU Cluster. For DSBU Cluster Members, DCR stores the same attributes and credentials as for standard devices, plus the credentials shown in Table 14-4.

Table 14-4 Additional DSBU Cluster Member Credentials

Credential	Description
dsbu_member_number	The number of the DSBU Cluster member. This number represents the order in which the device was added to the cluster.
parent_dsbu_id	The DCR Device ID of the parent DSBU Cluster device.

- **AUS Devices:** For the AUS device itself, DCR stores the same attribute data used for standard devices, but does not use standard credentials. Instead, it stores the credentials shown in Table 14-5.

Table 14-5 AUS-Specific Credentials

Credential	Description
aus_url	**The URL for the AUS device.**
aus_port	The port number of the AUS service running on the AUS device.
aus_username	**The user login providing access to the AUS device.**
aus_password	**The password for the corresponding aus_username.**

- **AUS- Managed Devices**: For each device managed by an AUS device, DCR stores the same attributes as for standard devices, plus the additional device_identity attribute. It does not use standard credentials; instead, it stores the credentials shown in Table 14-6.

Table 14-6 AUS-Managed Device: Additional Attribute and Device-Specific Credentials

Attribute/Credential	Description
device_identity	**This attribute is a string uniquely identifying the AUS-managed device.**
aus_username	**The user login providing access to the AUS-managed device.**
aus_password	**The password for the corresponding aus_username.**
parent_aus_id	The DCR Device ID of the managing AUS device.

The following are supported from CS3.0 SP1.

- CNS Configuration Engine(CNS Server):

  For the CNS Server, DCR stores the same attribute data used for standard devices.

- CNS Managed Devices:

  For the CNS Server, DCR stores the same attribute data used for standard devices. In addition to this, it will have the parent_cns_id attribute

Table 14-7 CNS Managed Devices Specific Device Credentials

Attribute/Credential	Description
parent_cns_id	**Device ID of the parent CNS server (CNS Configuration Engine)**
cns_config_id	**CNS Config ID of the device.**
cns_event_id	**CNS Event ID of the device.**
cns_image_id	CNS Image ID of the device.

How DCR Stores Enable-Mode Passwords

Most managed devices feature “enable password” and “enable secret” commands. Both permit an Enable Mode password to be set and stored within the configuration for a managed device. Only administrators who know this password can change the device configuration. The “enable secret” command adds an extra layer of protection by encrypting the stored password, preventing anyone from discovering the password by examining the device configuration or a copy of it.

DCR retains only one Enable Mode password. If the Enable Mode password was set using the “enable password” command only, then DCR will store only that password. If an “enable secret” command has been used as well as (or instead of) the “enable password” command, the “enable secret” password will take precedence.

If you are importing device data from a product that does not use DCR, you may find a user database where both the “enable password” and “enable secret” have been set. In this case, the “enable password” should be discarded, and the “enable secret” password moved into the new DCR database. If only one of the values is populated in the source database, then only that value should be migrated.

About User-Defined Fields

The DCR database allows users to specify up to 10 user-defined fields. These fields, once specified by a user, become part of the device record structure and their values are stored as part of device records. While application developers can access data in user-defined fields for reporting and other purposes, they cannot add or change user-defined fields programmatically. User-defined fields are for users only, and must be added using the DCA GUI only.

Integrating DCR with OGS

To enable operations on groups of devices, DCR provides the following pre-defined groups:

- System Defined: Includes all devices, arranged by mdf_type value.
- User Defined: Includes all devices, arranged by user-specified attribute values.

OGS (see Chapter 30, “Using Object Grouping Services”) incorporates these DCR Groups and provides the underlying shared and secured device grouping services to DCR.

The Device Selector provided with the CiscoWorks Home Page (see Chapter 7, “Using the CiscoWorks Home Page”) performs basic filtering of devices based on IP address, host name, display name, user-defined attributes, etc. Once the DCR filtering is done, the DCR Device Selector will show the filtered device list within the predefined DCR Groups provided by OGS.

Integrating DCR with ACS

When DCR starts and the CWCS Server is in ACS mode, DCR will register all known devices in its repository with CAM. This registration process enables DCR APIs to authorize the devices for the user appropriately. DCR also registers devices with CAM whenever device(s) are added, as part of the "add" operation, and updates CAM appropriately whenever DCR devices are modified.

Each DCR device is mapped to an ACS device based on the device's IP address in both DCR and ACS. If a DCR device has no IP address, then its DCR display name is mapped to an ACS host name.

The DCR APIs authorize the user at the task level at all times, using the DCR and ACS Task IDs shown in Table 14-8 (this table ignores DCR-related ACS tasks that cannot be triggered by DCR API calls). See Table 14-9 for the list of default DCR Task-to-Role mappings; this table includes some DCR-related ACS tasks that can only be triggered using the DCR GUI.

Device-level authorizations are applied only when the CWCS Server is in ACS mode. Note that an application running on the same server as the CWCS Server can disable task- and device-level authorization for local DCR access. If your application uses the APIs to perform DCR tasks, and authorizes their use by particular users based on role, be sure that your application-level authorizations match those used by the DCR API tasks.

Table 14-8 DCR APIs and Associated Tasks

DCR API	DCR Task	ACS Task
addDevice, addDevices	ADD_DEVICE_TASK	Add
getDCRDomainID	None	None
getDCRID	None	None
getDCRUpdates	VIEW_CREDENTIAL_TASK	View
getDeletedDevices(DeviceId[] managedDeviceids, APIExtraInfo apiExtraInfo) throws DCRException // all devices	VIEW_CREDENTIAL_TASK	View
public synchronized Device[] getDeletedDevices(long transactionId,APIExtraInfo apiExtraInfo) // all devices	VIEW_CREDENTIAL_TASK Deprecated. Do not use this API.	View
getDevice, getDevices	VIEW_CREDENTIAL_TASK	View
getDeviceIdentifiers	VIEW_DEVICE_TASK	View Devices
getDevices(DeviceId [] id, APIExtraInfo apiExtraInfo)	VIEW_CREDENTIAL_TASK	View
getDevices(DeviceId[] id, String[] attList, APIExtraInfo apiExtraInfo)	VIEW_CREDENTIAL_TASK	View
Device[] getDevices(long timeStamp, APIExtraInfo apiExtraInfo) throws DCRException // all devices	VIEW_CREDENTIAL_TASK	View
getIdentityAttributes	VIEW_DEVICE_TASK	View Devices
getMasterDCRID	None	None
getMatchingDevices	VIEW_DEVICE_TASK	View Devices
getMaxTransactionID	None	None

Table 14-8 DCR APIs and Associated Tasks (continued)

DCR API	DCR Task	ACS Task
getMissingDCRDevicesinACS	VIEW_DEVICE_TASK	View Devices
getNewDevices	VIEW_CREDENTIAL_TASK	View
getUnAuthorizedDevices	VIEW_DEVICE_TASK	View Devices
getUpdatedDevices	VIEW_CREDENTIAL_TASK	View
isMasterDCR	None	None
isMasterDCRRunning	None	None
isRunning	None	None
registerForHTTPEvents	REG_APP_TASK	Register/Unregister 3rd Party Application in DCR
unregisterForHTTPEvents	REG_APP_TASK	Register/Unregister 3rd Party Application in DCR
updateDevice, updateDevices	UPDATE_DEVICE_TASK	Edit

Table 14-9 DCR Task-to-Role Mapping

Task/Role	Help Desk	Approver	Network Operator	Network Administrator	System Administrator
Add			X	X	X
Edit				X	X
Delete				X	X
Reports	X	X	X	X	X
View		X		X	X
View Devices	X	X	X	X	X
Change Mode				X	X
Add User Defined Fields in DCR				X	X
Modify User Defined Fields in DCR				X	X
Delete User Defined Fields from DCR				X	X
Export				X	X
Bulk Import				X	X

About DCR Events

DCR uses CWCS Event Services Software (ESS) (see Chapter 19, “Using Event Services Software”) to broadcast events to all applications on the same CWCS Server. DCR clients listening to these events can choose to filter out unwanted data or events of no interest (e.g., events for classes of devices the application does not support). When filtering data, applications can query DCR for additional device details using the DCR APIs.

Applications that cannot use ESS can receive DCR events via HTTP. An application wanting to use this alternative method must first use the DCR API to register its HTTP URL with DCR. DCR will then use this URL to send events to the non-ESS application.

The DCR Server never listens to events broadcast by applications.

DCR generates:

- **Device events** whenever there is a change in the attributes or credentials of devices stored in the DCR repository (e.g, device additions, imports, etc.) For details, see the "About DCR Device Events" section on page 14-17.
- **Process events** when the DCR Server starts, stops, or changes modes. For details, see the "About DCR Process Events" section on page 14-19.
- **Restore events** whenever DCR data is restored from backup. For details, see the "About DCR Restore Events" section on page 14-21.

The normal process of database backup and restore can also generate combinations of these events and accompanying notifications. For a summary of these, see the "About DCR Events During Backup and Restore" section on page 14-22.

In Master/Slave implementations, the DCR Notifier generates synchronization and update notifications, so DCR Slaves can get device updates from Masters. These notifications are not broadcast to DCR client applications. The events include all significant repository changes, such as device additions, deletions, credential updates, etc.

Note that events are not the same as notifications sent from a DCR Master to a Slave (see the "About the DCR Components" section on page 14-5). With DCR Servers in Standalone mode, events are generated directly and sent to applications who use that DCR. When a DCR Master's devices are updated, it generates events directly to applications using it, and then sends notifications to its Slaves. These Slaves, in turn, get their updates from the Master, and then generate events for the applications using the Slaves.

The details of event generation for DCR Slaves are the same as in DCR Master. From the application's perspective:

- It will receive events from DCR when there is an update in DCR. The events can be for new, existing, or deleted devices in DCR.
- The application can fetch device lists and credentials data from DCR as needed.

When a new application (that does not contain any device credential data) is installed, the application can get the device list available from DCR and present the devices in a pick list. The user can then select the devices from the list for management.

