CIM SMASH/Server Management API Programming Guide

ESXi 6.5

EN-002100-00

vmware®

You can find the most up-to-date technical documentation on the VMware Web site at:

http://www.vmware.com/support/

The VMware Web site also provides the latest product updates.

If you have comments about this documentation, submit your feedback to:

docfeedback@vmware.com

VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

Contents

About This Book

The *CIM SMASH/Server Management API Programming Guide* provides information about developing applications using the CIM SMASH/Server Management API version 6.5.

VMware® provides many different APIs and SDKs for various applications and goals. This book provides information about developing management clients that use industry-standard data models. The System Management Architecture for Server Hardware (SMASH) is an industry standard for managing server hardware. This book describes the SMASH profiles implemented by VMware and contains suggestions for using the Common Information Model (CIM) classes to accomplish common use cases.

To view the current version of this book as well as all VMware API and SDK documentation, go to http://www.vmware.com/support/pubs/sdk_pubs.html.

Revision History

This book is revised with each release of the product or when necessary. A revised version can contain minor or major changes. Table 1 summarizes the significant changes in each version of this book.

Table 1. Revision History

Revision	Description
20161115	Update for vSphere 6.5. Removes support for CIM_SoftwareInstallationService..
20150312	Re-released for vSphere 6.0. No revisions.
20130912	Minor corrections only.
20120910	Updated to include information about adding CIM Provider VIBs. Updated to include information about adjusting resource pool allocation.
20110824	Added PCI Device use cases. Corrected Software Update use cases to match current design. Updated product version numbers. Corrected CIM profile version numbers. Removed Host Hardware RAID Controller profile support from default configuration. Revised Perl WS-Management section to bypass deprecated StubOps module. Removed section about Rebooting the Managed Server (deprecated feature). Revised sections about manufacturer, model, and serial number.

Table 1. Revision History (Continued)

Revision	Description
20100430	Added Active Directory Authentication. Added WS-Management code sample. Added Software Update use cases. Corrected Software Inventory use case. Updated version numbers for vSphere 4.1 release. Added Software Inventory use case. Corrected error in RAID controller illustration. Added information on crossing namespace boundaries. Corrected error in WS-Man Resource URI for VMware classes.
20090521	Updated product names for vSphere 4.0 release. Added use cases for SEL, and physical memory slots. Added namespace, ports, and XML schema information.
20080703	VMware ESX™ Server 3.5 Update 2 and ESX Server 3i version 3.5 Update 2 release. Replaced instance diagrams with expanded versions. Added use case for CPU core & threading model. Added use case for fan redundancy. Added use cases for Host Hardware RAID Controller profile. Added appendix about troubleshooting connections. Replaced Profile Reference appendix with a URL. Listed indications supported. Added ESX Server 3.5.
20080409	ESX Server 3i version 3.5 Update 1 release. Changed title (formerly *CIM SMASH API Programming Guide*) Updated URLs. Removed List of Tables. Added Physical Asset profile; listed properties for all profiles. Updated ElementName of Base Server registered profile. Added SMI-S RAID Controller profile. Divided chapter 2 into 2 parts, and expanded introductory material. Corrected typographical errors. Added some illustrations.
20071210	ESX Server 3i version 3.5 release.

Intended Audience

This book is intended for software developers who create applications that need to manage VMware vSphere® server hardware with interfaces based on CIM standards.

VMware Technical Publications Glossary

VMware Technical Publications provides a glossary of terms that might be unfamiliar to you. For definitions of terms as they are used in VMware technical documentation, go to http://www.vmware.com/support/pubs.

Document Feedback

VMware welcomes your suggestions for improving our documentation. Send your feedback to docfeedback@vmware.com.

Technical Support and Education Resources

The following sections describe the technical support resources available to you. To access the current versions of other VMware books, go to http://www.vmware.com/support/pubs.

Online Support

To use online support to submit technical support requests, view your product and contract information, and register your products, go to http://communities.vmware.com/community/developer.

Support Offerings

To find out how VMware support offerings can help meet your business needs, go to http://www.vmware.com/support/services.

VMware Professional Services

VMware Education Services courses offer extensive hands-on labs, case study examples, and course materials designed to be used as on-the-job reference tools. Courses are available onsite, in the classroom, and live online. For onsite pilot programs and implementation best practices, VMware Consulting Services provides offerings to help you assess, plan, build, and manage your virtual environment. To access information about education classes, certification programs, and consulting services, go to http://www.vmware.com/services.

Introduction to the CIM SMASH/Server Management API

1

VMware ESXi 6.5 includes a CIM Object Manager (CIMOM) that implements a set of server discovery and monitoring features that are compatible with the SMASH standard. With the VMware CIM SMASH/Server Management API, clients that use industry-standard protocols can do the following:

- Enumerate system resources
- Monitor system health data
- Upgrade installed software

The VMware implementation of the SMASH standard uses the open-source implementation of the Open Management with CIM (OMC) project. OMC provides tools and software infrastructure for hardware vendors and others who require a reliable implementation of the Distributed Management Task Force (DMTF) management profiles.

This chapter includes the following topics:

Platform Product Support

The VMware CIM SMASH/Server Management API is supported by ESXi 6.5. Hardware compatibility for ESXi is documented in the hardware compatibility guides, available on the VMware Web site. See http://www.vmware.com/support/pubs.

Supported Protocols and Versions

The VMware CIM SMASH/Server Management API supports the following protocols:

- CIM-XML over HTTP or HTTPS
- WS-Management over HTTP or HTTPS
- SLP

CIM Version

The CIM standard is an object model maintained by the DMTF, a consortium of leading hardware and software vendors. ESXi 6.5 is compatible with version 2.26.0 Final of the CIM schema.

SMASH Version

The SMASH standard is maintained by the Server Management Working Group (SMWG) of the DMTF. ESXi 6.5 is compatible with version 1.0.0 of the SMASH standard.

Supported Profiles

The VMware CIM SMASH/Server Management API supports a subset of the profiles defined by the SMWG. These profiles have overlapping structures and can be used in combinations to manage a server.

This VMware CIM implementation also includes a profile from the SMI specification developed by the Storage Networking Industry Association (SNIA). The implementation uses SMI-S version 1.3.

In some situations, the version of a profile supported by the CIMOM is important. The following table shows the version of each profile that is implemented by the VMware CIM SMASH/Server Management API for this release of ESXi.

Some profiles are only partially implemented by VMware. The implementation does not include all mandatory elements specified in the profile. These profiles are listed with "N/A" in the Version column. For information about which elements are implemented, see the VMware CIM SMASH/Server Management API and Profile Reference at http://pubs.vmware.com/vsphere-51/topic/com.vmware.cimsdk.smashref.doc/title_page.html.

Table 1-1. Profile Versions

Profile	Version
Base Server	1.0.0
Battery	1.0.0
CLP Admin Domain	N/A
CPU	1.0.0
Ethernet Port	N/A
Fan	1.0.1
Host Discovered Resources	N/A
Host LAN Port	N/A
Indications	N/A
IP Interface	N/A
Job Control	1.3.0
PCI Device	N/A
Physical Asset	1.0.2
Power State Management	1.0.1
Power Supply	1.0.1
Profile Registration	1.0.0
Record Log	1.0.0
Sensors	1.0.0
Software Inventory	1.0.0
Software Update	1.0.0
System Memory	1.0.0

The Job Control subprofile is specified by the SNIA, as part of the SMI-S. All other profiles are specified by the DMTF.

The Software Update profile is not supported in the base installation. It requires a separate VIB installation.

CIM and SMASH Resources Online

The following resources related to the CIM, SMASH, and SMI standards are available:

- http://www.dmtf.org (DMTF home page)
- http://www.dmtf.org/standards/cim (CIM standards)
- http://www.dmtf.org/standards/published_documents (DMTF publications)
- http://www.snia.org (SNIA home page)
- http://www.snia.org/tech_activities/standards/curr_standards/smi (SMI-S)

Installing CIM Provider VIBs

The 3.5 and 4.1 versions of vSphere included the LSI and HP RAID CIM providers in the default VIB for the ESXi server.

In vSphere 5.0 and later, the LSI and HP provider VIBs are not included in the default VIBs. Therefore, if you are using an LSI or HP RAID controller card on your host with vSphere 5.0 or later, you will need to install an LSI or HP VIB before the associated RAID storage device will show up in your vCenter Server Inventory.

Downloading CIM Provider VIBs

The following procedure gives you the general steps for downloading a VIB from a third-party website. The instructions may be slightly different for each third-party site.

To download a CIM Provider VIB

1 Go to the website of the CIM Provider, and look for the 'Support' or 'Downloads' section. For example, on the HP website, the section is called, 'HP Drivers and Support'. On the LSI Corporation website, the section is called, 'Support Downloads By Product' under the 'Support' tab.

2 Enter the hardware type for the VIB you want to download, or select the type from a list.

3 Choose the VIB bundle that contains the words 'VMware' and the VMware product, such as 'ESXi'.

Adding a CIM Provider VIB to your ESXi Image

You can add a CIM Provider VIB to your ESXi image using the vSphere ESXi Image Builder CLI. Install VIBs from only one OEM vendor at a time.

Before you begin, install the VMware PowerCLI software.

Use the following steps to add a new VIB to your image:

1 Run `Add-EsxSoftwareDepot` for each depot you want to work with.

Run `Add-EsxSoftwareDepot -DepotUrl` *`depot_url`*

or

Run `Add-EsxSoftwareDepot -DepotUrl` *`C:\file_path\offline-bundle.zip`*

The cmdlet returns one or more SoftwareDepot objects.

2 Run `Get-EsxImageProfile` to list all image profiles in all currently visible depots.

```
Get-EsxImageProfile
```

The cmdlet returns all available profiles. You can narrow your search by using the optional arguments to filter the output.

3 Clone the profile and make changes to the clone if the image profile is read only.

```
New-EsxImageProfile -CloneProfile My_Profile -Name "Test Profile Name"
```

Image profiles published by VMware and its partners are read only.

4 Run `Add-EsxSoftwarePackage` to add a new package to one of the image profile.

```
Add-EsxSoftwarePackage -ImageProfile My_Profile -SoftwarePackage partner-package
```

The cmdlet runs the standard validation tests on the image profile. If validation succeeds, the cmdlet returns a modified, validated image profile. If the VIB that you want to add depends on a different VIB, the cmdlet displays that information and includes the VIB that would resolve the dependency. If the acceptance level of the VIB that you want to add is lower than the image profile acceptance level, an error results. Change the acceptance level of the image profile to add the VIB.

Your image profile now includes the new VIB.

See http://pubs.vmware.com/vsphere-60/topic/com.vmware.vsphere.install.doc/GUID-B81FE465-A43B-462D-BBBF-85B27F5BBAE8.html for more information about how to add a VIB to an image profile.

Adjusting the Resource Pool Allocation

When you install several CIM provider VIBs on an ESXi system, you might find that the providers as a whole exceed the default capacity of the memory resource pool allocated for plug-ins. Therefore, if you experience memory contention after adding more than one CIM plug-in, you may need to adjust the memory pool on your ESXi server.

To adjust the resource pool using the vSphere client

1 Navigate to the Host->Configuration->System Resource Allocation->Advanced page.

2 Select a resource pool, click the right mouse button, and select 'Edit Settings'.

3 Use the slider mechanism or the up and down arrows to adjust the resource allocation for each pool.

Developing Client Applications for the CIM API

2

A basic CIM client that allows you to connect to a CIM server can be outlined as several steps that build on prior steps. Each step is explained and illustrated with pseudocode. You can expand this outline to create clients that allow you to manage the server.

The CIM client outline presented in this chapter shows a recommended general approach to accessing the CIM objects from the Interop namespace. This approach assumes no advance knowledge of the specifics of the CIM implementation. If your client is aware of items such as the Service URL and the namespaces used in the VMware implementation, see "Using the CIM Object Space" on page 25 for more information about accessing specific objects in the Implementation namespace.

This chapter includes the following topics:

CIM Server Ports

CIM servers are available for both CIM-XML and WS-Management protocols, and for both secured and non-secured HTTP connections. Select one of the ports that corresponds to the type of connection you want to make. Table 2-1 shows the default port numbers used by the CIM servers.

Table 2-1. Port Numbers for CIM Client Connections

Connection Type	Port Number	Active in the Default Configuration?
CIM-XML/HTTP	5988	No
CIM-XML/HTTPS	5989	Yes
WS-Man/HTTP	80	No
WS-Man/HTTPS	443	Yes

CIM Object Namespaces

To access a CIM object directly, you must know the namespace in which the object is stored. A managed server can have several CIM namespaces. This guide uses the Interop namespace and the Implementation namespace.

Most CIM objects are stored in the Implementation namespace. If you know the URL and the Implementation namespace in advance, you can enumerate objects directly by connecting to that namespace.

The Interop namespace contains a few CIM objects, particularly instances of `CIM_RegisteredProfile`. One of these instances exists for each CIM profile that is fully implemented on the managed server.

`CIM_RegisteredProfile` acts as a repository of information that can be used to identify and access objects in the Implementation namespace. For each registered CIM profile, the CIM server has an association that you can follow to move from the Interop namespace to the Implementation namespace.

Some profiles in the VMware implementation are only partially implemented. The implementation does not include all the mandatory properties and methods for those profiles. The Interop namespace does not contain instances of `CIM_RegisteredProfile` for profiles that are only partially implemented. To access unregistered profiles, you must know the Implementation namespace.

Crossing Between Namespaces

The ElementConformsToProfile association crosses the boundary between the Interop namespace and the Implementation namespace. The association is instantiated in both namespaces, so you can enumerate it in either namespace.

The endpoint references in any instance of the ElementConformsToProfile association include the namespace for the endpoint. If you access the referenced endpoint, such as with a `GetInstance()` method, the request is directed to the provider in the correct namespace.

For example, if you enumerate the class `OMC_ElementConformsToRecordLogProfile` in the Interop namespace, you get an object that associates an instance of `OMC_RegisteredRecordLogProfile` in the Interop namespace with an instance of `OMC_IpmiRecordLog` in the Implementation namespace. The endpoint references look similar to these:

```
ConformantStandard =
  root/interop:OMC_RegisteredRecordLogProfile.InstanceID="IPMI:vmware-host SEL Log"

ManagedElement =
  root/cimv2:OMC_IpmiRecordLog.InstanceID="IPMI:vmware-host SEL Log (Node 0)"
```

If you enumerate the class `OMC_ElementConformsToRecordLogProfile` in the Implementation namespace, you get an object in the Implementation namespace that is otherwise identical to the object in the Interop namespace.

Regardless of which namespace provides the ElementConformsToProfile instance, the endpoint references work the same. If you do a `GetInstance()` for the ConformantStandard endpoint, the CIM server returns an instance of `OMC_RegisteredRecordLogProfile` in the Interop namespace. If you do a `GetInstance()` for the ManagedElement endpoint, the CIM server returns an instance of `OMC_IpmiRecordLog` in the Implementation namespace.

To simplify the diagrams in this document, the ElementConformsToProfile association is pictured as a single object on the boundary between namespaces, rather than as two objects, one in each namespace. See "Base Server Scoping Instance Associated with Profile Registration" on page 19 for an example diagram.

Determining the Namespaces in Your Installation

You can hard-code namespaces in the client, or specify them at run time, or you can obtain the namespaces from a Service Location Protocol (SLP) Service Agent. Table 2-2 lists the namespaces used by ESXi.

Table 2-2. ESXi Namespaces

	Interop Namespace	Implementation Namespace
ESXi	`root/interop`	`root/cimv2`

You can obtain both the Interop namespace and the Implementation namespace for your managed server from SLP. You can identify the Interop namespace more conveniently than the Implementation namespace in the SLP output.

The approach preferred in this document is to use SLP to obtain the Interop namespace and the URL to enumerate `CIM_RegisteredProfile`, and then move to the Scoping Instance of the Base Server profile in the Implementation namespace. The Scoping Instance represents the managed server and is associated with many other objects in the Implementation namespace. The Scoping Instance provides a reliable point from which to navigate to CIM objects that represent any part of the managed server.

WS-Management Resource URIs

For WS-Management connections, the client must also specify a resource URI when accessing CIM objects. The URI represents an XML namespace associated with the schema definition.

The choice of URI depends on the class name of the CIM object. The prefix of the class name determines which URI the client must use. Table 2-3 shows which URI to use for each supported class name prefix.

Table 2-3. Resource URIs for CIM classes

Class Name Prefix	Resource URI (Namespace only - link will not work in a browser)
VMware_	http://schemas.vmware.com/wbem/wscim/1/cim-schema/2/
OMC_	http://schema.omc-project.org/wbem/wscim/1/cim-schema/2/
CIM_	http://schemas.dmtf.org/wbem/wscim/1/cim-schema/2/

Note that the URIs given above do not resolve to a web page location. Although they look like a web address, they just represent a section of the CIM XML schema that you need to specify.

Example:

```
xmlns="http://schemas.vmware.com/wbem/wscim/1/cim-schema/2/"
```

See http://www.w3schools.com/schema/schema_example.asp for more information about XML namespaces.

Locating a Server with SLP

If you do not know the URL to access the WBEM service of the CIMOM on the ESXi machine, or if you do not know the namespace, use SLP to discover the service and the namespace before your client makes a connection to the CIMOM.

SLP-compliant services attached to the same subnet respond to a client SLP query with a Service URL and a list of service attributes. The Service URL returned by the WBEM service begins with the service type `service:wbem:https://` and follows with the domain name and port number to connect to the CIMOM.

Among the attributes returned to the client is `InteropSchemaNamespace`. The value of this attribute is the name of the Interop namespace.

For more information about SLP, see the following links:

- http://tools.ietf.org/html/rfc2608
- http://tools.ietf.org/html/rfc3059

CIM Ticket Authentication

A CIM client must authenticate before it can access data or perform operations on an ESXi host. The client can authenticate in one of the following ways.

- Directly with the CIMOM on the managed host by supplying a valid user name and password for an account that is defined on the managed host.
- With a sessionId that the CIMOM accepts in place of the user name and password. The sessionId (called a "ticket") can be obtained by invoking the `AcquireCimServicesTicket()` method on VMware vCenter™ Server.

As a best practice, use CIM ticket authentication for servers managed by vCenter. If the managed host is operating in lockdown mode, the CIMOM does not accept new authentication requests from CIM clients. However, the CIMOM does continue to accept a valid ticket obtained from vCenter Server.

The ticket must be obtained by using the credentials of any user that has administrative privileges on vCenter Server. For more information about CIM ticket authentication, see the VMware technical note CIM Authentication for Lockdown Mode.

Active Directory Authentication

ESXi hosts implement the Pluggable Authentication Module (PAM) framework, which can be configured to support authentication of Active Directory users. This feature is transparent to the CIM client. The client uses Active Directory authentication by supplying a user name and password that were previously entered into the Active Directory database.

System administrators can use the vSphere Client or the Web Services SDK to add an ESXi host to the Active Directory domain and to grant access rights to specific users. Hosts configured to use Active Directory authentication can also be configured to accept local users that have been granted access rights.

Making a Connection to the CIMOM

Before you can enumerate classes, invoke methods, or examine properties of the managed server, you must create a connection object in your client. The connection object manages the connection with the CIM server, accepts CIM methods by proxy, and passes them to the CIM server. The following pseudocode illustrates how to create a connection by using command-line parameters passed to the client.

To make a connection to the CIMOM

1 Collect the connection parameters from the environment.

```
use os

function parse_environment()
   ///Check if all parameters are set in the shell environment.///
   VI_SERVER = VI_USERNAME = VI_PASSWORD = VI_NAMESPACE=Null
   ///Any missing environment variable is cause to revert to command-line arguments.///
   try
      return { 'VI_SERVER':os.environ['VI_SERVER'], \
               'VI_USERNAME':os.environ['VI_USERNAME'], \
               'VI_PASSWORD':os.environ['VI_PASSWORD'], \
               'VI_NAMESPACE':os.environ['VI_NAMESPACE'] }
   catch
      return Null

use sys

function get_params()
   ///Check if parameters are passed on the command line.///
   param_host = param_user = param_password = param_namespace = Null
   if len( sys.argv ) == 5
      print 'Connect using command-line parameters.'
      param_host, param_user, param_password, param_namespace = sys.argv [ 1:5 ]
      return { 'host':param_host, \
               'user':param_user, \
               'password':param_password, \
               'namespace':param_namespace }
   env = parse_environment()
   if env
      print 'Connect using environment variables.'
      return { 'host':env['VI_SERVER'], \
               'user':env['VI_USERNAME'], \
               'password':env['VI_PASSWORD'], \
               'namespace':env['VI_NAMESPACE'] }
   else
      print 'Usage: ' + sys.argv[0] + ' <host> <user> <password> [<namespace>]'
      print '  or set environment variables: VI_SERVER, VI_USERNAME, VI_NAMESPACE'
      return Null

params = get_params()
if params is Null
   exit(-1)
```

2 Create the connection object in the client.

```
use wbemlib
connection = Null

function connect_to_host( params )
   ///Connect to the server.///
   connection = wbemlib.WBEMConnection( 'https://' + params['host'], \
             ( params['user'], params['password'] ), \
             params['namespace'] )
   return connection

if connect_to_host( params )
   print 'Connected to: ' + params['host'] + ' as user: ' + params['user']
else
   print 'Failed to connect to: ' + params['host'] + ' as user: ' + params['user']
```

With some client libraries, creating a connection object in the client does not send a request to the CIMOM. A request is not sent until a method is called. To verify that such a client can connect to and authenticate with the server, see another use case, such as "Listing Registered Profiles" on page 18.

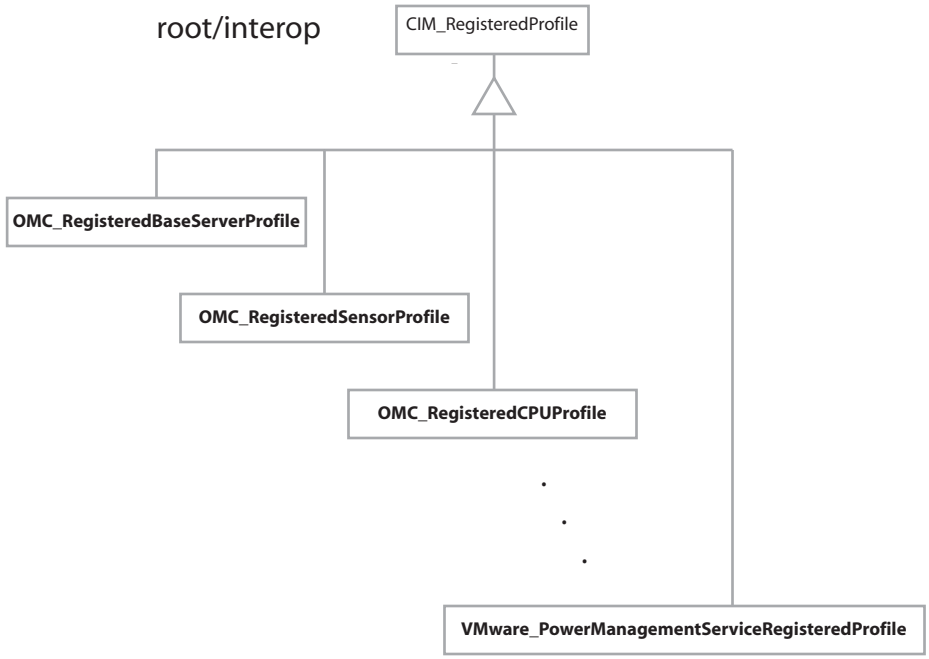
Listing Registered Profiles

VMware recommends that CIM clients list the registered profiles before you use them for other purposes. If a profile is not present in the registration list (`CIM_RegisteredProfile`), the profile is not implemented or is incompletely implemented.

SMASH profiles are registered in the Interop namespace, even when they are implemented in the Implementation namespace. A client exploring the CIM objects on the managed server can use the associations to move from `CIM_RegisteredProfile` to the objects in the Implementation namespace.

The `CIM_RegisteredProfile` class is instantiated with subclasses that represent the profiles that are registered in the Interop namespace. Each instance represents a profile that is fully implemented in the Implementation namespace. Figure 2-1 shows a few instances of `CIM_RegisteredProfile` subclasses.