About the DCR Domain ID and Transaction ID

The Master DCR Server in every Master/Slave DCR setup has its own DCR Domain ID. The DCR Domain ID identifies the logical DCR domain in which the Master and its Slaves participate. Your application must:

- Maintain the DCR Domain ID as part of its data.
- Check to see if the DCR Domain ID has changed:
  - At application startup. You can do this by calling getDCRDomainID() during startup.
  - Whenever the application receives a DCR_DATA_RESTORED or DCR_DATA_RESTORED_FROM_DIFFERENT_DOMAIN event. You can do this by comparing the DCR Domain ID contained in the event with the DCR Domain ID stored in the application database.

- If the DCR Domain ID has changed with a DCR_DATA_RESTORED event, update the DCR Domain ID in the application database.
- If the DCR Domain ID has changed with a DCR_DATA_RESTORED_FROM_DIFFERENT_DOMAIN event:
  - Update the DCR Domain ID in the application database.
  - Clean up and refresh the device list.

Every DCR Server update of any kind (e.g., device additions, deletions, updates, etc.) has a transaction ID, which is simply a serial number. The DCR Transaction ID is the ID of the last transaction (e.g., device update, deletion, addition, etc.) an application conducted with its DCR Server. This transaction ID is maintained in the application database, and allows the application to determine whether updates occurred while the application was offline. To make use of it, your application must

- Maintain the DCR Transaction ID in its own database.
- At application startup, check to see if the last (or maximum) transaction ID recorded on the DCR Server is greater than the DCR Transaction ID recorded in the application database. You can get the last transaction ID for the Server by calling getMaxDcrTransactionId() during startup.
- If the transaction ID on the server is higher than the DCR Transaction ID stored in the application:
  - Update the DCR Transaction ID in the application database.
  - Get all the device updates since the last transaction recorded in the application.

For examples of code making use of the Domain ID and the Transaction ID, see the "Using DCR Domain and Transaction IDs" section on page 14-44.

About DCR Device Events

In addition to events for DCR changes (see the "About DCR Process Events" section on page 14-19) and data restores (see the "About DCR Restore Events" section on page 14-21), DCR broadcasts events whenever devices are updated (including addition or deletion).

A DCR device event contains the event's:

- Subject (also treated as the event ID). For example: `Event Subject Name = "cisco.mgmt.cw.cmf.dcr";`.
- Data, which consists of:
  - The type of device change. For example: DEVICE_ADDED.
  - Identifying details for the device(s) affected by the event. For example: the Device ID, sysObjectID, etc.
  - The event Timestamp.

Table 14-10 shows the data for all types of device events. Applications can use this data to decide whether to process the event or not, and to query DCR for details about the target device(s).

Table 14-10 DCR Device Update Event Data

Device Change	Event Data
Added	`<DCREvent> <EventType>DEVICES_ADDED</EventType> <EventSource> <AppName>DCRUI</AppName> <AppVersion>1.0</AppVersion> <AppHostName>server1</AppHostName> </EventSource> <Devices> <Device> <DeviceId>14</DeviceId> <SysObjectId>222</SysObjectId> <IpAddress>sds</IpAddress> <HostName>null</HostName> <DisplayName>sd44s</DisplayName> <TransactionId>556598</TransactionId> <MDFID>12345</MDFID> </Device> </Devices> </DCREvent>`
Added in Bulk	`<DCREvent> <EventType>BULK_DEVICES_ADDED</EventType> <EventSource> <AppName>DCRUI</AppName> <AppVersion>1.0</AppVersion> <AppHostName>server1</AppHostName> </EventSource> </DCREvent>`
Deleted	`<DCREvent> <EventType>DEVICES_DELETED</EventType> <EventSource> <AppName>DCRUI</AppName> <AppVersion>1.0</AppVersion> <AppHostName>server1</AppHostName> </EventSource> <Devices> <Device> <DeviceId>14</DeviceId> <SysObjectId>222</SysObjectId> <IpAddress>sds</IpAddress> <HostName>null</HostName> <DisplayName>sd44s</DisplayName> <TransactionId>null</TransactionId> <MDFID>12345</MDFID> </Device> </Devices> </DCREvent>`

Table 14-10 DCR Device Update Event Data (continued)

Device Change	Event Data

Deleted in Bulk	`<DCREvent>`
	`   <EventType>BULK_DEVICES_DELETED</EventType>`
	`      <EventSource>`
	`         <AppName>DCRUI</AppName>`
	`         <AppVersion>1.0</AppVersion>`
	`         <AppHostName>server1</AppHostName>`
	`      </EventSource>`
	`</DCREvent>`

Updated	`<DCREvent>`
	`   <EventType>DEVICES_UPDATED</EventType>`
	`      <EventSource>`
	`         <AppName>DCRUI</AppName>`
	`         <AppVersion>1.0</AppVersion>`
	`         <AppHostName>server1</AppHostName>`
	`      </EventSource>`
	`   <Devices>`
	`      <Device>`
	`         <DeviceId>14</DeviceId>`
	`         <SysObjectId>222</SysObjectId>`
	`         <IpAddress>sds</IpAddress>`
	`         <HostName>null</HostName>`
	`         <DisplayName>sd44s</DisplayName>`
	`         <TransactionId>null</TransactionId>`
	`         <MDFID>12345</MDFID>`
	`      </Device>`
	`   </Devices>`
	`</DCREvent>`

Updated in Bulk	`<DCREvent>`
	`   <EventType>BULK_DEVICES_UPDATED</EventType>`
	`      <EventSource>`
	`         <AppName>DCRUI</AppName>`
	`         <AppVersion>1.0</AppVersion>`
	`         <AppHostName>server1</AppHostName>`
	`      </EventSource>`
	`</DCREvent>`

About DCR Process Events

DCR broadcasts process events whenever the DCR Server starts, which is a common event. It also broadcasts events when the server changes modes, which happens much less often. A DCR process event consists of the event's:

- Subject (also treated as the event ID).

  For example: `Event Subject Name = "cisco.mgmt.cw.cmf.dcr";`.
- Data, which consists of:
  - The type of change. For example: MASTER_CHANGED_TO_SLAVE.
  - Identifying details for the DCR affected by the event. For example: "DCR Server 1".

Example 14-1 shows the event data for the only normal process event, a DCR Server start. Table 14-11 shows event data for the uncommon DCR mode-change process events. All of these events are broadcast because applications (such as OGS Server) need to know about them. Most applications can ignore process events *unless* they are also accompanied by a "data restored" event, in which case applications *must* receive and process them (see the "About DCR Restore Events" section on page 14-21).

Example 14-1 DCR Server Start Event Data (continued)

```
<DCREvent>
   <EventType>DCR_SERVER_START</EventType>
   <EventSource>
      <AppName>DCR Server</AppName>
      <AppVersion>1.0</AppVersion>
      <AppHostName>server1</AppHostName>
   </EventSource>
</DCREvent>
```

Table 14-11 DCR Mode-Change Event Data

Mode Change	Event Data

Standalone to Master	`<DCREvent>`
	`   <DCRDomainID>Group123456</DCRDomainID>`
	`   <EventType>STANDALONE_CHANGED_TO_MASTER</EventType>`
	`   <EventSource>`
	`      <AppName>DCR Server</AppName>`
	`      <AppVersion>1.0</AppVersion>`
	`      <AppHostName>server1</AppHostName>`
	`   </EventSource>`
	`</DCREvent>`

Standalone to Slave	`<DCREvent>`
	`   <DCRDomainID>Group123456</DCRDomainID>`
	`   <EventType> STANDALONE_CHANGED_TO_SLAVE</EventType>`
	`   <EventSource>`
	`      <AppName>DCR Server</AppName>`
	`      <AppVersion>1.0</AppVersion>`
	`      <AppHostName>server1</AppHostName>`
	`   </EventSource>`
	`</DCREvent>`

Slave to Master	`<DCREvent>`
	`   <DCRDomainID>Group123456</DCRDomainID>`
	`   <EventType> SLAVE_CHANGED_TO_MASTER</EventType>`
	`   <EventSource>`
	`      <AppName>DCR Server</AppName>`
	`      <AppVersion>1.0</AppVersion>`
	`      <AppHostName>server1</AppHostName>`
	`   </EventSource>`
	`</DCREvent>`

Table 14-11 DCR Mode-Change Event Data (continued)

Mode Change	Event Data

Slave to Standalone	`<DCREvent>`
	`   <DCRDomainID>Group123456</DCRDomainID>`
	`   <EventType>SLAVE_CHANGED_TO_STANDALONE</EventType>`
	`   <EventSource>`
	`      <AppName>DCR Server</AppName>`
	`      <AppVersion>1.0</AppVersion>`
	`      <AppHostName>server1</AppHostName>`
	`   </EventSource>`
	`</DCREvent>`

Master to Slave	`<DCREvent>`
	`   <DCRDomainID>Group123456</DCRDomainID>`
	`   <EventType>MASTER_CHANGED_TO_SLAVE </EventType>`
	`   <EventSource>`
	`      <AppName>DCR Server</AppName>`
	`      <AppVersion>1.0</AppVersion>`
	`      <AppHostName>server1</AppHostName>`
	`   </EventSource>`
	`</DCREvent>`

Master to Standalone	`<DCREvent>`
	`   <DCRDomainID>Group123456</DCRDomainID>`
	`   <EventType>MASTER_CHANGED_TO_STANDALONE </EventType>`
	`   <EventSource>`
	`      <AppName>DCR Server</AppName>`
	`      <AppVersion>1.0</AppVersion>`
	`      <AppHostName>server1</AppHostName>`
	`   </EventSource>`
	`</DCREvent>`

Slave to New Master	`<DCREvent>`
	`   <DCRDomainID>Group123456</DCRDomainID>`
	`   <EventType>SLAVE_CHANGED_TO_NEW_MASTER</EventType>`
	`   <EventSource>`
	`      <AppName>DCR Server</AppName>`
	`      <AppVersion>1.0</AppVersion>`
	`      <AppHostName>server1</AppHostName>`
	`   </EventSource>`
	`</DCREvent>`

About DCR Restore Events

DCR sends the restore events shown in Table 14-12 when DCR data is changed due to specific DCR mode changes or a data restore from backup. The restore events are sent in different conditions:

- DCR DATA RESTORED FROM DIFFERENT DCR DOMAIN: DCR will send this event when:
  - A DCR Server in Standalone or Master mode changes to Slave mode. The new Slave-mode DCR Server receives this event before it begins receiving DCR DATA RESTORED events.
  - A DCR Server in Slave mode is reassigned to a DCR Master in a different domain.
  - A DCR Server in Slave mode has its DCR Master change its host name.
  - A CWCS restore is performed and the restore contains different domain data.

  Applications receiving this event must:
  - Alert the application administrator to act appropriately.
  - Clean up the application-managed device list.
  - Call getDCRUpdates() to refresh the application device list.

- DCR DATA RESTORED: DCR will send this event when:
  - A DCR Server in Standalone or Master mode changes to Slave mode. The new Slave-mode DCR Server receives this event after the DCR DATA RESTORED FROM DIFFERENT DCR DOMAIN event, and continues to receive it until the Slave-mode Server syncs with its new Master.
  - When a DCR Server in Master mode changes its port and the user re-registers with a Slave with this Master.
  - When a DCR Server in Slave mode changes its port and the user re-registers with this Slave with its Master.
  - When DCR Server in Slave mode changes to Standalone or Master mode.

  Applications receiving this event are expected to call getDCRUpdates() to get all new, updated, or deleted devices from DCR.

Table 14-12 DCR Restore Events

Restore Event	Event Data

Data restored from different domain	`<DCREvent>`
	`<DCRDomainID>Group123456</DCRDomainID>`
	`<EventType>DCR_DATA_RESTORED_FROM_DIFFERENT_DCR_DOMAIN`
	`   </EventType>`
	`   <EventSource>`
	`      <AppName>DCR Server</AppName>`
	`      <AppVersion>1.0</AppVersion>`
	`      <AppHostName>server1</AppHostName>`
	`   </EventSource>`
	`</DCREvent>`

Data restored from same domain	`<DCREvent>`
	`<DCRDomainID>Group123456</DCRDomainID>`
	`<EventType>DCR_DATA_ RESTORED</EventType>`
	`   <EventSource>`
	`      <AppName>DCR Server</AppName>`
	`      <AppVersion>1.0</AppVersion>`
	`      <AppHostName>server1</AppHostName>`
	`   </EventSource>`
	`</DCREvent>`

About DCR Events During Backup and Restore

DCR uses the CWCS backup and restore framework (see the Chapter 12, "Using Backup and Restore") to protect the DCR data stored in the main CWCS database and the DCR configuration file.

Data changes are a normal part of any restore from a backup. However, because DCR is a distributed system with varying modes, it is also possible for any restored DCR to:

- Change modes (see the "How DCR Masters and Slaves Interact" section on page 14-6). For example, a Standalone DCR can be set after a backup to act as a Slave. When the restore is performed, it will be reset to Standalone mode.
- Change master/slave relationships. For example: A DCR Slave may be using Master A at the time a backup is taken. Later, the domain will be changed to use Master B, and the Slave reset to use Master B. When the restore is performed, the Slave will attempt to use Master A.

- Change management domains. For example: A DCR Slave using a Master in domain A can be reset to become a Slave of a Master in domain B. When the restore is performed, the Slave will attempt to replicate with the Master in domain A.

Because any restore can involve a combination of data, mode, master/slave and domain changes, DCR provides special processing functions to handle:

- Domain Changes: DCRs running in Master or Slave mode always have an associated DCR Group ID that indicates the Server's management domain. This Group ID is generated when a DCR is set to Master mode, and communicated to all Slaves later assigned to that Master. The restore process checks the DCR Group ID for appropriate action. If the Group ID changes due to the restore, then DCR broadcasts a "DCR DATA RESTORED FROM DIFFERENT DCR DOMAIN" event. If the DCR Group ID does not change, DCR assumes that no domain change took place, and broadcasts a "DCR DATA RESTORED" event.
- Mode and Master/Slave Changes: These vary according to the individual DCR's working or pre-restore mode (the mode at the time the restore process is started), its end or post-restore mode (the mode after the restore is finished), and its existing Master/Slave relationships. Mode changes for DCR Servers in Master mode also generate "DCR MASTER MODE CHANGE" events. Table 14-13 summarizes all possible mode and master/slave changes for restored DCRs and the actions they take in response.

Applications will need to process domain, mode, and master/slave change notifications from DCRs appropriately, as well as processing data change events. This includes:

- Synchronizing with the DCR during application startup.
- Making provision for the fact that, during a restore, the application may be down for a significant period and may not have a chance to process restore-related events.
- Querying DCR, using DCR Device IDs, for information about deleted and updated devices. Applications that persist any device identity attributes must also pass them to DCR so that DCR can identify whether the device information is modified. DCRProxy provides several APIs to detect deleted and updated devices, including getDeletedDevices() and getDCRUpdates() (see the "How DCR Masters and Slaves Interact" section on page 14-6).

Table 14-13 DCR Restore Behavior

If the Post-Restore (End) Mode is:	And the Pre-Restore (Working) Mode was: Standalone	Slave	Master
Standalone	If the DCR Group ID is different, DCR sends a DCR DATA RESTORED FROM DIFFERENT DCR DOMAIN event. If the DCR Group ID is the same, DCR sends a DCR DATA RESTORED event. Applications must process these events and query DCR to get updates.	After restore, the former Slave unregisters from the old Master and deletes old Master information. If the former Slave could not unregister and the old Master sends it notifications, it ignores them (as it is now in Standalone mode). If the DCR Group ID is different, DCR sends a DCR DATA RESTORED FROM DIFFERENT DCR DOMAIN event. If the DCR Group ID is the same, DCR sends a DCR DATA RESTORED event. Applications must process these events and query DCR to get updates.	After restore, the former Master sends DCR MASTER MODE CHANGED events to all its former Slaves. These Slaves delete the Master contact information. If the DCR Group ID is different, DCR sends a DCR DATA RESTORED FROM DIFFERENT DCR DOMAIN event. If the DCR Group ID is the same, DCR sends a DCR DATA RESTORED event. Applications must process these events and query DCR to get updates.
Slave	After restore, the server is in Master mode. We do this to preserve the device context[1]. To finish conversion to Slave mode, use the GUI or CLI commands. If the DCR Group ID is different, DCR sends a DCR DATA RESTORED FROM DIFFERENT DCR DOMAIN event. If the DCR Group ID is the same, DCR sends a DCR DATA RESTORED event. Applications must process these events and query DCR to get updates.	After restore, if the DCR Group ID is different, then the server moves to Standalone mode, and sends a mode change event and a DCR DATA RESTORED FROM DIFFERENT DCR DOMAIN event. If the DCR Group ID is same and Master is different, then the DCR will unregister with the old Master, delete all data it took from the old Master, register and sync with the new Master, and send a "DCR DATA RESTORED" event. Applications must process these events and query DCR to get updates.	After restore, the server is in Slave mode. It sends a DCR MASTER MODE CHANGED event to all former Slaves. These Slaves delete the contact information for the former Master. The new DCR Slave uses the restored Slave configuration information to contact, re-register and sync data with its Master. If the DCR Group ID is different, DCR sends a DCR DATA RESTORED FROM DIFFERENT DCR DOMAIN event. If the DCR Group ID is the same, DCR sends a DCR DATA RESTORED event. Applications must process these events and query DCR to get updates.

Table 14-13 DCR Restore Behavior (continued)

If the Post-Restore (End) Mode is:	And the Pre-Restore (Working) Mode was: Standalone	Slave	Master
Master	After restore, the server is in Master mode. Some Slave information may be invalid[2], so the new DCR Master will delete all Slave information. If the DCR Group ID is different, DCR sends a DCR DATA RESTORED FROM DIFFERENT DCR DOMAIN event. If the DCR Group ID is the same, DCR sends a DCR DATA RESTORED event. Applications must process these events and query DCR to get updates.	After restore, the server is in Master mode. Some Slave information may be invalid[2], so the new Master will delete all Slave information. It also deletes all Master contact information left over from Slave mode. If the DCR Group ID is different, DCR sends a DCR DATA RESTORED FROM DIFFERENT DCR DOMAIN event. If the DCR Group ID is the same, DCR sends a DCR DATA RESTORED event. Applications must process these events and query DCR to get updates.	After restore, the server is still in Master mode. It compares information about its Slaves that comes from the backup with the Slave information from pre-restore operations. It retains Slaves that exist in both and deletes all others. If the DCR Group ID is different, DCR sends a DCR DATA RESTORED FROM DIFFERENT DCR DOMAIN event. If the DCR Group ID is the same, DCR sends a DCR DATA RESTORED event. Applications must process these events and query DCR to get updates.

1. Suppose we have a DCR Server in Standalone mode, and restore it using Slave data, If we convert directly to Slave mode (retain the Slave configuration from the backup) then the context of the devices known to the applications using the Standalone server will be different from the devices in the Slave. Since Slave and Master data are tightly coupled, the risk is high that all the restored Slave data will be overwritten or lost as soon as communications with the Master begin.
2. Some of the Slaves included in the backup information for this Master may have moved to another Master.

Using DCR

The following topics provide guidelines on and examples of how to integrate DCR with your application:

- Getting Started With DCR
- DCR Tasks to Perform During Application Startup
- Using the DCR APIs
- Responding to DCR Events
- Using DCR Domain and Transaction IDs
- Using the DCR Command-Line Interface
- Enhancing DCR Performance

Getting Started With DCR

Use the following steps to create and populate a standalone DCR Server.

Step 1 Install CiscoWorks Common Services (CWCS).

DCR Server is included in this version of CWCS, and will be installed automatically. Upon conclusion of the installation, the CWCS Daemon Manager will start a local DCR Server. The DCR Server will run in Standalone mode, so it will not be part of a management domain or share devices with any other DCR.

If you need to manually control the DCR Server at any time after installation, enter the following at the command line:

- To stop DCR, enter `pdterm DCRServer`
- To start DCR, enter `pdexec DCRServer`

Step 2 Log on to CWCS.