Figure 2-1. Registered Profile Subclasses in Interop Namespace



The following pseudocode shows one way to identify the profiles registered on the managed server. The pseudocode in this topic depends on the pseudocode in "Making a Connection to the CIMOM" on page 16.

To list registered profiles

1 Connect to the server URL.

Specify the Interop namespace, supplied as a parameter, for the connection.

```
use wbemlib
use sys
use connection renamed cnx
connection = Null

params = cnx.get_params()
if params is Null
   sys.exit(-1)
interop_params = params
interop_params['namespace'] = 'root/interop'
connection = cnx.connect_to_host( interop_params )
if connection is Null
   print 'Failed to connect to: ' + params['host'] + ' as user: ' + params['user']
```

2 Enumerate instances of `CIM_RegisteredProfile`.

```
function get_registered_profile_names( connection )
   ///Get instances of RegisteredProfile.///
   instance_names = connection.EnumerateInstanceNames( 'CIM_RegisteredProfile' )
   if instance_names is Null
      print 'Failed to enumerate RegisteredProfile.'
      return Null
   else
      return instance_names

instance_names = get_registered_profile_names( connection )
if instance_names is Null
   sys.exit(-1)
```

3 For each instance of `CIM_RegisteredProfile`, print the name and version of the profile.

```
function print_profile( instance )
   print '\n' + ' [' + instance.classname + '] ='
   for prop in ( 'RegisteredName', 'RegisteredVersion' )
      print ' %30s = %s' % ( prop, instance[prop] )

for instance_name in instance_names
   instance = connection.GetInstance( instance_name )
   print_profile( instance )
```

Identifying the Base Server Scoping Instance

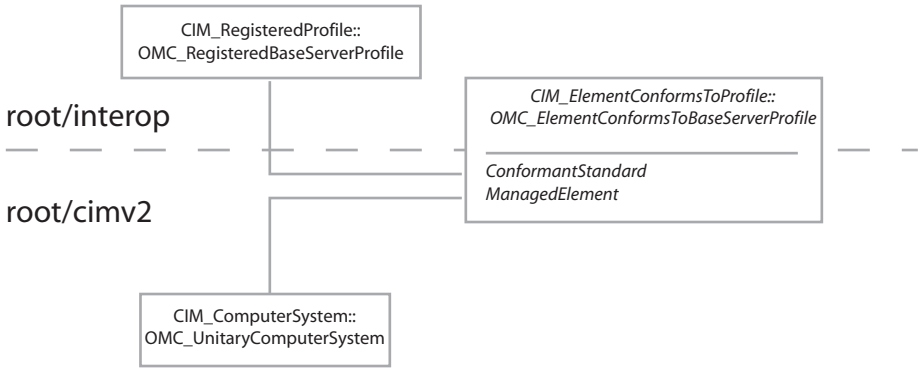
The Scoping Instance of `CIM_ComputerSystem` for the Base Server profile is an object that represents the managed server. Various hardware and software components of the managed server are represented by CIM objects associated with this Scoping Instance.

A client can locate CIM objects by using one of the following ways:

- Enumerate instances in the Implementation namespace, and then filter the results by their property values. This approach requires specific knowledge of the Implementation namespace and the subclassing used by the SMASH implementation on the managed server.
- Locate the Base Server Scoping Instance that represents the managed server, and then traverse selected association objects to find the desired components. This approach requires less knowledge of the implementation details.

Figure 2-2 shows the association between the profile registration instance in the Interop namespace and the Base Server Scoping Instance in the Implementation namespace.

Figure 2-2. Base Server Scoping Instance Associated with Profile Registration



The following pseudocode shows how to traverse the association to arrive at the Base Server Scoping Instance. This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 16.

To identify the Base Server Scoping Instance

1 Connect to the server URL.

Specify the Interop namespace, supplied as a parameter, for the connection.

```
use wbemlib
use sys
use connection renamed cnx
connection = Null

params = cnx.get_params()
if params is Null
   sys.exit(-1)
interop_params = params
interop_params['namespace'] = 'root/interop'
connection = cnx.connect_to_host( interop_params )
if connection is Null
   print 'Failed to connect to: ' + params['host'] + ' as user: ' + params['user']
   sys.exit(-1)
```

2 Enumerate instances of `CIM_RegisteredProfile`.

```
use registered_profiles renamed prof

profile_instance_names = prof.get_registered_profile_names( connection )
if profile_instance_names is Null
   print 'No registered profiles found.'
   sys.exit(-1)
```

3 Select the instance that corresponds to the Base Server profile.

```
function isolate_base_server_registration( connection, instance_names )
   ///Isolate the Base Server registration.///
   for instance_name in instance_names
      instance = connection.GetInstance( instance_name )
      if instance[ 'RegisteredName' ] == 'Base Server'
         return instance_name
   return Null

profile_instance_name = isolate_base_server_registration( connection, \
                                                          profile_instance_names )
if profile_instance_name is Null
   print 'Base Server profile is not registered in namespace ' + namespace
   sys.exit(-1)
```

4 Traverse the `CIM_ElementConformsToProfile` association to reach the Scoping Instance.

```
function associate_to_scoping_instance( connection, profile_name )
   ///Follow ElementConformsToProfile from RegisteredProfile to ComputerSystem.///
   instance_names = connection.AssociatorNames( profile_name, \
                               AssocClass = 'CIM_ElementConformsToProfile', \
                               ResultRole = 'ManagedElement' )
   if len( instance_names ) > 1
      print 'Error:  %d Scoping Instances found.' % len( instance_names )
      sys.exit(-1)
   return instance_names.pop()

function print_instance( instance )
   print '\n' + ' [' + instance.classname + '] ='
   for prop in instance.keys()
      print ' %30s = %s' % ( prop, instance[prop] )

scoping_instance_name = associate_to_scoping_instance( connection, profile_instance_name )
if scoping_instance_name is Null
   print 'Failed to find Scoping Instance.'
   sys.exit(-1)
else
   print_instance( connection.GetInstance( scoping_instance_name )
```

Mapping Integer Property Values to Strings

Many of the properties defined in CIM contain integer values that represent status or configuration information. The qualifiers for those properties define a mapping to human-readable string values.

This example shows a general-purpose routine for converting an integer value to the corresponding string value. The example assumes that the client library you are using has support for introspecting class property information available in the qualifiers.

The following function expects three parameters:

- A connection object that you have previously created, as described in "Making a Connection to the CIMOM" on page 16
- An instance of the class that you have retrieved from the CIMOM
- A string value containing the name of a property of that instance, to be mapped to its string descriptor

```
use wbemlib
use connection

function map_instance_property_to_string( connection, instance, prop )
   class_info = connection.GetClass( instance.classname, includeQualifiers=True )
   qualifiers = class_info.properties[ prop ].qualifiers
   if qualifiers.key( 'ValueMap' ) and qualifiers.key( 'Values' )
      strings = qualifiers[ 'Values' ]
      nums = qualifiers[ 'ValueMap' ]
      prop_val = instance[ prop ]
      for ( i=0; len( nums ) - 1; i++ )
         if str( nums[ i ] ) == str( prop_val )
            return strings[ i ]
   return Null
```

Using the Web Services for Management Perl Library

VMware ESXi supports the WS-Management protocol in addition to the CIM-XML protocol for passing CIM information between client and server. VMware provides WS-Management client libraries as part of the vSphere SDK for Perl.

In the VMware Web Services for Management Perl Library there are two API layers recommended for Perl clients:

- WSMan::WSBasic implements serialization and deserialization of objects transported with the SOAP protocol.
- WSMan::GenericOps implements a wrapper interface for WSMan::WSBasic. WSMan::GenericOps provides CIM objects in the form of Perl hashes.

NOTE The StubOps API layer, which provided a wrapper for WSMan::GenericOps, was deprecated in ESXi 5.0. You can use the GenericOps API layer to get the same results.

Using the WSMan::GenericOps layer of the SDK is similar to using a CIM-XML client library. The client creates a connection object, enumerates instances, and traverses associations in the same general way as described in "Making a Connection to the CIMOM" on page 16, "Listing Registered Profiles" on page 18, and "Identifying the Base Server Scoping Instance" on page 19. For more information about the vSphere SDK for Perl, see the *vSphere SDK for Perl Programming Guide*.

The following code example shows how you can make a connection to the CIM server, enumerate registered profiles, and follow the ElementConformsToProfile association to the Base Server Scoping Instance of ComputerSystem.

```
#!/usr/bin/perl
use strict;
use warnings;
use VMware::VIRuntime;
use WSMan::GenericOps;
use VMware::VILib;
$Util::script_version = "1.0";
=pod
  USAGE:: perl central_server.pl --server myserver.example.com --username abc
          --password xxxx [--namespace xxx/xxx] [--timeout numsecs]
=cut
my %opts = (
   namespace  => {
      type     => '=s',
      help     => 'Namespace for queries. Default is root/interop for profile registration.',
      required => 0,
      default => 'root/interop',
   },
   timeout  => {
      type  => '=s',
      help  => 'Default http timeout for all the queries. Default is 120',
      required => 0,
      default => '120'
   },
);
Opts::add_options( %opts );
Opts::parse();
Opts::validate();

Opts::set_option( 'protocol', 'http' );
Opts::set_option( 'servicepath', '/wsman' );
Opts::set_option( 'portnumber', '80');

sub create_connection_object
{
   my %args = (
      path => Opts::get_option( 'servicepath' ),
      username => Opts::get_option( 'username' ),
      password => Opts::get_option( 'password' ),
      port => Opts::get_option ( 'portnumber' ),
      address => Opts::get_option ( 'server' ),
      namespace => Opts::get_option( 'namespace' ),
      timeout  => Opts::get_option( 'timeout' )
   );
   my $client = WSMan::GenericOps->new( %args );
   if ( not defined $client ) {
      print "Failed to create connection object.\n";
      return undef;
   }
   # Add resource URIs for derived classes:
   $client->register_class_ns( OMC => 'http://schema.omc-project.org/wbem/wscim/1/cim-schema/2',
                               VMware => 'http://schemas.vmware.com/wbem/wscim/1/cim-schema/2',
                             );
   return $client;
}
```

```
sub get_registered_profiles
{
   my ($client) = @_;
   my @instances = ();
   eval {
      @instances = $client->EnumerateInstances(
                               class_name => 'CIM_RegisteredProfile' );
   };
   if ( $@ ) {
      print "Failed EnumerateInstances() on CIM_RegisteredProfile.\n";
      die $@;
   }
   return @instances;
}

sub isolate_base_server_registration
{
   my ($client, @instances) = @_;
   foreach my $instance (@instances) {
      my $class_name = ( keys %$instance )[ 0 ];
      my $profile = $instance->{ $class_name };
      if ( $profile->{'RegisteredName'}
        && $profile->{'RegisteredName'} eq 'Base Server' ) {
         return $instance;
      }
   }
   return undef;
}

sub associate_to_scoping_instance
{
   my ($client, $instance) = @_;
   my $class_name = ( keys %$instance )[ 0 ];
   my $profile = $instance->{ $class_name };
   my @instances = ();
   eval {
      @instances = $client->EnumerateAssociatedInstances(
                                class_name => $class_name,
                                selectors => $profile,
                                associationclassname => 'CIM_ElementConformsToProfile',
                                resultrole => 'ManagedElement' );
   };
   if ( $@ ) {
      print "Failed EnumerateAssociatedInstances() for Base Server profile registration.\n";
      die $@;
   }
   if ( scalar( @instances ) > 1 ) {
      print "Error: " . scalar( @instances ) . " Scoping Instances found.\n";
      return undef;
   }
   pop @instances;
}

# Create client connection object for ESX host:
my $client = create_connection_object();
if ( not defined $client ) {
   die "Aborting.\n";
}
my @profile_instances = get_registered_profiles( $client );
if ( scalar( @profile_instances ) == 0 ) {
   die( 'No registered profile instances found on '
        . Opts::get_option( 'server' ) . ':'
        . Opts::get_option( 'namespace' ) . "\n"
      );
}
```

```
my $profile_instance = isolate_base_server_registration( $client, @profile_instances );
if ( not defined $profile_instance ) {
   die( "Base Server profile is not registered in namespace.\n" );
}
my $scoping_instance = associate_to_scoping_instance( $client, $profile_instance );
if ( not defined $scoping_instance ) {
   die( "No managed element found for base server.\n" );
}
print "Base Server profile Scoping Instance properties:\n";
my $class_name = ( keys %$scoping_instance )[ 0 ];
my $base_server = $scoping_instance->{ $class_name };
for my $property (keys %$base_server ) {
   my $value = 'undefined';
   if ( defined $base_server->{$property} ) {
      $value = $base_server->{$property}
   }
   print '  ', $property, ': ', $value, "\n";

}
```

Using the CIM Object Space

3

You can learn how to use the CIM object space to get information and manage a server that runs VMware ESXi by studying these examples. Each example describes a goal to accomplish, steps to accomplish the goal, and a few lines of pseudocode to demonstrate the steps used in the client. These examples are chosen primarily to explain features of the VMware implementation of the profiles, and secondarily to demonstrate common operations.