Point your browser to the CWCS installation at http://server_ip_address:1741. Log into CiscoWorks by providing a username and password (the default for both is `admin`).

Step 3 Start populating the local DCR Server by manually entering data for a few devices, as follows:

a. On the Common Services application panel, select **Device & Credentials Admin > Device Management**. CiscoWorks displays the Device Management screen.

b. Click **Add** to begin adding devices. CiscoWorks displays a Device Properties page.

c. Enter the required device Display Name and select the Device Type.

d. Enter one of the following required attributes: IP address, Host Name, or AUS Device ID (only if the device is AUS-managed).

e. Click **Add to list** to add the device to DCR.

f. Repeat steps a through e for the other devices you want to add to DCR.

g. When you have added all the devices you want, click **Next** to specify credentials for them.

h. When you have added all the credentials needed, click **Next** to specify user-defined fields.

i. When you are finished, click **Finish**.

Step 4 If you want a large number of devices, you can use the DCR Export feature to create a CSV file, and edit the CSV file to add all the other devices you want. Then use the Import feature to load the updated CSV file. For example:

a. On the Device Management screen, click **Export** to export to a CSV file the devices you have already added.

b. Select **File**, specify the file name, and then click **Export**.

c. When the export operation is completed, edit the CSV file as needed. You can use an ASCII text editor to do this.

Alternatively, you can use a spreadsheet application to create a CSV file from scratch, using the exported CSV file as a model for your entries. You can then load it using the following steps.

d. On the Device Management screen, click **Import**.

e. Select **File**, specify the file name of the CSV file you edited, and then click **Import**.

DCR Tasks to Perform During Application Startup

Every application using DCR should be sure to complete the following tasks at startup:

Note After start-up, your application should not process DCR events sent before the DCR_SERVER_START event. Ideally, your application should set a Daemon Manager dependency on DCRServer to avoid this.

1. Get the current DCR transaction ID stored in the application database.
2. If the transaction ID is zero, then this is the first time the application has been started. In this case:
   a. Call the getNewDevices() API and get the device list from DCR.
   b. Update the application-managed device list from the DCR device list.
   c. Get the DCR Domain ID and store it in the application database.
   d. Derive the maximum transaction ID from the device list and update it in the application database.
3. If the transaction ID is not zero, fetch the current DCR Domain ID from the DCR Server and compare it with the DCR Domain ID stored in the application database.
4. If the Server and application DCR Domain IDs are different:
   a. If necessary, clean the application-managed device list, or perform any other required application-specific action.
   b. Set the application transaction ID to zero.
   c. Call the getNewDevices() API and get the device list from DCR.
   d. Update the application-managed device list from the DCR device list.
5. If the Server and application DCR Domain IDs are identical:
   a. Call the getDCRUpdates() API and get the device updates from DCR
   b. Update the application-managed devices from the DCR device updates.

For an example of startup code that performs all of these tasks, see the "Using DCR Domain and Transaction IDs" section on page 14-44.

Using the DCR APIs

Once you have set up a working DCR Server as described in the "Getting Started With DCR" section on page 14-26, you can begin using the DCR APIs to interact with it.

To do this, your application must instantiate an object of class DCRProxy. DCRProxy is the main DCR abstraction. It provides the methods you need to access DCR device data, and hides communication and request details.

The following topics provide guidelines and examples for using DCRProxy and other DCR API classes and methods in application development:

- About the DCR APIs
- Creating the DCRProxy Object
- Creating the APIExtraInfo Object
- Adding Devices to DCR

- Updating a DCR Device
- Adding and Updating Devices in Bulk
- Retrieving DCR Device Objects
- Retrieving DCR Devices in Bulk
- Retrieving Data From a Device Object
- Comparing Two Device Objects
- Registering Third-Party Applications with DCR
- Guidelines for DCR Application Development
- DCR Error Codes and Interpretations

About the DCR APIs

In addition to DCRProxy, DCR provides the classes shown in Table 14-14.

Note that:

- All DCR APIs write APIs (e.g., addDevice, updateDevice) use CSTM to access DCR. The DCR read APIs (e.g., getDevice, getDevices) communicate directly with CWCS database processes.
- You cannot delete devices from DCR using API calls. Only the administrator can delete devices, and using the DCA interface from the CiscoWorks Home Page.
- All DCR access involving credentials information must be user-authenticated and task-authorized. To assist with this security processing, pass APIExtraInfo with your DCRProxy method calls.
- All credentials data is encrypted while stored and during transport, and DCRProxy decrypts and parses them before passing them to the calling process.
- The DCR APIs are SOAP-enabled, so any non-Java application can make a request using SOAP.

All DCR classes are installed in *$NMSROOT*/CSCOpx/lib/classpath/com/Cisco/nm/dcr. All of them share the `com.cisco.nm.dcr` Java package name.

For DCR API Javadocs, see the CWCS 3.0 SDK portal.

Table 14-14 DCR API Classes

Class	Description
APIExtraInfo	Used with the classes AppID, SourceContext, and SecurityContext to store information about the application that initiated a DCRProxy API call and the form of authentication and authorization each API call will request.
AppId	Stores the name, version and the host name of the machine running the application that initiated the DCRProxy API call.
Attribute	Abstracts a single device attribute and its value, with a complete set of Get and Set methods.
DCRException	Encapsulates standard error/exception handling for all DCR operations.
DCRReturnValues	Represents the return values for bulk Add and Update operations.
DCRUpdates	Encapsulates new, updated and deleted device information.
Device	Abstracts a single device and all of its attributes. This class has a complete set of Get and Set methods.

Table 14-14 DCR API Classes (continued)

Class	Description
DeviceId	Represents the internal DCR Device ID. Applications use the DCR Device ID in string format.
SecurityContext	Stores the user name, password, Secret Key and Secret User information needed to authenticate or authorize a DCRProxy API call.
SourceContext	Stores information about the application making the DCRProxy API call.

Creating the DCRProxy Object

DCRProxy is the main DCR class. An object of class DCRProxy:

- Represents a DCR Server instance to the application.
- Provides all the methods needed to access and modify data in the DCR Server instance.
- Conceals whether the call is local or remote.
- Provides the CSTM client used to handle communications between DCR and applications.
- Permits registration of HTTP clients.

To start using DCR, your application should instantiate a single DCRProxy object, as shown below:

```
DCRProxy dcrProxy = new DCRProxy() ;
```

Note that creating only a single instance of DCRProxy in the application or session eliminates the overhead required to establish socket-level communication with a local DCR Server.

Also note that DCRProxy uses CSTM classes for logging via log4j. Your application may do so as well. In order for CSTM to log its messages using log4J, either your application or DCRProxy must load the log4J message categories for CSTM (for more on interactions between CSTM and log4j, see the "Controlling CSTM Logging" section on page 31-3).

The log4j framework permits message categories to be loaded only once per JVM, and in one class loader. This requirement applies irrespective of the number of consumers for CSTM classes. For example: Since both the DCRProxy class and the application-related classes that use CSTM will be loaded in one JVM, the log4J categories for CSTM classes must be loaded either by DCRProxy or by your application class – not by both. If both load the log4J categories for CSTM, then the categories loaded last are the only active categories. If this occurs, log4J will throw exceptions whenever an inactive category object is used for logging, and all logged messages may go to different destinations.

To help you avoid these situations, DCRProxy provides an integer argument that controls the loading of log4J categories for DCR and CSTM classes. You specify this argument at instantiation; for example:

```
DCRProxy dcrProxy = new DCRProxy(0) ;
```

Table 14-15 shows the possible values for the argument and when to use them.

Table 14-15 Controlling log4j Category Loading

Use	When You Want DCR Proxy to Load
`DCRProxy(0) ;`	The categories for both CSTM and DCR classes. Use this option if your application does not use or load the log4J categories for CSTM classes.

Table 14-15 Controlling log4j Category Loading

Use	When You Want DCR Proxy to Load
`DCRProxy(1);`	The categories for DCR classes only. It will not load the categories for CSTM classes. Use this option if your application uses CSTM and loads the log4J categoris for CSTM classes.
`DCRProxy(2);`	No categories at all. This option assumes that your application loads the categories for both the DCR and CSTM classes.

Note that `DCRProxy(2)` will not work for any web-based application context that runs under Tomcat. In this case, the application will not be able to load the categories for DCR classes because the DCR classes are loaded via the class loader, which is different from an application context-specific class loader. If your application runs under Tomcat, use either `DCRProxy(0)` or `DCRProxy(1)`, depending on whether or not you are having the application load the log4J categories for CSTM classes.

If you pass any value other than those shown in Table 14-15 to this constructor, DCRProxy will reset the argument value to 0 and load the categories for both CSTM and DCR classes. The default constructor, `DCRProxy()`, assumes the argument value is 0 and loads the categories for both CSTM and DCR classes. Use the default constructor if your application does not use or load the log4J categories for CSTM classes.

Creating the APIExtraInfo Object

Most DCRProxy API calls accept objects of class APIExtraInfo in addition to their regular arguments. APIExtraInfo encapsulates:

- The AppID. This object is the unique ID of your application, and establishes your application name, version number, and host information for use in the SourceContext object.
- The SourceContext. This object establishes your application and its context as the source of the DCRProxy API call.
- The SecurityContext. This object contains the information needed to authenticate or authorize the DCRProxy API request. Applications residing on the same machine as the DCR Server have access to the local version of the DCR APIs, and need pass only the requesting username. Applications on remote servers must use the remote (or “north-bound”) versions of the DCR APIs, and must pass the requesting username and password.

Example 14-2 shows a modifiable code fragment that specifies this information. You will need to change this code before using it (i.e., comment out one of the two SecurityContext alternatives, and choose to pass either the password or secret key value, but not both, for remote API calls).

Example 14-2 Instantiating APIExtraInfo

```
AppId Myapp = new AppId("Device Manager", //unique AppID and application name
               "1.3.2", //application version number
               "192.168.1.15"); // host on which the application is running
SourceContext source = new SourceContext(Myapp) ;
// For Local API calls
SecurityContext security = new SecurityContext("username");

// For North-Bound API calls
SecurityContext security = new SecurityContext("username",
                       "password",
                        "secretKey")
//Wrapper for Source and Security information:
APIExtraInfo extraInfo = new APIExtraInfo(security,
                              source);
```

Adding Devices to DCR

Example 14-3 shows sample code for adding a standard (non-proxy) device to DCR using the addDevice() call.