This chapter includes the following topics:

Many of the examples build on the basic steps described in "Developing Client Applications for the CIM API" on page 13.

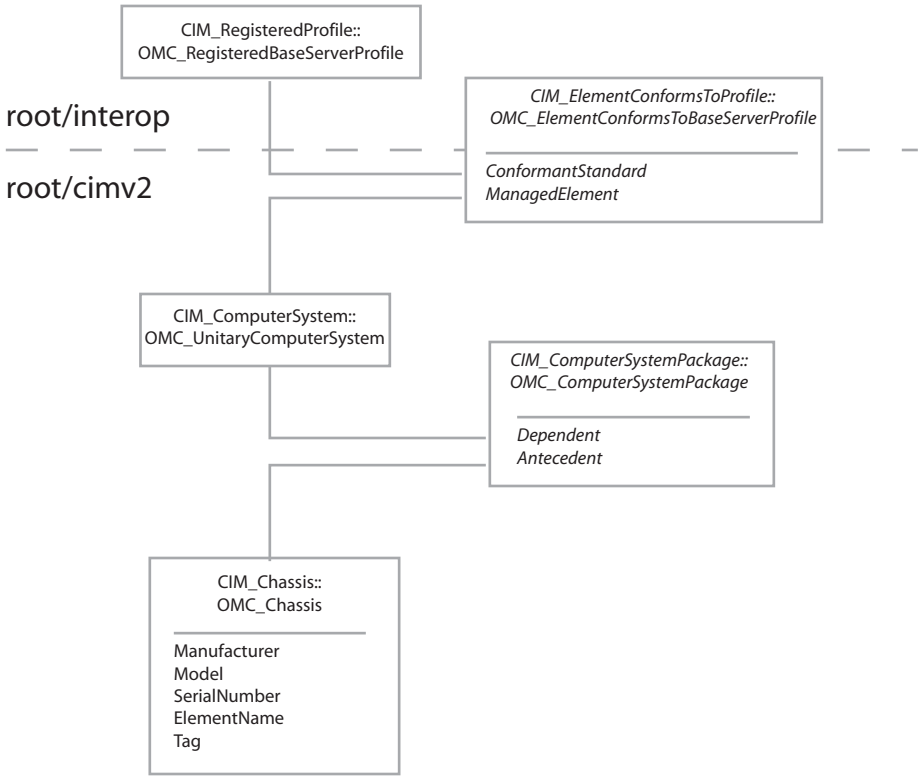
Reporting Manufacturer, Model, and Serial Number

Taking an inventory of systems in your datacenter can be a first step to monitoring the status of the servers. You can store the inventory data for future use when you monitor configuration changes.

This example shows how to get physical identifying information from the Interop namespace by traversing associations to the `CIM_Chassis` for the Scoping Instance. Figure 3-1 shows the relationships of the CIM objects involved.

If you know the Implementation namespace in advance, you can bypass the Interop namespace. For information about getting physical identifying information by using only the Implementation namespace, see "Reporting Manufacturer, Model, and Serial Number By Using Only the Implementation Namespace" on page 28.

Figure 3-1. Locating Chassis Information from the Base Server Scoping Instance



This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 16 and "Identifying the Base Server Scoping Instance" on page 19.

To report manufacturer, model, and serial number

1 Connect to the server URL.

Specify the Interop namespace, supplied as a parameter, for the connection.

```
use wbemlib
use sys
use connection renamed cnx
connection = Null

params = cnx.get_params()
if params is Null
   sys.exit(-1)
interop_params = params
interop_params['namespace'] = 'root/interop'
connection = cnx.connect_to_host( interop_params )
if connection is Null
   print 'Failed to connect to: ' + params['host'] + ' as user: ' + params['user']
   sys.exit(-1)
```

2 Locate the Base Server Scoping Instance of `CIM_ComputerSystem`.

```
use scoping_instance renamed si

scoping_instance_name = si.get_scoping_instance_name( connection )
if scoping_instance_name is Null
   print 'Failed to find Scoping Instance.'
   sys.exit(-1)
```

3 Traverse the `CIM_ComputerSystemPackage` association to reach the `CIM_Chassis` instance that corresponds to the managed server.

```
instance_names = connection.AssociatorNames( scoping_instance_name, \
                           AssocClass = 'CIM_ComputerSystemPackage', \
                           ResultClass = 'CIM_Chassis' )
if len( instance_names ) > 1
   print 'Error:  %d Chassis instances found for Scoping Instance.'  \
         % len ( instance_names )
   sys.exit(-1)
```

4 Print the `Manufacturer`, `Model`, and `SerialNumber` properties.

This example prints additional properties to help identify physical components.

```
instance_name = instance_names.pop()
instance = connection.GetInstance( instance_name )
print '\n' + 'CIM_Chassis [' + instance.classname + '] ='
for property_name in [ 'ElementName', 'Tag', \
      'Manufacturer', 'Model', 'SerialNumber' ]
   if instance.key( property_name )
      value = instance[ property_name ]
   else
      value = '(not available)'
   print '  %30s : %s' % ( property_name, value )
```

A sample of the output looks like the following:

```
CIM_Chassis [OMC_Chassis] =
                     ElementName : Chassis
                             Tag : 23.0
                    Manufacturer : Cirrostratus Systems
                           Model : 20KF6KM
                    SerialNumber : 67940851
```

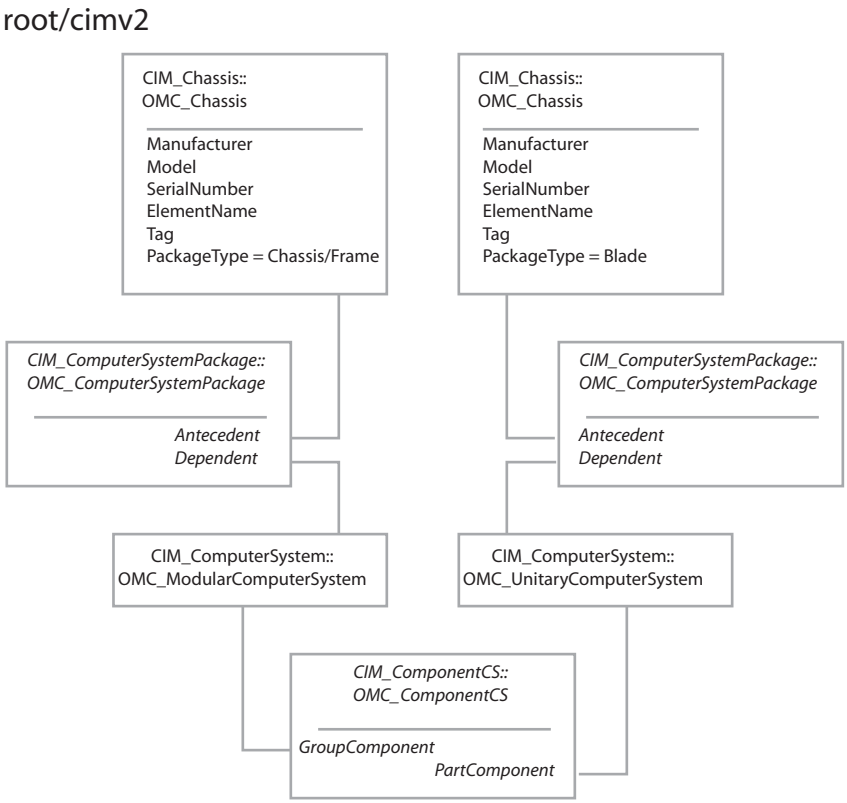
Reporting Manufacturer, Model, and Serial Number By Using Only the Implementation Namespace

Taking an inventory of systems in your datacenter can be a first step to monitoring the status of the servers. You can store the inventory data for future use in monitoring configuration changes.

This example shows how to get the physical identifying information from the Implementation namespace by enumerating `CIM_Chassis` for the managed server. This approach is convenient when the namespace is known in advance. For information about getting physical identifying information by using the Interop namespace, see "Reporting Manufacturer, Model, and Serial Number" on page 26.

You might see more than one instance of `CIM_Chassis` if the managed server is a blade system. Figure 3-2 shows an example of a server with two instances of `CIM_Chassis`, one for a blade and the other for the blade enclosure.

Figure 3-2. Locating Chassis Information in a Blade Server



This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 16, "Identifying the Base Server Scoping Instance" on page 19, and "Mapping Integer Property Values to Strings" on page 21.

To report Manufacturer, Model, and Serial Number by using only the Implementation namespace

1 Connect to the server URL.

Specify the Implementation namespace, supplied as a parameter, for the connection.

The actual namespace you will use depends on your installation.

```
use wbemlib
use sys
use connection renamed cnx
connection = Null

params = cnx.get_params()
if params is Null
```

```
   sys.exit(-1)
connection = cnx.connect_to_host( params )
if connection is Null
   print 'Failed to connect to: ' + params['host'] + ' as user: ' + params['user']
   sys.exit(-1)
```

2 Use the `EnumerateInstances` method to get all the `CIM_Chassis` instances on the server.

```
chassis_instance_names = connection.EnumerateInstanceNames( 'CIM_Chassis' )
if len( chassis_instance_names ) is 0
   print 'No %s instances were found.' % ('CIM_Chassis')
   sys.exit(0)
```

3 Print the `Manufacturer`, `Model`, and `SerialNumber` properties of the Chassis instances.

This example prints additional properties to help identify physical components.

```
use value_mapper renamed mapper

for instance_name in chassis_instance_names
   print_chassis( connection, instance_name )

function print_chassis( connection, instance_name )
   instance = connection.GetInstance( instance_name )
   print '\n' + 'CIM_Chassis [' + instance.classname + '] ='
   for property_name in [ 'ElementName', 'Tag', 'Manufacturer', \
     'Model', 'SerialNumber' ]
      if instance.key( property_name )
         value = instance[ property_name ]
      else
         value = '(not available)'
      print '  %30s : %s' % ( property_name, value )
   for property_name in [ 'PackageType', 'ChassisPackageType' ]
      if instance.key( property_name )
         value = mapper.map_instance_property_to_string( connection,
                                                         instance,
                                                         property_name )
         if value is Null
            value = ''
      else
         value = '(not available)'
      print '  %30s : %s' % ( property_name, value )
```

A sample of the output looks like the following:

```
CIM_Chassis [OMC_Chassis] =
                     ElementName : Chassis
                             Tag : 23.0
                    Manufacturer : Cirrostratus Systems
                           Model : 20KF6KM-02
                    SerialNumber : 67940851
                     PackageType : Blade
              ChassisPackageType : None

CIM_Chassis [OMC_Chassis] =
                     ElementName : Chassis
                             Tag : 23.1
                    Manufacturer : Cirrostratus Systems
                           Model : 20KF6KM-W
                    SerialNumber : 439-41902
                     PackageType : Chassis/Frame
              ChassisPackageType : Blade Enclosure
```