Example 14-3 Adding a Standard Device

```
Device device = new Device();
device.SetDCRDeviceCategory(Device.STANDARD_DEVICE);
device.SetAttribute("display_name", "Bldg X Floor Y Switch 1");
device.SetAttribute("management_ip_address", "1.2.3.2");
device.SetAttribute("sysObjectID", "1234567");
// set other attributes
DeviceId id = null;
try
{
id = dcrProxy.addDevice(device,extraInfo);
}
catch (DCRException de)
{
    System.out.println("Error in adding Device " +
     de.getMessage());
    // Handle any error from DCR
}
catch (Exception e)
{
    System.out.println("Error in adding Device " +
     e.getMessage());
    // Do application-specific things
}
System.out.println("ID for New Device = "+ id.getValue());
// This is the value that the application will retain
```

:

To add other types of devices using the same call, simply change the SetDCRDeviceCategory and SetAttribute portions of the code as appropriate for each device, as shown in the following examples.

Example 14-4 Adding an AUS Device

```
Device device = new Device();
device.SetDCRDeviceCategory(Device.AUS_DEVICE);
device.SetAttribute("display_name", "AUS 1");
device.SetAttribute("management_ip_address", "1.2.3.2");
device.SetAttribute("sysObjectID", "UNKNOWN");
// set other attributes
DeviceId id = null;
...
```

Example 14-5 Adding an AUS-Managed Device

```
Device device = new Device();
device.SetDCRDeviceCategory(Device.STANDARD_DEVICE);
device.SetAttribute("parent_aus_id", "999999");// Set the parent AUS id
device.SetAttribute("display_name", "AUS Managed Device 1");
device.SetAttribute("management_ip_address", "1.2.3.2");
device.SetAttribute("sysObjectID", "1234567890");
// set other attributes
DeviceId id = null;
...
```

Example 14-6 Adding a DSBU Cluster

```
Device device = new Device();
device.SetDCRDeviceCategory(Device.DSBU_DEVICE);
device.SetAttribute("display_name", "DSBU-Cluster 1");
device.SetAttribute("management_ip_address", "1.2.3.2");
device.SetAttribute("sysObjectID", "UNKNOWN");
device.SetAttribute("mdf_type","278283831");
// set other attributes
DeviceId id = null;
...
```

Example 14-7 Adding a DSBU Cluster Member

```
Device device = new Device();
device.SetDCRDeviceCategory(Device.STANDARD_DEVICE);
device.SetAttribute("parent_dsbu_id", "110");// Where "110" is the parent DSBU Cluster ID
device.SetAttribute("display_name", "DSBU-Cluster Managed Device 1");
device.SetAttribute("management_ip_address", "1.2.3.2");
device.SetAttribute("sysObjectID", "122333");
// set other attributes
DeviceId id = null;
...
```

Example 14-8 Adding a CNS Configuration Engine (CNS Server)

```
Device device = new Device();
device.SetDCRDeviceCategory(Device.CNS_DEVICE);
device.SetAttribute("display_name", "CNS_Server_1");
device.SetAttribute("management_ip_address", "1.2.3.2");
device.SetAttribute("sysObjectID", "UNKNOWN");
device.SetAttribute("mdf_type","277587376");

// set other attributes
DeviceId id = null;
...
```

Example 14-9 Adding a CNS managed device

```
Device device = new Device();
device.SetDCRDeviceCategory(Device.STANDARD_DEVICE);
device.SetAttribute("parent_cns_id", "110");// Where "110" is the parent CNS Server ID
device.SetAttribute("display_name", "CNS_managed_1");
device.SetAttribute("management_ip_address", "1.2.3.2");
device.SetAttribute("sysObjectID", "UNKNOWN");

// set other attributes
DeviceId id = null;
...
```

Updating a DCR Device

You can use code like that shown in Example 14-10 to update attributes or credentials for any existing DCR Device.

Example 14-10 Updating a DCR Device

```
DeviceId deviceID = new DeviceId("<known-device-id>");
Device device = new Device(deviceID);
// Set new attribute values
device.SetAttribute("display_name", "Device 2 New");
// set other attributes


try
{
    dcrProxy.updateDevice(device,extraInfo);
}
catch (DCRException de)
{
    System.out.println("Error in updating Device " +
     de.getMessage());
    // Do application specific things
}
catch (Exception e)
{
    System.out.println("Error in updating Device " +
     e.getMessage());
    // Do application specific things
}
```

Adding and Updating Devices in Bulk

These two APIs let you add or update more than one device object at a time:

- `public DCRReturnValues addDevices(Device[]` *`devices`*`, APIExtraInfo` *`apiExtraInfo`*`) throws DCRException`
- `public DCRReturnValues updateDevices(Device[]` *`devices`*`, APIExtraInfo` *`apiExtraInfo`*`) throws DCRException`

Both APIs return an object of class `DCRReturnValue`, which holds all information about the operation on each device. The information is stored in two arrays: one for the DeviceId objects, and the second for operation error codes. The length of each array is always same as the number of `Device` objects you pass in the API call. The objects in these arrays correspond to each Device object in the *`devices`* array that you pass.

To get this information from the appropriate arrays in DCRReturnValue, use the following methods:

- `public DeviceId getDeviceId(int index)` returns the DeviceId object at the index you pass. Use this method after the `addDevices` call to retrieve and process all the newly created Device IDs.
- `public int getErrorCode(int index)` returns the error code associated with the operation on the Device object at the index you pass.

Example 14-11 shows typical code for a bulk addition. A bulk update would use the different method but have essentially the same structure.

Example 14-11 Adding Devices in Bulk

```
       DCRReturnValues drv = null;

    // Call add API and collect its output in above DCRReturnValues object

…

int nErrorCode;
if(drv != null)
    {
        for(int nLoop = 0; nLoop < numberOfDevicesAdded; nLoop ++)
        {
            // check if device was added successfully
// If return array contains valid DeviceId, this means that device was added successfully
            if(drv.getDeviceId(nLoop) != null) continue; // Valid DeviceId returned

            // Otherwise retrieve error code
nErrorCode = drv.getErrorCode(nLoop);

// The error code represents the ID of DCRException.

// Handle the error here …

        }// end for
    }// end if
```

Retrieving DCR Device Objects

Example 14-12 demonstrates how to retrieve selected DCR Device Objects and their data using a list of DCR Device IDs supplied by an update event.

Example 14-12 Retrieving Device Objects

```
DeviceId[] deviceIDs = <populate device ids...>

String[] requiredAttributes = { "display_name",
                                 "management_ip_address",
                                      "snmp_v2_ro_comm_string"
                                           };

Device[] devices = null;

try
{
    devices = dcrProxy.getDevices(deviceIDs,
                        requiredAttributes,extraInfo);
}
catch (DCRException de)
{
    System.out.println("Error in getting Devices  " +
                e.getMessage());
    // Do application-specific things
}
catch (Exception e)
{
    System.out.println("Error in getting Device " +
     e.getMessage());
    // Do application specific things
}
```

Retrieving DCR Devices in Bulk

Example 14-13 demonstrates how to fetch a large number of DCR Devices at once. This code follows the guidelines given in the "Enhancing DCR Performance" section on page 14-49, specifically those for for h andling groups of devices with more than 5,000 members. Note that the DeviceIdIterator is a class in the package com.cisco.nm.dcr.

Example 14-13 Retrieving Devices in Bulk

```
    DeviceId[] deviceIDs = <the array of DeviceId to be fetched>
    …
    Device[] devices = null;
    Vector v = null;

    try
    {
      DeviceIdIterator it = new DeviceIdIterator(deviceIDs);
      DeviceId[] onlyFewDeviceIDs = null;

      while(it.hasNext())
      {
        onlyFewDeviceIDs = it.next();
        devices = _dcrProxy.getDevices(onlyFewDeviceIDs, attributes, aei);
        if(devices != null)
        {
          if(v == null) v = new Vector();
          v.addAll(Arrays.asList((Object[])devices));
        }
      }// end while
    }
    catch(DCRException ex)
    {
        // Handle exception
    }
    catch(Exception ex)
    {
        // Handle exception
    }

    if(v == null) return null;
    v.trimToSize();
    if(v.size() <= 0) return null;

    devices = new Device[v.size()];
    v.toArray(devices);

    // Here the devices object contain all the devices
```

Retrieving Data From a Device Object

Example 14-14 shows a couple of ways to get specific device data from a device object. This example assumes you have already retrieved one or more device objects using code such as the example shown in the"Retrieving DCR Device Objects" section on page 14-34.

Example 14-14 Retrieving Specific Device Data

```
Device device = <fetched device via DCR API …>
//Get a single Attribute or Credential value
Attribute attr = device.GetAttribute("snmp_v2_ro_comm_string");
// Note that attribute object may be null
if(attr != null)
{
    String strValue = attr.getValue();
    …
}
// Alternate code: Get a single Attribute or Credential value
if(device. isAttributeAvailable("snmp_v2_ro_comm_string"))
{
Attribute attr = device.GetAttribute("snmp_v2_ro_comm_string");
    String strValue = attr.getValue();
    …
}
//Alternate code: Get all attributes for each device
Iterator it = device.GetAttributes();
//Then use the it iterator to cycle through all the attributes for the current device
```

Two methods you can use for retrieving specific device data will return special codes:

- `public int GetDCRDeviceCategory()` returns the device category of a device added with any valid value for dcr_device_type. The possible return values are:
  - 0 for a standard device
  - 1 for a Logical DSBU Cluster
  - 3 for an Auto Update Server (AUS)
  - 4 for a CNS Configuration Engine (CNS)
- `public int GetDeviceAccessType()` returns an integer indicating how the device is or should be accessed. The possible return values are:

- 0 means no access type, or the device is a standard device, or the access information is not available.
- 1 means the device is part of a DSBU cluster and can be accessed as a member of a DSBU cluster. Such device objects will also have a 'parent_dsbu_id' attribute that is the Device ID of a logical DSBU cluster in DCR.
- 3 means the device is accessed via AUS. Such device objects will also have a parent_aus_id attribute that is device ID of a Auto Update Server in DCR.
- 4 means the device is accessed via CNS. Such device objects will also have a parent_cns_id attribute that is device ID of a CNS Server in DCR.

Note that this method assumes that complete device information (specifically, the parent DSBU and AUS ID attributes) have been fetched from DCR. If not, this method will always return 0.

Comparing Two Device Objects

The following code fragment shows how to compare two device objects. Note that two DCR devices are equal if their DCR Device ID values are the same and their attributes and attribute values are the same.

```
DeviceId id = device.GetID();
device.equals(device2Object);
```

Registering Third-Party Applications with DCR

Although CWCS-based applications use ESS to get events indicating updates to DCR devices, third-party applications must listen for events using HTTP. Example 14-15 shows how to register a third-party application so DCR will send events to it.

Example 14-15 Registering to Receive DCR Events Via HTTP

```
try
{
    dcrProxy. registerForHTTPEvents (appID,
                   "<application-URL-to-receive-events>"
                   extraInfo);
}
catch (DCRException de)
{
    System.out.println("Error in registering app  " +
                e.getMessage());
    // Do application specific things
}
catch (Exception e)
{
    System.out.println("Error in registering app " +
     e.getMessage());
    // Do application specific things
}
```

Guidelines for DCR Application Development

Keep the following guidelines in mind when creating applications that use data from DCR:

- DCR never communicates with devices directly.
- DCRProxy conceals all communications details. You do not need to now the DCR Server modeor any other communications details to use DCRProxy successfully.
- Your application must be listening for DCR events in order to receive them.
- Your application is responsible for filtering events for devices that are not applicable to it. DCR itself performs no filtering.
- When filtering, do not rely solely on the sysObjectID or mdf_type value contained in the event to identify the device for you. Since DCR relies on many applications to populate its device list, these attributes cannot be guaranteed to have values other than “unknown”. Your application should test for situations where these attributes have the value “unknown”, and attempt to identify the device either by calling the device directly or by other means.
- Your application is not allowed to delete the devices in DCR. In Master/Slave setups, these device deletions are propagated to all DCR Slave applications, which can seriously affect the operation of other applications sharing the DCR data.
- Your application may cache device credentials in memory, but for security reasons, should never persist or save credentials to disk.
- Your application should always check the availability of DCR for updates before performing critical operations, such as changing device credentials.
- DCR APIs do not validate the values of attributes and credentials while adding and updating devices.
- You cannot leave both sysObjectID and mdf_type attributes blank, but you can specify either one or both of them as “unknown” DCR will determine the sysObjectID value automatically if only the mdf_type is specified, and vice versa. If you enter both, DCR will *not* check that they map properly.
- If you maintain an application-specific store of device information, including device references and other data, remember:
  - You will need to maintain a mapping between your application-specific device identifiers and the DCR Device IDs. This is required because your application can only fetch device details from DCR by using the DCR Device ID.
  - Your application must maintain a “last updated” timestamp value for each device in its application-specific device store. This is needed so that your application can get device updates from DCR using the getNewDevices() API call.
  - Your application should always filter incoming DCR events before populating the application-specific device list with new entries or presenting it to users.

DCR Error Codes and Interpretations

The DCR error codes and what they stand for are as follows:

Table 14-16 DCR Error Codes and Interpretations

Error Number	Error String	Meaning
0	NO_ERROR	No error.
-1	UNKNOWN_EXCEPTION	Unknown error occured.

Table 14-16 DCR Error Codes and Interpretations (continued)

Error Number	Error String	Meaning
-3	NO_ID_ATTRIBUTES_SPECIFIED	Atleast one mandatory attribute was not specified.
-4	MORE_PARENT_IDS_SPECIFIED	More than one Parent ID was specified.
-5	DEVICE_RECORD_EXISTS	Duplicate device. Record for specified device already exists.
-6	DUPLICATE_DEVICE_NAME	Specified device name already exists.
-7	UNSUPPORTED_ARGUMENT_VALUE	Argument value is not supported.
-8	NO_SUCH_DEVICE	Specified device does not exist in records.
-9	DEVICE_EXCLUDED	Device is marked as excluded.
-10	DEVICE_ID_FORMAT_ERROR	Invalid device ID format.
-11	DEVICE_ID_COUNT_ERROR	Internal error during device ID count.
-12	DUPLICATE_DEVICE_NAME_UNDER_PARENT	Duplicate device ID under the same Parent.
-13	NO_DEVICE_NAME	No device name specified.
-15	DUPLICATE_ATTRIBUTE	User defined field with the same name already exists.
-16	INVALID_APP_ID	Invalid application ID.
-17	APP_ID_COUNT_ERROR	Internal error during application ID count.
-18	DUPLICATE_DEVICE_RECORD	Duplicate record found for this device.
-19	AUTHENTICATION_FAILED	User is not authenticated.
-20	DEVICE_AUTHORIZATION_FAILED	User is not authorized to perform this task on the device.
-21	EXCLUDE_FILE_LOAD_ERROR	Error in loading exclude file. Check the file name and path.
-22	SYSOID_VALUE_ERROR	Must specify sysObjectID/MDF type.
-23	CTM_COMM_ERROR	Error in communicating with DCA Server.
-24	NO_DSBU_MEMBER_NUMBER_SPECIFIED	DSBU Cluster member number not specified.
-25	INVALID_ATTRIBUTE_INFO	Invalid attribute specified.
-26	DUPLICATE_MEMBER_UNDER_PARENT	DSBU Cluster member number already exists.
-27	DCR_MODE_IS_NOT_SLAVE	The DCR mode is not set as Slave.
-28	DCR_MODE_IS_NOT_MASTER	The DCR mode is not set as Master.
-29	MASTER_DATA_RESTORED	Master data is restored.
-30	INVALID_SLAVE	The specified slave is not valid.
-31	NUM_ATTRIBUTE_LIMIT	Attribute limit .
-33	TASK_AUTHORIZATION_FAILED	User is not authorized to perform this task .
-34	CAM_EXCEPTION	Internal error occured during security checks.
-35	AUS_DEVICE_ID_NOT_SPECIFIED	AUS device ID is not specified.
-36	INVALID_ATTR_VALUE	One of the attribute values is invalid.
-37	ERROR_IN_OPENING_EXCLUDE_FILE	Error in opening exclude file. Check file name and path

Table 14-16 DCR Error Codes and Interpretations (continued)

Error Number	Error String	Meaning
-38	ERROR_IN_PARSING_EXCLUDE_FILE	Error in parsing exclude file. Please check the format of the file.
-39	CANNOT_CHANGE_MASTER	Error in configuring the slave because domain IDs of the slave and master are the same. Change mode to Standalone first and then change it to Slave.
-40	CANNOT_CONFIG_MASTER	Error in configuring the slave because domain IDs of the slave and master are the same .
-41	INVALID_DCR_DEVICE_TYPE	The specified device type is not valid.
-42	INVALID_ATTR_VAL_LENGTH	The attribute value length is not valid.
-43	INVALID_CRED_VAL_LENGTH	The crdential value length is not valid.
Database Related		
-101	DATABASE_EXCEPTION	Error in database.
-102	PRIMARY_KEY_NOT_UNIQUE	Primary key specified already exists.
-103	SQL_SYNTAX_ERROR	Syntax error in the SQL statement.
CSTM related		
-201	DCR_SERVER_NOT_RUNNING_ERROR	DCA Server is not running.
-202	CSTM_INTERNAL_ERROR	Internal error in communication channel.
-203	ERROR_IN_COMM_WITH_SERVER	Error in communicating with Peer Server. Ensure that self-signed certificate is generated or copied correctly and System Identity Setup is done correctly.
-211	PEER_DCR_SERVER_NOT_RUNNING_ERROR	Peer DCA Server is not running.
-213	ERROR_IN_COMM_WITH_PEER_SERVER	Error in communicating with Peer Server. Ensure that self-signed certificate is generated or copied correctly and System Identity Setup is done correctly.
-221	PEER_DCR_SECURITY_SERVER_NOT_RUNNING_ERROR	Peer DCR security server is not running.
-223	ERROR_IN_COMM_WITH_PEER_DCR_SECUIRTY_SERVER	Error while communicating with Peer DCR Security Server.

Responding to DCR Events

Example 14-16 shows code for a sample application that can respond to DCR Device and DCR RESTORE Events.

Note This code is incomplete. It is provided as an *illustration* of application-side DCR event processing only.