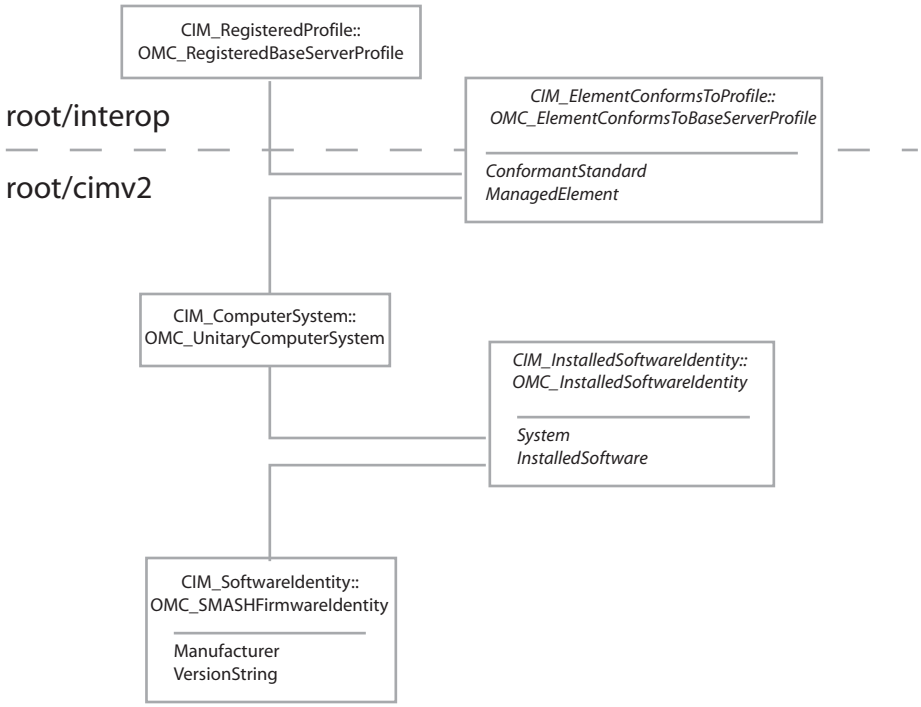
Reporting the BIOS Version

System administrators can query the BIOS version of the managed server as part of routine maintenance.

This example shows how to get the BIOS version string by traversing the `CIM_InstalledSoftwareIdentity` association from the server Scoping Instance. The VMware implementation of the Software Inventory profile uses `CIM_InstalledSoftwareIdentity` to associate firmware and hypervisor instances of `CIM_SoftwareIdentity` to the server Scoping Instance. VMware does not implement the `CIM_ElementSoftwareIdentity` association for firmware and hypervisor instances, so you must use `CIM_InstalledSoftwareIdentity` to locate the system BIOS instance of `CIM_SoftwareIdentity`.

Figure 3-3 shows the relationships of the CIM objects involved.

Figure 3-3. Locating the BIOS Version from the Base Server Scoping Instance



The VMware implementation of `CIM_SoftwareIdentity` makes the version available in the `VersionString` property rather than in the `MajorVersion` and `MinorVersion` properties.

This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 16 and "Identifying the Base Server Scoping Instance" on page 19.

To report the BIOS version

1 Connect to the server URL.

Specify the Interop namespace, supplied as a parameter, for the connection.

```
use wbemlib
use sys
use connection renamed cnx
connection = Null

params = cnx.get_params()
if params is Null
   sys.exit(-1)
interop_params = params
interop_params['namespace'] = 'root/interop'
connection = cnx.connect_to_host( interop_params )
if connection is Null
   print 'Failed to connect to: ' + params['host'] + ' as user: ' + params['user']
   sys.exit(-1)
```

2 Locate the Base Server Scoping Instance that represents the managed server.

```
use scoping_instance renamed si

scoping_instance_name = si.get_scoping_instance_name( connection )
if scoping_instance_name is Null
   print 'Failed to find server Scoping Instance.'
   sys.exit(-1)
```

3 Traverse the `CIM_InstalledSoftwareIdentity` association to reach the `CIM_SoftwareIdentity` instances that correspond to the software on the managed server.

```
instance_names = connection.Associators( scoping_instance_name, \
                           AssocClass = 'CIM_InstalledSoftwareIdentity', \
                           ResultRole = 'InstalledSoftware' )
```

4 Select the `CIM_SoftwareIdentity` instance that represents the BIOS of the managed server, and print the `Manufacturer` and `VersionString` properties.

```
function print_info( connection, instance_name )
   instance = connection.GetInstance( instance_name )
   print '\n' + 'CIM_SoftwareIdentity' + ' [' + instance.classname + '] ->'
   for prop in [ 'Manufacturer', 'VersionString' ]
      print ' %30s = %s' % ( prop, instance[prop] )

for instance_name in instance_names
   instance = connection.GetInstance( instance_name )
   if instance['Name'] == 'System BIOS'
      print_info( connection, instance_name )
```

Reporting Installed VIBs

System administrators can use a CIM client application to query the name and version information for the vSphere Installation Bundles (VIBs) that are installed on the managed server. This information is valuable for diagnosing software problems.

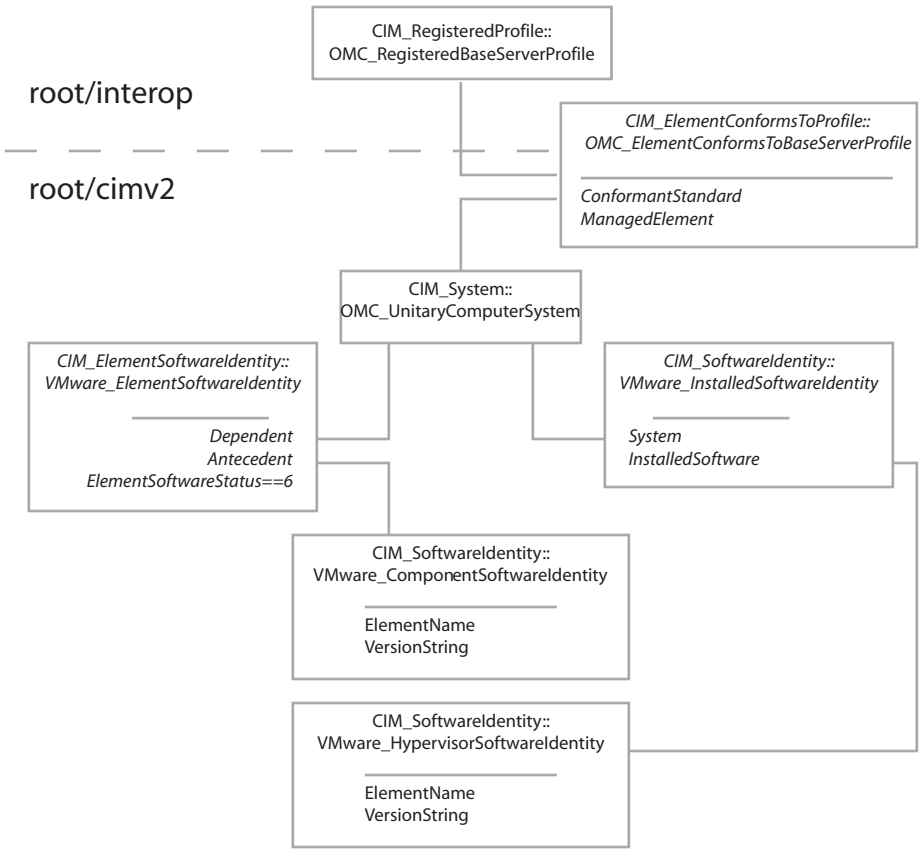
NOTE The Software Update profile is not supported in the base installation. It requires a separate VIB installation.

This example shows how to get the name and software version string by traversing the `CIM_ElementSoftwareIdentity` association from the server Scoping Instance. The VMware implementation of the Software Inventory profile uses `CIM_InstalledSoftwareIdentity` to associate only firmware and hypervisor instances of `CIM_SoftwareIdentity` to the server Scoping Instance. For VIBs, VMware implements the `CIM_ElementSoftwareIdentity` association. The `ElementSoftwareStatus` property of the `CIM_ElementSoftwareIdentity` association contains the value 6 (Installed).

Figure 3-4 shows the relationships of the CIM objects involved. VIBs are modeled with instances of `VMware_ComponentSoftwareIdentity`.

The `CIM_InstalledSoftwareIdentity` association that leads to the instance of `VMware_HypervisorSoftwareIdentity` is included in the illustration for comparison only.

Figure 3-4. Locating the Installed Software Versions from the Base Server Scoping Instance



The VMware implementation of `CIM_SoftwareIdentity` for VIBs makes the version available in the `VersionString` property rather than in the `MajorVersion` and `MinorVersion` properties.

This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 16 and "Identifying the Base Server Scoping Instance" on page 19.

To report the VIB versions

1 Connect to the server URL.

Specify the Interop namespace, supplied as a parameter, for the connection.

```
use wbemlib
use sys
use connection renamed cnx
use registered_profiles renamed prof
connection = Null

params = cnx.get_params()
if params is Null
   sys.exit(-1)
interop_params = params
interop_params['namespace'] = 'root/interop'
connection = cnx.connect_to_host( interop_params )
if connection is Null
   print 'Failed to connect to: ' + params['host'] + ' as user: ' + params['user']
   sys.exit(-1)
```

2 Locate the Base Server Scoping Instance that represents the managed server.

```
use scoping_instance renamed si

scoping_instance_name = si.get_scoping_instance_name( connection )
if scoping_instance_name is Null
   print 'Failed to find server Scoping Instance.'
   sys.exit(-1)
```

3 Use the `CIM_ElementSoftwareIdentity` association to identify the `CIM_SoftwareIdentity` instances that correspond to the software on the managed server.

```
element_softwares = connection.References( scoping_instance_name, \
                                           ResultClass = 'VMware_ElementSoftwareIdentity' )
if len( element_softwares ) < 1
   print 'No software was found for the server Scoping Instance.'
   sys.exit(-1)
```

4 Select only those instances for which the `ElementSoftwareStatus` property of the `CIM_ElementSoftwareIdentity` association has a value of 6 (Installed).

Print the `ElementName` and `VersionString` properties of the `CIM_SoftwareIdentity` instances.

```
function print_info( instance )
   print '  Software = %s' % ( instance[ 'ElementName' ] )
   print '     (Version %s)' % ( instance[ 'VersionString' ] )

print 'Installed software:'
count = 0
for software in element_softwares
   if software[ 'ElementSoftwareStatus' ] == [6L]
      print_instance( connection.GetInstance( software[ 'Antecedent' ] ) )
      count = count + 1
if not count
   print '   None'
```

Installing VIBs

The VMware implementation of the DMTF Software Update profile allows system administrators to update ESXi software by using CIM client applications. The CIM software installation service applies an offline bundle file to update the software on the managed server. To identify the current software version, see "Reporting Installed VIBs" on page 31.

NOTE The Software Update profile is not supported in the base installation. It requires a separate VIB installation.

This example shows how to locate the `CIM_SoftwareInstallationService` by traversing the `CIM_HostedService` association from the server Scoping Instance. The `InstallFromURI()` method starts the update process on the managed server and returns a `CIM_ConcreteJob` instance that you can use to monitor completion of the installation.