Example 14-16 Responding to DCR Device and RESTORE Events

```
public class DCREventReceiver {

    // Instantiate Tibco receiver and subscribe to "cisco.mgmt.cw.cmf.dcr" event
    public  DCREventReceiver() {

        try {
            _parser = SAXParserFactory.newInstance().newSAXParser();
            _handler = new SXP();
            log.info("INFO: Creating SAX Parser object");
        } catch (Exception ex) {
            log.fatal("Error creating: " + ex.getMessage());
        }

        try {
            TopicConnectionFactory factory;
            factory = new TopicConnectionFactoryImp();
            _con = factory.createTopicConnection();
            _session = _con.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
        } catch (JMSException e) {
            log.fatal(
              "ERROR: cannot connect to Topic for receiving events.");
            log.fatal(MoMUtils.getStackTrace(e));
            return;
        }

        log.info("Listening on the topic: " + "cisco.mgmt.cw.cmf.dcr");
        try {
            Topic topic = _session.createTopic("cisco.mgmt.cw.cmf.dcr");
            _sub = _session.createSubscriber(topic);
            _sub.setMessageListener(this);
        } catch (JMSException e) {
            log.fatal("ERROR: Cannot create the Topic listener.");
        }
    }

    public void processEvents() {
        try {
            _con.start();
            log.info("INFO: Starting to receive message");
        } catch (JMSException e) {
            log.fatal("ERROR: cannot process events from topic");
            log.fatal(MoMUtils.getStackTrace(e));
        }
    }

    public void onMessage(Message jmsMsg) {
        log.info("Calling on mesage to create a thread.... ");
        try {
            TextMessage text = null;
            if (jmsMsg instanceof TextMessage) {
                text = (TextMessage) jmsMsg;
            } else {
                log.debug("Message is not TextMessage: " + jmsMsg);
                return ;
            }

            String xmlData = text.getText();
            log.info("Inserting event in queue: " + xmlData);
            processMessage(xmlData);
        } catch(Exception exception) {
            log.fatal("Problem in onMessage " + exception);
        }
    }
```

```
// process the Event data
//
void processMessage(String txt) {
    dcrDeviceVec.clear();
    dcrEventType = "";
    byte[] txtBytes = txt.getBytes();
    ByteArrayInputStream bis = new ByteArrayInputStream(txtBytes);
    try {
        _parser.parse(bis, _handler);
    } catch (Exception ex) {
        log.fatal("Error parsing  event data: " + txt
          + ". Reason: " + ex.getMessage());
        return;
    }

    for (int i = 0; i < dcrDeviceVec.size(); i++) {
    // Device data
    }

    log.info("Event type is"+ dcrEventType);
}

class SXP extends DefaultHandler {
    public void startDocument() throws SAXException {
        log.info("SXP: StartDocument method is called");
    }

    public void endDocument() throws SAXException {
        log.info("SXP: EndDocument method is called");
    }

    public void startElement(String namespaceURI, String sName,
      String qName, Attributes attrs) {
        log.info("SXP: StartElement method is called " );
        log.info("SXP: NameSpaceURI is = " + namespaceURI);
        log.info("SXP: simple Name is = " + sName);
        log.info("SXP: Qualified Name is = " + qName);

        for (int i = 0; i < attrs.getLength(); ++i) {
            log.info("SXP: Local Name is = "
              + attrs.getLocalName(i));
            log.info("SXP: QName is = "
              + attrs.getQName(i));
            log.info("SXP: Attribute value is = "
              + attrs.getValue(i));
        }

        if (qName.equalsIgnoreCase("Device")) {
            processDevice = true;
        }
    }

    public void endElement(String namespaceURI, String sName, String qName) {
        log.info("SXP: EndElement method is called " );
        log.info("SXP: NameSpaceURI is = " + namespaceURI);
        log.info("SXP: simple Name is = " + sName);
        log.info("SXP: Qualified Name is = " + qName);

        // for now, ignore bulk events, restore events etc.

        if (dcrEventType.equals("BULK_DEVICES_ADDED") ||
            dcrEventType.equals("BULK_DEVICES_DELETED") ||
            dcrEventType.equals("BULK_DEVICES_UPDATED") ||
            dcrEventType.equals("DCR_DATA_RESTORED_FROM_DIFFERENT_DOMAIN") ||
                dcrEventType.equals("DCR_DATA_RESTORED")) {
                processDevice = false;
                return;
            }

            if (!processDevice)
                return;   // ignore devices we do not want

            for (int i = 0; i < dcrEventFieldNames.length; i++) {
                dcrDeviceFields[i] = "";
                if (qName.equalsIgnoreCase(dcrEventFieldNames[i])) {
                    if (value != null && !value.equals(""))
                        dcrDeviceFields[i] = value.toString();
                }
                if (qName.equalsIgnoreCase("SysObjectId")) {
        // Is the device is supported by the application
        //
                    if (! applicationSupportedType(dcrDeviceFields[i])) {
                        processDevice = false;
                        return;
                    }
                }
                // skip if the device id is not in our database
                // for a delete event.
                if (qName.equalsIgnoreCase("DeviceId")) {
                    if (dcrEventType.equals(DCR_DEVICES_DELETED)) {
                        synchronized(application.deviceIdsHash) {
                            if
(!application.deviceIdsHash.containsKey(dcrDeviceFields[i])) {
                                processDevice = false;
                                return;
                            }
                        }
                    }
                }
            }

            if (qName.equalsIgnoreCase("Device") && processDevice) {
                // store all device attributes in an array
                dcrEventType = dcrDeviceFields[0];
                String dcrDeviceId = dcrDeviceFields[1];
                String sysObjectId = "";
                String ipAddress = "";
                String hostName = "";
                String displayName = "";
                long dcrTransactionId = 0;
                if (!dcrEventType.equals("DEVICES_DELETED")) {
                    // delete event has only dcrId.
                    sysObjectId = dcrDeviceFields[2];
                    ipAddress = dcrDeviceFields[3];
                    hostName = dcrDeviceFields[4];
                    displayName = dcrDeviceFields[5];
                    dcrTransactionId = Long.parseLong(dcrDeviceFields[6]);
                }

        // Application-specific processing here...
            }
        }

        public void characters(char[] ch, int start, int length) {

            log.info("SXP: Characters method is called");
            String s = (new String(ch,start,length) ).trim();
            log.info("SXP: The value is " + s);
            if (value == null) {
                value = new StringBuffer(s);
            } else {
                value.append(s);
            }
        }

    } // End of Inner class - SXP

    //Vector to store the list of device Info
    Vector devDataVec=null;
    StringBuffer value=null;
    String dcrEventType = null;

    // parsing specifc variables
    DefaultHandler _handler = null;
    SAXParser _parser = null;

    public static final Class _Class = DCREventReceiver.class;

    private TopicConnection _con = null;
    private TopicSession _session = null;
    private TopicSubscriber _sub = null;
    private SharedQueue _queue = null;
    Vector dcrDeviceVec = null;
    boolean processDevice = false;
    String[] dcrDeviceFields = null;
    Logger log;
}**
```

Using DCR Domain and Transaction IDs

Example 14-17 shows code for a sample application that checks the Domain and Transaction at startup.

Note This code is incomplete. It is provided as an *illustration* of application-side DCR event processing only.

Example 14-17 Using the DCR Domain and Transaction IDs During Startup

```
//initiate the process of listening for events from the DCR
    dcrEventListner = new DCREventReceiver();
    dcrEventListner.processEvents();

// get the latest transaction ID from the application database
    public long getMaxDcrTransactionId() {

}
// Filter device IDs; get only application-specific device IDs from DCR
public Vector getFilteredDcrDevices(String action, DeviceId[] dcrDevices) {
for (int i = 0; i < dcrDevices.length; i++) {
            DeviceId meDev = dcrDevices[i];
            if (action.equals("DEVICES_DELETED")) {
                // only device ID will be in the event
                String dcrDeviceId = meDev.getValue();
            } else {  // extract more event details from payload
                find out sysObjectId or mdf_type of the device...
```

```
                    find out whether application supports this specific device
                }
            }
                return devicesVec;
            }

        /** this routine reads DCR data upon startup and syncs up data **/
        /** in preparation for startup synching with DCR to catch any offline updates **/

            getDCRUpdatesOnStartup(long MaxDcrTransactionId){

                Device dcrDev = new com.cisco.nm.dcr.Device();
                DeviceId[] ids = null;
                Vector allIdsVec = new Vector();
                Vector newDevicesVec     = null,
                       updatedDevicesVec = null,
                       deletedDevicesVec = null;

                String dcrDomainId = dcr.getDCRDomainID(extraInfo);

            StringoldDcrDomainId = get dcrDomainId from application datatbase

                if (oldDCRDomainId not availble in application database) // first time
                   store dcrDomainId in application database
                    } else {
                         if (!oldDCRDomainId.equals(dcrDomainId)) {
                   clean application database and start resync DCR data or
                   indicate application about the change
                   application specific action
                         }
                    }
                }

                if (appMaxDcrTransactionId == 0) {  //first time startup -
               get devices from DCR through
                   Devices[] devices = getNewDevices(appMaxDcrTransactionId,extraInfo)
                    filteredDevices = getFilteredDcrDevices("DEVICES_ADDED",
        updateInfo.getNewDevices());
                   update application managed list
                   return ;
                }

                // If not, check if DCR transaction ID is different
                // If DCR transaction ID is different, adds/deletes/updates took
                // place when application was offline. Get these changes now.
                // Load existing information from application database.
            //
                long maxDcrTransactionId = dcr.getMaxTransactionID(extraInfo);
                if (maxDcrTransactionId == appMaxDcrTransactionId)
                    return null;
            } else (if appMaxDcrTransactionId < maxDcrTransactionId ) {

                // Otherwise, there were updates when application was offline.
                // Retrieve the current device IDs from application and then start sync.

                    DCRUpdateData updateInfo = dcr.getDCRUpdates(appMaxDcrTransactionId,
                                               dcrDomainId, dcrDeviceIds, extraInfo);
                    newDevicesVec = getFilteredDcrDevices("DEVICES_ADDED",
        updateInfo.getNewDevices());
                    deletedDevicesVec = getFilteredDcrDevices("DEVICES_DELETED",
        updateInfo.getDeletedDevices());
                    updatedDevicesVec = getFilteredDcrDevices("DEVICES_UPDATED",
        updateInfo.getUpdatedDevices());
        // Updated application managed list based on the new, deleted and updated data from DCR.
```

```
}
    }
/************************************************
main method for the application.
************************************************/
public static void main(String[] args) throws Exception {
// initialize first: Start new thread to listen to DCR events, and push to the queue.
startDCREventListener();
        long MaxDcrTransactionId = application.getMaxDcrTransactionId();
// Now start threads to  handle all device processing.
        syncWithDCRAtStartup(MaxDcrTransactionId);
    }
}
```

Using the DCR Command-Line Interface

The DCR command-line intepreter allows you to conduct via the command line most of the important tasks normally accessible only via the DCR API or the GUI. As with the GUI, all CLI commands must be executed on the same machine on which the target DCR server is running.

Table 14-17 shows the commands and their usage.

To start the DCR command line interpreter:

1. Change to the *NMSROOT*/bin folder
2. Execute `dcrcli`.
3. At the prompt, enter a valid Cisco Works user ID and password. When dcrcli finishes authenticating this user name, it displays the dcrcli> prompt.