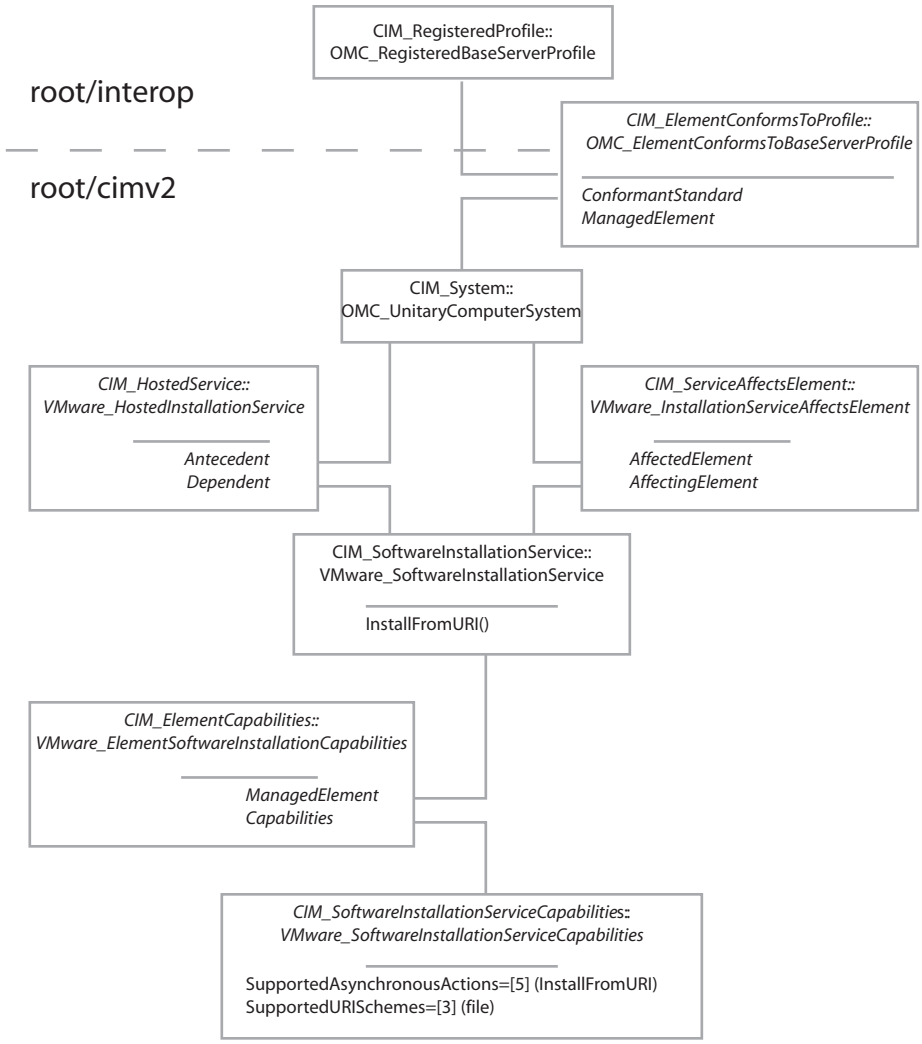
The VMware implementation of the Software Update profile does not include a `CIM_ServiceAffectsElement` association between the instance of `CIM_SoftwareInstallationService` and the instance of `CIM_SoftwareIdentity` that represents a VIB. As a result, you cannot use the `InstallFromSoftwareIdentity()` method that is described in the Software Update profile specification.

To use the `InstallFromURI()` method, you must know the location of the offline bundle in a local file system. You supply the path to the offline bundle in the form of a URI when you invoke the method. For example, you might pass "`file:///vmfs/Storage1/bundle.zip`" as the value of the URI parameter.

NOTE You cannot use an online depot in the URI that you pass to the `InstallFromURI()` method. For instructions to create an offline bundle from a set of VIBs in an online depot, see "Creating Offline Bundles" on page 71.

Figure 3-5 shows the relationships of the CIM objects involved in the installation of VIBs by using CIM. The `CIM_SoftwareInstallationService` instance in Figure 3-5 represents the CIM provider that starts the software installation.

Figure 3-5. Starting an Update of ESXi Software



The `CIM_SoftwareInstallationServiceCapabilities` instance advertises the `InstallFromURI` action and the supported URI schemes that it supports. Figure 3-5 includes the instance for completeness. The pseudocode example does not use it.

This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 16 and "Identifying the Base Server Scoping Instance" on page 19.

To install VIBs

1 Connect to the server URL.

Specify the Interop namespace, supplied as a parameter, for the connection.

```
use wbemlib
use sys
use connection renamed cnx
connection = Null

params = cnx.get_params()
if params is Null
   sys.exit(-1)
interop_params = params
```

```
interop_params['namespace'] = 'root/interop'
connection = cnx.connect_to_host( interop_params )
if connection is Null
   print 'Failed to connect to: ' + params['host'] + ' as user: ' + params['user']
   sys.exit(-1)
```

2 Locate the Base Server Scoping Instance that represents the managed server.

```
use scoping_instance renamed si

scoping_instance_name = si.get_scoping_instance_name( connection )
if scoping_instance_name is Null
   print 'Failed to find server Scoping Instance.'
   sys.exit(-1)
```

3 Use the `CIM_HostedService` association to identify the `CIM_SoftwareInstallationService` instance that represents the Software Update provider on the managed server.

The VMware implementation includes only one instance of `CIM_SoftwareInstallationService`.

```
installation_services = connection.AssociatorNames( scoping_instance_name, \
                                    AssocClass = 'CIM_HostedService', \
                                    ResultClass = 'CIM_SoftwareInstallationService' )
if len( installation_services ) != 1
   print 'Failed to find the software installation service for the scoping computer system.'
   sys.exit(-1)
installation_service = installation_services.pop()
```

4 On the `CIM_SoftwareInstallationService` instance, invoke the `InstallFromURI()` method with the following parameters.

- A URI that identifies the offline bundle file containing the VIBs that you choose to install.
- A reference to the `CIM_ComputerSystem` instance that represents the managed server.
- An empty list for the `InstallOptions` parameter. The CIM provider ignores any install options that you specify.

The method returns a single output parameter, which is a reference to an instance of `CIM_ConcreteJob`. You can use the instance to monitor completion of the software installation.

```
function launch_installation( service, \
                              bundle_file, \
                              server, \
                              cli_options )
   metadata_uri = 'file://%s' % bundle_file
   method_params = { 'URI'   : metadata_uri, \
                     'Target' : server, \
                    'InstallOptions' : cli_options }
   ( error_return, output ) = connection.InvokeMethod( 'InstallFromURI', \
                                                       service, \
                                                       **method_params )
   if error_return == 4096
      print 'Software installation in progress...'
      job_ref = output[ 'Job' ]
      return job_ref
   else
      print 'Invalid method parameters; error = %s' % error_return
      sys.exit( -1 )
vib = params['extra_params'][0]
cli_options = []
job_ref = launch_installation( installation_service, \
                           vib, \
                           scoping_instance_name, \
                           cli_options )
```

If there is an error in the method parameters, such as a mismatch in the option lists, the `InstallFromURI()` method returns immediately.

If the method returns the value 4096, the provider has accepted the method parameters and will start the update process. You can monitor the installation by using the instance of `CIM_ConcreteJob` that is returned by the method. See "Monitoring VIB Installation" on page 36.

Monitoring VIB Installation

The VMware implementation of the DMTF Software Update profile allows system administrators to use CIM client applications to update ESXi software. See "Installing VIBs" on page 33. The update can take several minutes to complete. For a CIM client, this is an asynchronous operation because the CIM server returns before the update is complete.

NOTE The Software Update profile is not supported in the base installation. It requires a separate VIB installation.

You can monitor the status of the update operation in one of two ways:

- You can poll for status of the operation by using the `CIM_ConcreteJob` class.
- You can subscribe to any of the supported indications that report changes in the status of the update operation. The supported indications are shown in Table 3-1.

Table 3-1. Indications Supported by the VMware Implementation of the Software Update Profile

Condition	CQL Expression
Any job creation	SELECT * from CIM_InstCreation WHERE SourceInstance ISA CIM_ConcreteJob
Any job change	SELECT * from CIM_InstModification WHERE SourceInstance ISA CIM_ConcreteJob
Any job deletion	SELECT * from CIM_InstDeletion WHERE SourceInstance ISA CIM_ConcreteJob

This example shows how to monitor the update and report completion status by polling an instance of `CIM_ConcreteJob`.

Figure 3-6 shows the relationships of the CIM objects involved.

Figure 3-6. Monitoring an Update of ESXi Software

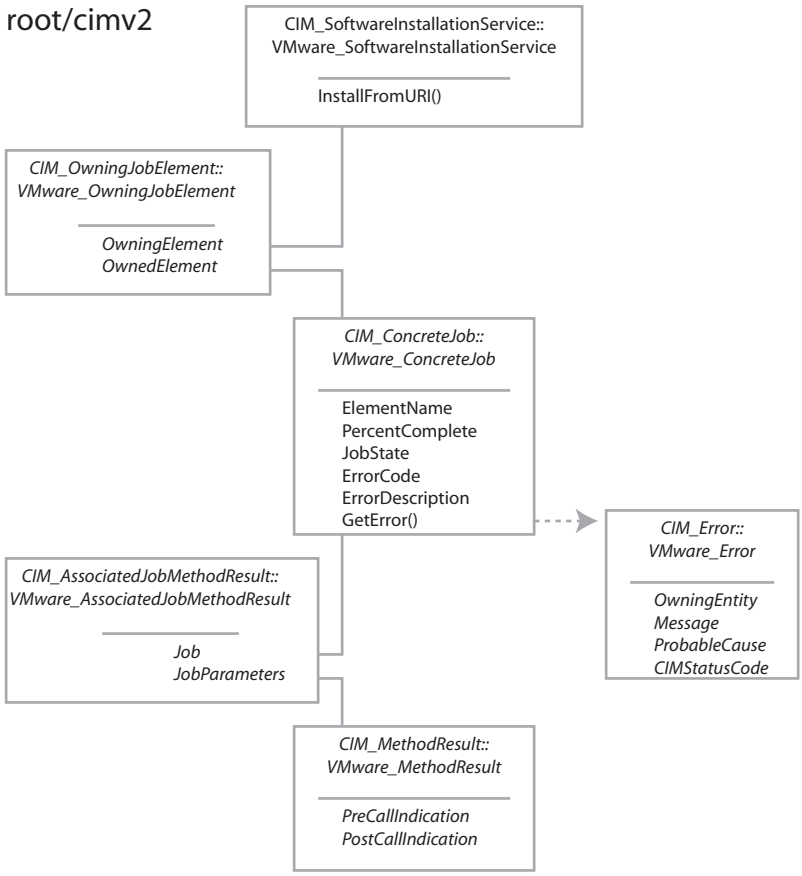


Figure 3-6 shows some classes, such as `CIM_Error`, that you can use to provide detail on status of the software update operation, but their use is not shown here. This example pseudocode relies only on the properties available in the `CIM_ConcreteJob` instance that represents the status of an operation in progress. The `CIM_ConcreteJob` instance remains in existence for a few minutes after the job completes.

This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 16 and "Identifying the Base Server Scoping Instance" on page 19.

To monitor VIB installation

1 After invoking the `InstallFromURI()` method, save the object reference returned in the `Job` output parameter.

The output parameter is a reference to an instance of `CIM_ConcreteJob` that you can use to monitor progress of the software update operation.

```
( error_return, output ) = connection.InvokeMethod( 'InstallFromURI', \
                                                    service, \
                                                    **method_params )
...
job_ref = output[ 'Job' ]
...
```

2 Retrieve the referenced instance of `CIM_ConcreteJob` and test the value of the `PercentComplete` property.

Repeat this step until the `PercentComplete` property has the value `100`.

```
function check_job_done( job_ref )
   job = connection.GetInstance( job_ref )
   print ' percent complete %3d' % job[ 'PercentComplete' ]
   print ' job status: %s' % job[ 'JobStatus' ]
   if job[ 'PercentComplete' ] == 100
```

```
      return 1
   else
      return 0

use time
ticks = 0
while not check_job_done( job_ref )
   print 'Job time elapsed: %d seconds' % ticks
   print
   time.sleep( 10 )
   ticks = ticks + 10
print '      error code: %s' % job[ 'ErrorCode' ]
print '     description: %s' % job[ 'ErrorDescription' ]
print 'Time elapsed: %d seconds' % ticks
```

While the software update operation is in progress, the property has an arbitrary value less than `100`. After the operation completes, the `PercentComplete` property takes the value `100` and the CIM server no longer updates the `CIM_ConcreteJob` instance.

A sample of the output looks like the following:

```
Software installation in progress...
   percent complete 10
   job status: Scanning URI for installable packages
Time elapsed: 0 seconds

   percent complete 10
   job status: Scanning URI for installable packages
Time elapsed: 10 seconds

   percent complete 10
   job status: Scanning URI for installable packages
Time elapsed: 20 seconds

   percent complete 30
   job status: Scan of URI Complete and installable packages found. Starting Update.
Time elapsed: 30 seconds

   percent complete 30
   job status: Scan of URI Complete and installable packages found. Starting Update.
Time elapsed: 40 seconds

...

   percent complete 100
   job status: The update completed successfully, but the system needs to be rebooted for the
               changes to be effective.
         error code: None
        description: None
Time elapsed: 1000 seconds
```

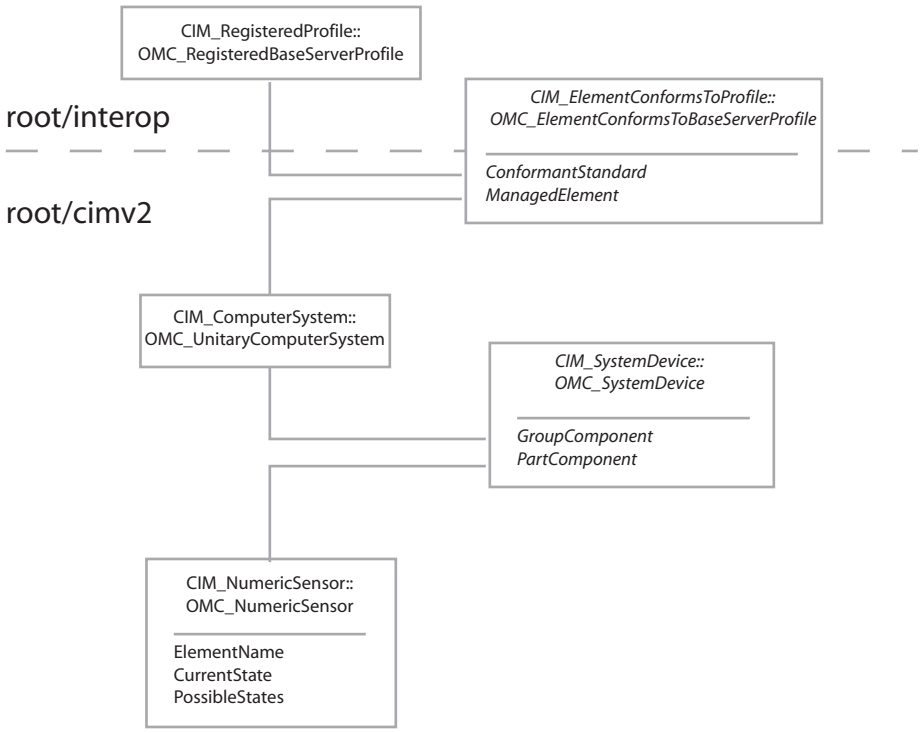
Monitoring State of All Sensors

This information is useful to system administrators who need to monitor system health. This example shows how to locate system sensors, report their current states, and flag any sensors that have abnormal states.

The example uses only `CIM_NumericSensor` instances for simplicity. You can also query discrete sensors by substituting `CIM_Sensor` for `CIM_NumericSensor`. Determining which values constitute normal sensor state is hardware-dependent.

This example shows how to get the sensor states by starting from the Interop namespace and traversing associations from the managed server Scoping Instance. Figure 3-7 shows the relationships of the CIM objects involved. For information about getting sensor states by using only the Implementation namespace, see "Monitoring State of All Sensors By Using Only the Implementation Namespace" on page 40.

Figure 3-7. Locating Sensor State from the Base Server Scoping Instance



This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 16 and "Identifying the Base Server Scoping Instance" on page 19.

To report state for all sensors

1 Connect to the server URL.

Specify the Interop namespace, supplied as a parameter, for the connection.

```
use wbemlib
use sys
use connection renamed cnx
connection = Null

params = cnx.get_params()
if params is Null
   sys.exit(-1)
interop_params = params
interop_params['namespace'] = 'root/interop'
connection = cnx.connect_to_host( interop_params )
if connection is Null
   print 'Failed to connect to: ' + params['host'] + ' as user: ' + params['user']
   sys.exit(-1)
```

2 Locate the Base Server profile Scoping Instance of `CIM_ComputerSystem`.

```
use scoping_instance renamed si

scoping_instance_name = si.get_scoping_instance_name( connection )
if scoping_instance_name is Null
   print 'Failed to find server Scoping Instance.'
   sys.exit(-1)
```

3 Traverse the `CIM_SystemDevice` association to reach the `CIM_NumericSensor` instances on the managed server.

```
instances = connection.Associators( scoping_instance_name, \
                     AssocClass = 'CIM_SystemDevice', \
                     ResultClass = 'CIM_NumericSensor' )
if len( instances ) is 0
   print 'Error:  No sensors associated with server Scoping Instance.'
   sys.exit(-1)
```

4 For each sensor instance, print the `ElementName` and `CurrentState` properties.

You can flag any abnormal values you find. Abnormal values depend on the sensor type and its `PossibleStates` property.

```
function print_info( instance, base_class )
   print '\n' + base_class + ' [' + instance.classname + '] ='
   if instance['CurrentState'] != 'Normal'
      print '********* SENSOR STATE WARNING *********\n'
   for prop in [ 'ElementName', 'CurrentState' ]
      print ' %30s = %s' % ( prop, instance[prop] )

for instance in instances
   print_info( instance, 'CIM_NumericSensor' )
```

A sample of the output looks like the following:

```
CIM_NumericSensor [OMC_NumericSensor] =
                    ElementName = FAN 1 RPM for System Board 1
                   CurrentState = Normal
CIM_NumericSensor [OMC_NumericSensor] =
                    ElementName = Ambient Temp for System Board 1
                   CurrentState = Normal
```

Monitoring State of All Sensors By Using Only the Implementation Namespace

This information is useful to system administrators who need to monitor system health. This example shows how to locate system sensors, report their current states, and flag any sensors with abnormal states.

The example uses only `CIM_NumericSensor` instances for simplicity. You can also query discrete sensors by substituting `CIM_Sensor` for `CIM_NumericSensor`. Determining which values constitute normal sensor state is hardware-dependent.

This example shows how to get the sensor states from the Implementation namespace, assuming you already know its name. For information about getting sensor state by using the standard Interop namespace, see"Monitoring State of All Sensors" on page 38.

This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 16.

To report state of all sensors by using only the Implementation namespace

1 Connect to the server URL.

Specify the Implementation namespace, supplied as a parameter, for the connection.

The actual namespace you will use depends on your installation.

```
use wbemlib
use sys
use connection renamed cnx
connection = Null

params = cnx.get_params()
if params is Null
   sys.exit(-1)
connection = cnx.connect_to_host( params )
if connection is Null
   print 'Failed to connect to: ' + params['host'] + ' as user: ' + params['user']
   sys.exit(-1)
```

2 Enumerate instances of `CIM_NumericSensor`.

```
instances = connection.EnumerateInstances( 'CIM_NumericSensor' )
if len( instances ) is 0
   print 'Error:  No sensors found on managed server.'
   sys.exit(-1)
```

3 Iterate over the sensor instances, printing the properties `ElementName` and `CurrentState`.

```
function print_info( instance )
   print '\n' + 'CIM_NumericSensor [' + instance.classname + '] ='
   if instance['CurrentState'] != 'Normal'
      print '********* SENSOR STATE WARNING *********\n'
   for prop in [ 'ElementName', 'CurrentState' ]
      print ' %30s = %s' % ( prop, instance[prop] )

for instance in instances
   print_info( instance )
```

A sample of the output looks like the following:

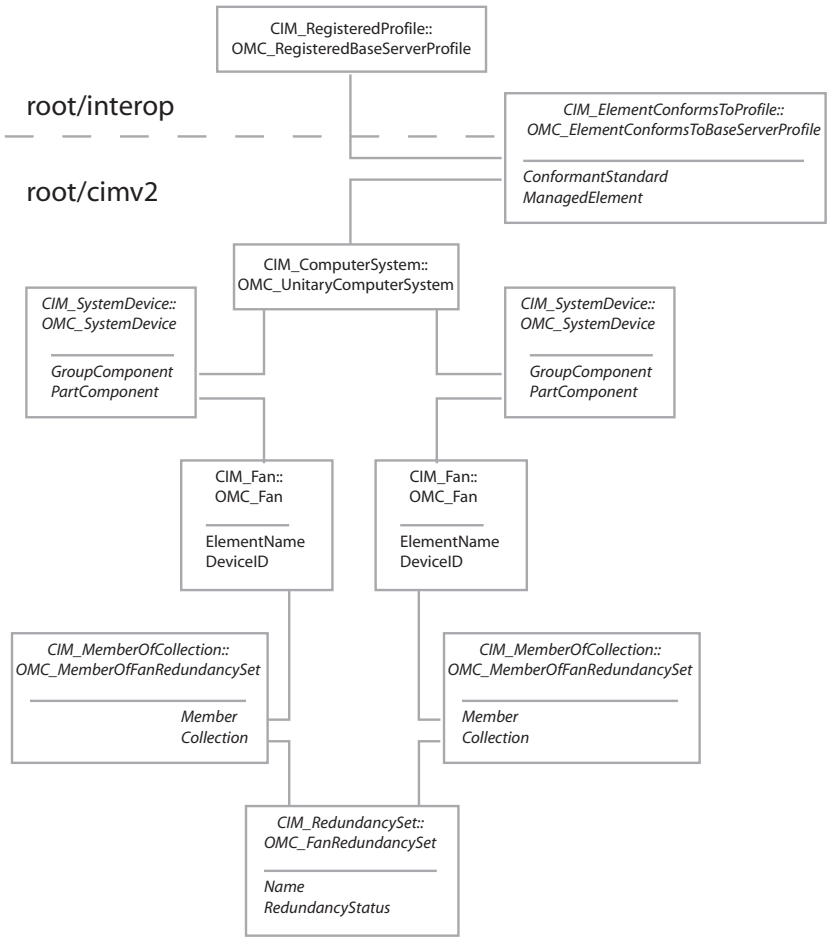
```
CIM_NumericSensor [OMC_NumericSensor] =
                    ElementName = FAN 1 RPM for System Board 1
                   CurrentState = Normal
CIM_NumericSensor [OMC_NumericSensor] =
                    ElementName = Ambient Temp for System Board 1
                   CurrentState = Normal
```

Reporting Fan Redundancy

Fan redundancy information is useful to system administrators who need to monitor system health. This example shows how to locate system fans and query the CIMOM for redundant fan relationships.

This example shows how to enumerate the fans by starting from the Interop namespace and traversing associations from the managed server Scoping Instance. Figure 3-8 shows the relationships of the CIM objects involved. If the managed server provides redundant cooling, the redundancy is modeled in the CIMOM by an instance of `CIM_RedundancySet` that is associated with two (or more) redundant fans.

Figure 3-8. Locating Redundant Fans



This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 16 and "Identifying the Base Server Scoping Instance" on page 19.