Table 14-17 DCR CLI Commands

Command	Syntax & Description
add	**add ip**=*value* **hn**=*value* **di**=*value* **dn**=*value* **-a** *attname*=*value* Adds the specified device to the DCR server's device list. You must specify an IP address (ip), host name (hn) or device Identity(di). You must also specify the the Display Name (dn) and the Attribute name (-a attname). The attribute sysObjectID is mandatory. You can specify as many comma-separated attribute name=value pairs as needed. For example: **add ip=1.1.1.1 hn=device1 dn=cisco.com -a sysObjectID=1.3.6.1.4.1.9.1.6**
mod	**mod id**=*value* **ip**=*value* **hn**=*value* **di**=*value* **dn**=*value* **-a** *attname*=*value* Modifies the specified device. You must enter the following: 1. Enter the Device ID (id). 2. Enter either the IP Address (ip), Hostname (hn), or Device Identity (di). 3. Enter the Display Name (dn) and the Attribute name (-a attname). You can add multiple attributes. For example: **mod id=54341 ip=2.2.2.2 dn=cisco.com -a display_name=new_name**

Table 14-17 DCR CLI Commands (continued)

Command	Syntax & Description

del	`del id=`*`value`*
	Deletes the specified device. You must specify the Device ID (id).
	For example:
	`del id=256666989`

impAcs	**`mpACS ot=`***OS Type* **`hn=`***ACS Server Name or IP address* **`un=`***ACS admin user name* **`pwd=`***ACS admin password* **`prt=`***port number* ***`cr`*****`=`***conflict resolution option* Import sdevice information directly from a remote ACS system. You must specify Operating System Type [ot], ACS Server Name or IP address [hn], ACS admin user name [un], ACS admin password [pwd] and port number [prt]. The default port number is 2002. Conflict resolution option in dcrcli that helps to override dcr data from import source file,rnms,nms,acs). dcrcli will have an extra cr (conflict resolution) option added to the end of all the import commands. This option will take two values viz. dcr for keeping the dcr data and {file, nms, rnms, acs} for keeping the data from import source. If we are not specifying cr option explicitly, dcr will be taken as the default value for cr option for all the import commands. Example: **impAcs ot=WIN2K hn=1.2.3.4 un=acsadmin pwd=acspwd prt=2002**
impFile	**`impFile fn=`***file name* **`ft=`***file type* ***`cr`*****`=`***conflict resolution option* Imports device information from a file. You must specify a filename(fn) with complete path, and the file type (ft). CSV and XML are the valid values for file type. Conflict resolution option in dcrcli that helps to override dcr data from import source file,rnms,nms,acs). dcrcli will have an extra cr (conflict resolution) option added to the end of all the import commands. This option will take two values viz. dcr for keeping the dcr data and {file, nms, rnms, acs} for keeping the data from import source. If we are not specifying cr option explicitly, dcr will be taken as the default value for cr option for all the import commands. Example: `impFile fn=d:/mypath/myImportFile.xml ft=xml cr={dcr|file}`
impNms	**`impNms nt=`***NMS type* **`il=`***Installation location* ***`cr`*****`=`***conflict resolution option* Imports device information directly from a local NMS. You must specify the NMS type [nt]. Valid values for NMS Type are are HPOV6.x and Netview7.x. Conflict resolution option in dcrcli that helps to override dcr data from import source file, rnms, nms, acs). dcrcli will have an extra cr (conflict resolution) option added to the end of all the import commands. This option will take two values viz. dcr for keeping the dcr data and {file, nms, rnms, acs} for keeping the data from import source. If we are not specifying cr option explicitly, dcr will be taken as the default value for cr option for all the import commands. Example: **impNms nt=HPOV6.x il=/opt/OV**

Table 14-17 DCR CLI Commands (continued)

Command	Syntax & Description
impRNms	**`impRNms`** **`nt=`***NMS type* **`hn=`***hostname* **`un=`***Remote User Name* **`il=`***Installation location* **`ot=`***OS Type* **`cr=`***conflict resolution option* Imports device information directly from a remote NMS. You must specify the NMS type [nt], Remote Host Name or IP address [hn], Remote User Name [un], Installation location of the NMS [il], and OS Type[ot]. Valid values of NMS Type are HPOV6.x and Netview7.x. You can specify your OS types as HPUX, AIX, or SOL. Conflict resolution option in dcrcli that helps to override dcr data from import source file, rnms, nms, acs). dcrcli will have an extra cr (conflict resolution) option added to the end of all the import commands. This option will take two values viz. dcr for keeping the dcr data and {file, nms, rnms, acs} for keeping the data from import source. If we are not specifying cr option explicitly, dcr will be taken as the default value for cr option for all the import commands. Example: **impRNms nt=HPOV6.x hn=1.2.3.4 un=root il=/opt/OV ot=SOL**

exp	`exp fn=`*filename* `ft={`*`csv`*`|`*`xml`*`}`
	
	Exports the current DCR device list to a file in CSV or XML format. You must specify a filename with complete path, and whether the file type is in CSV or XML format. For example:
	
	`exp fn=d:/mypath/myExportFile.xml ft=xml`

lsids	`lsids {all|dn=displayName|ip=IPAddress}`
	
	Lists the DCR device ID for devices stored on the DCR Server. It will list all devices if you specify no additional parameters [KR: or must you specify `lsids all` in order to list all?], or only the device ID for the device with the specified display name or IP address. If several devices share the same IP address, the command will list the device IDs for all of them. For example:
	
	`lsids ip=168.192.1.20`

detail	`detail id=`*`deviceID`*
	
	Lists all the details about the device with the ID you have specified.
	
	For example:
	
	`detail id=89992921023`

lsattr	`lsattr` This lists Attribute Name, Attribute Description, and Attribute Type. Attribute Type is a constant that identifies an Attribute Name. For example, Attribute Type 1072 identifies the attribute name display_name. For example, Attribute Type 1072 identifies the attribute name display_name.
lsmode	`lsmode` Lists the DCR ID, the DCR Group ID, the current DCR mode, and the associated Master or Slaves.
setmaster	`setmaster` Sets the DCR server to Master mode. The command takes no parameters.
setstand	`setstand` Sets the DCR server to Standalone mode. The command takes no parameters.

Table 14-17 DCR CLI Commands (continued)

Command	Syntax & Description

setslave	`setslave master=DCRGroupID port=portNumber`
	Sets the DCR server to Slave mode. You must specify the DCR Group ID for the new Master with which this slave will communicate. You must also specify the Master's port number if it is anything except 443 (443 is the default Master port). For example:
	`setslave master=DCRMaster221 port=1099`

exit	Exits the DCR command line interpreter shell.

Enhancing DCR Performance

A number of DCR APIs can cause performance problems under certain circumstances. To help you avoid these problems, the CWCS team makes the following recommendations:

1. **Avoid the `getDevices(ApiExtraInfo extranifo)` method.**

This paricular method returns the complete details of all attributes and credentials of all devices in DCR. This API should be used only very rarely, if at all, since there are very few cases when an application will want to retrieve that much information with a single call. If your DCR server contains a very large number of devices, use of this API can consume large amounts of memory, and increase the risk of CSTM errors, serialization/deserialization issues, and database-connection time out errors. This is compounded if several applications are using the same DCR, and all call this API at or near the same time.

Instead, use other DCR APIs that return only requested devices, such as `getDevices` and `getIdentityAttributes` (see recommendation 3, below). Most of these APIs require you to specify DCR Device IDs as arguments. Your application can retrieve the Device ID for DCR devices using the `getDeviceIdentifiers()` API call.

2. **Observe a 5,000-device limit on API calls.**

All DCR APIs that return an "Array of Device objects" or take as arguments an "Array of DeviceId objects" (such as `addDevices` and `updateDevices`) are subject to the potential performance problems and risk described in recommendation 1, above.

To avoid this, do not use these APIs for more than 5,000 devices in a single call. If your application needs to call the API for more than 5, 000 devices, segment the list of devices and call the API multiple times. Since all applications have a wrapper around the DCR APIs, this logic should be fairly easy to implement.

This is especially important because performance and scaling for multiple applications making simultaneous DCR API calls can not be easily handled. Large add or update operations can block API calls from other applications. It is the responsibility of applications making such calls to yield control so other applications can effectively use the DCR APIs.

3. **Use the `getIdentityAttributes()` API as much as possible.**

Use this API in preference to `getDevices()` wherever possible. This API returns a device object that contains only the following attributes: display_name, management_ip_address, host_name, domain_name, device_identity (if it is an AUS device), SysObjectId and mdf_type. Most of the time, application needs for device information, especially for display purposes (such as in the Device Selector), are satisfied by this subset of attribute data. In addition, this API executes much more quickly. For example: To process 1,000 devices, `getDevices` takes approximately 10 seconds, while `getIdentityAttributes` takes about 500 milliseconds.

Note that recommendation 2, above, also applies to `getIdentityAttributes` (that is, do not use it to fetch more than 5,000 devices at a time).

4. **Use DCR Server status-check APIs.**

Before making some API calls, you will want to ensure you are not generating useless traffic by checking that the DCR Server is actually running. DCR provides two APIs to check the status of the DCR Server:

- `isRunning` checks the status of the local DCR Server. Most of the time, you will want to use this call to check DCR Server status.
- `isMasterDCRRunning` checks the status of the Master DCR server in the domain.

5. **Ensure you have sufficient JVM thread stack and heap allocations.**

Large numbers of device objects require large amounts of memory. If your Java application will process large numbers of device objects on a regular basis, you should adjust the JVM thread and heap parameters accordingly. Based on tests, large groups of DCR device objects typically require the following memory spaces:

Objects	Memory
1,000	3.9MB
2,000	6.3MB
5,000	13.4MB
10,000	24.6MB
20,000	47.7MB
50,000	115MB

6. **Call the DCRProxy close() API at the end.**

The DCRProxy class provides a close() method to clean up and free resources, including any open DB connections, when you are finished using it. Make sure that your code calls the close() method on each proxy object once you are done using the proxy object.

7. **Use the proper DCRProxy constructor.**

You need to ensure that you use the proper DCRProxy object constructor: the default, or one that controls the loading of log4J categories for DCR and CSTM classes . For more information, see the "Creating the DCRProxy Object" section on page 14-29.