To report fan redundancy

1 Connect to the server URL.

Specify the Interop namespace, supplied as a parameter, for the connection.

```
use wbemlib
use sys
use connection renamed cnx
connection = Null

params = cnx.get_params()
if params is Null
   sys.exit(-1)
interop_params = params
interop_params['namespace'] = 'root/interop'
connection = cnx.connect_to_host( interop_params )
if connection is Null
   print 'Failed to connect to: ' + params['host'] + ' as user: ' + params['user']
   sys.exit(-1)
```

2 Locate the Base Server Scoping Instance of `CIM_ComputerSystem`.

```
use scoping_instance renamed si

scoping_instance_name = si.get_scoping_instance_name( connection )
if scoping_instance_name is Null
   print 'Failed to find server Scoping Instance.'
   sys.exit(-1)
```

3 Traverse the `CIM_SystemDevice` association to reach the `CIM_Fan` instances on the managed server.

```
fan_instances = connection.Associators( scoping_instance_name, \
                                        AssocClass = 'CIM_SystemDevice', \
                                        ResultClass = 'CIM_Fan' )
if len( fan_instances ) is 0
   print 'Error:  No fans associated with server Scoping Instance.'
   sys.exit(-1)
```

4 For each fan instance, print the `ElementName` and `DeviceID` properties.

```
function print_info( instance )
   print '\n' + 'CIM_Fan [' + instance.classname + '] ='
   for prop in [ 'ElementName', 'DeviceID' ]
      print ' %30s = %s' % ( prop, instance[prop] )

for fan_instance in fan_instances
   print_info( fan_instance )
```

5 For each fan instance, traverse the `CIM_MemberOfCollection` association to reach any instances of `CIM_RedundancySet`.

```
set_instances = connection.Associators( scoping_instance_name, \
                                        AssocClass = 'CIM_MemberOfCollection', \
                                        ResultClass = 'CIM_RedundancySet' )
```

6 For each fan instance, print the redundancy status. If the fan is not a member of a redundancy set, the redundancy status is not applicable.

```
if len( set_instances ) is 0
   print '   Redundancy status: N/A'
else
   for instance in set_instances
      name = instance['Name']
      status = instance['RedundancyStatus']
      print '   redundancy set (%s) status = %s' %
               ( instance['Name'], (status==2 ? 'Fully Redundant' : 'unknown or degraded')
```

A sample of the output looks like the following:

```
CIM_Fan [OMC_Fan] =
                    ElementName = FAN 1 RPM
                    DeviceID = 48.0.32.99
   redundancy set (117.0.32.0) status = Fully Redundant
CIM_Fan [OMC_Fan] =
                    ElementName = FAN 2 RPM
                    DeviceID = 49.0.32.99
   redundancy set (117.0.32.0) status = Fully Redundant
```

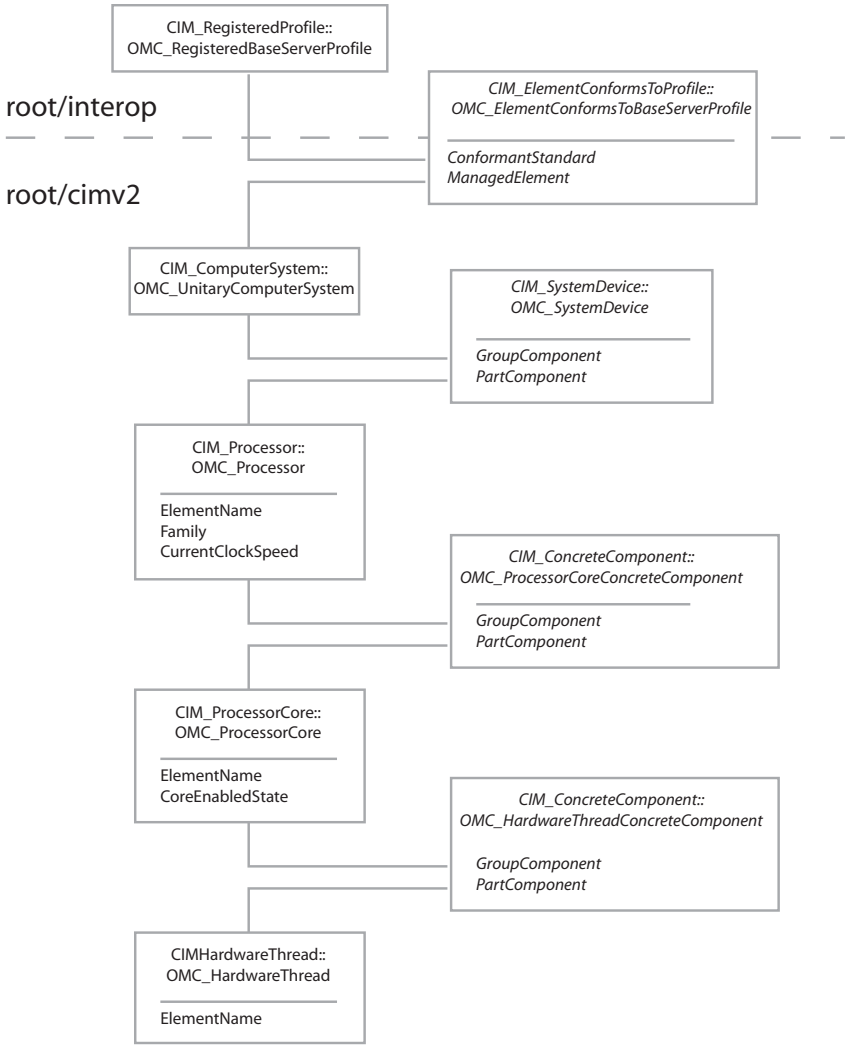
Reporting CPU Cores and Threads

This information is useful to system administrators who need to monitor system health. This example shows how to enumerate the processor cores and hardware threads in a managed server.

The VMware implementation does not include instances of `CIM_ProcessorCapabilities`, but cores and hardware threads are modeled with individual instances of `CIM_ProcessorCore` and `CIM_HardwareThread`.

This example shows how to locate information about the CPU cores and threads by starting from the Interop namespace and traversing associations from the managed server Scoping Instance. A managed server has one or more processors, each of which has one or more cores with one or more threads. Figure 3-9 shows the relationships of the CIM objects involved. For simplicity, the diagram shows only a single processor with one core and one hardware thread.

Figure 3-9. Locating CPU Cores and Hardware Threads



This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 16 and "Identifying the Base Server Scoping Instance" on page 19.

To report CPU cores and threads

1 Connect to the server URL.

Specify the Interop namespace, supplied as a parameter, for the connection.

```
use wbemlib
use sys
use connection renamed cnx
connection = Null

params = cnx.get_params()
if params is Null
   sys.exit(-1)
interop_params = params
interop_params['namespace'] = 'root/interop'
connection = cnx.connect_to_host( interop_params )
if connection is Null
   print 'Failed to connect to: ' + params['host'] + ' as user: ' + params['user']
   sys.exit(-1)
```

2 Locate the Base Server Scoping Instance of `CIM_ComputerSystem`.

```
use scoping_instance renamed si

scoping_instance_name = si.get_scoping_instance_name( connection )
if scoping_instance_name is Null
   print 'Failed to find server Scoping Instance.'
   sys.exit(-1)
```

3 Traverse the `CIM_SystemDevice` association to reach the `CIM_Processor` instances on the managed server.

```
proc_instance_names = connection.AssociatorNames( scoping_instance_name, \
                                                  AssocClass = 'CIM_SystemDevice', \
                                                  ResultClass = 'CIM_Processor' )
if len( proc_instance_names ) is 0
   print 'Error:  No processors associated with server Scoping Instance.'
   sys.exit(-1)
```

4 For each `CIM_Processor` instance, print the `ElementName`, `Family`, and `CurrentClockSpeed` properties.

```
for proc_instance_name in proc_instance_names
   instance = connection.GetInstance( proc_instance_name )
   print ‘ %s (Family: %s) (%sMHz)’ %
         ( instance[‘ElementName’], instance[‘Family’], instance[‘CurrentClockSpeed’] )
```

5 For each `CIM_Processor` instance, traverse the `CIM_ConcreteComponent` association to reach the `CIM_ProcessorCore` instances on the managed server.

```
core_instance_names = connection.AssociatorNames( proc_instance_name, \
                                                  AssocClass = 'CIM_ConcreteComponent', \
                                                  ResultClass = 'CIM_ProcessorCore' )
if len( core_instance_names ) is 0
   print 'No processor cores associated with this CPU.'
   sys.exit(-1)
```

6 For each `CIM_ProcessorCore` instance, print the `ElementName` and `CoreEnabledState` properties.

```
for core_instance_name in core_instance_names
   instance = connection.GetInstance( core_instance_name )
   print ’      %s (%s)’ % \
         ( instance[‘ElementName’], \
           (instance[‘CoreEnabledState’]==’Enabled’)?’Enabled’:’Disabled’ )
```

7 For each `CIM_ProcessorCore` instance, traverse the `CIM_ConcreteComponent` association to reach the `CIM_HardwareThread` instances on the managed server.

```
thread_instance_names = connection.AssociatorNames( core_instance_name, \
                                                    AssocClass = 'CIM_ConcreteComponent', \
                                                    ResultClass = 'CIM_HardwareThread' )
if len( thread_instance_names ) is 0
   print 'No hardware threads associated with this CPU core.'
   sys.exit(-1)
```

8 For each `CIM_HardwareThread` instance, print the `ElementName` property.

```
for thread_instance_name in thread_instance_names
   instance = connection.GetInstance( thread_instance_name )
   print '           %s' % instance['ElementName']
```

A sample of the output looks like the following:

```
CPU1 (Family: 179) (2667MHz)
     CPU1 Core 1 (Enabled)
          CPU1 Core 1 Thread 1
     CPU1 Core 2 (Enabled)
          CPU1 Core 2 Thread 1
CPU2 (Family: 179) (2667MHz)
     CPU2 Core 1 (Enabled)
          CPU1 Core 1 Thread 1
     CPU2 Core 2 (Enabled)
          CPU1 Core 2 Thread 1
```

Reporting Empty Memory Slots By Using Only the Implementation Namespace

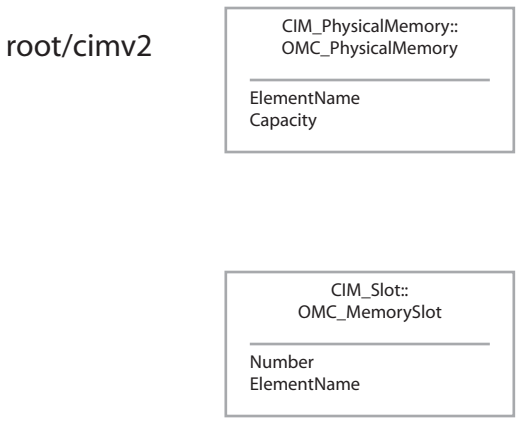
This example describes how to determine the empty slots available for new memory cards. This information is useful to system administrators who want to upgrade the capacity of a managed server.

This example shows how to locate information about the installed memory and available slots by using only the objects in the Implementation namespace. Figure 3-10 shows the CIM objects involved.

You can locate used memory slots by enumerating physical memory instances. To locate unused slots, you also enumerate the `OMC_MemorySlot` instances and compare the results. The set of unused slots comprises all those `OMC_MemorySlot` instances whose `ElementName` property does not match any of the instances of `OMC_PhysicalMemory`.

NOTE This example assumes that the managed server is a single-node system.

Figure 3-10. Locating Physical Memory Slots



This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 16.

To report empty memory slots

1 Connect to the server URL.

Specify the Implementation namespace, supplied as a parameter, for the connection.

The actual namespace you will use depends on your implementation.

```
use wbemlib
use sys
use connection renamed cnx
connection = Null

params = cnx.get_params()
if params is Null
   sys.exit(-1)
connection = cnx.connect_to_host( params )
if connection is Null
   print 'Failed to connect to: ' + params['host'] + ' as user: ' + params['user']
   sys.exit(-1)
```

2 Enumerate the `OMC_PhysicalMemory` instances.

```
chip_instances = connection.EnumerateInstances( 'OMC_PhysicalMemory' )
if len( chip_instances ) is 0
   print 'Error:  No physical memory instances were found.'
   sys.exit(-1)
```

3 Enumerate the `OMC_MemorySlot` instances.

```
slot_instances = connection.EnumerateInstances( 'OMC_MemorySlot' )
if len( slot_instances ) is 0
   print 'Error:  No memory slot instances were found.'
   sys.exit(-1)
```

4 For each `OMC_MemorySlot` instance, compare the `ElementName` property with the set of `OMC_PhysicalMemory` instances, and discard the instances that have matching `ElementName` properties.

For other instances, print the `ElementName` property.

```
function slot_filled( slot, chips )
   for chip in chips
      if slot['ElementName'] == chip['ElementName']
         return True
   return False

empty_slots = []
for slot_instance in slot_instances
   if not slot_filled( slot_instance, chip_instances )
      empty_slots.append( slot_instance )
print '  %s empty memory slots found.' % len( empty_slots )
for slot_instance in empty_slots
   print slot_instance['ElementName']
```

A sample of the output looks like the following:

```
  4 empty memory slots found.
DIMM 3C
DIMM 4D
DIMM 7C
DIMM 8D
```

Reporting the PCI Device Hierarchy By Using Parent DeviceIDs

This example describes a simple way to enumerate the PCI devices present in the managed server. This information is useful to system administrators who want to troubleshoot device problems or upgrade the hardware in a managed server.

The PCI Device profile specification allows flexibility in how the profile is implemented. Designers can apply one of three approaches to modeling PCI device connections, or they can combine these approaches for a more complete implementation. Device connections can be modeled with a combination of the following approaches.

- DeviceConnection associations
- PCIPortGroup instances that express relationships between PCI ports
- Primary and secondary bus numbers that relate PCI devices to bridges and switches

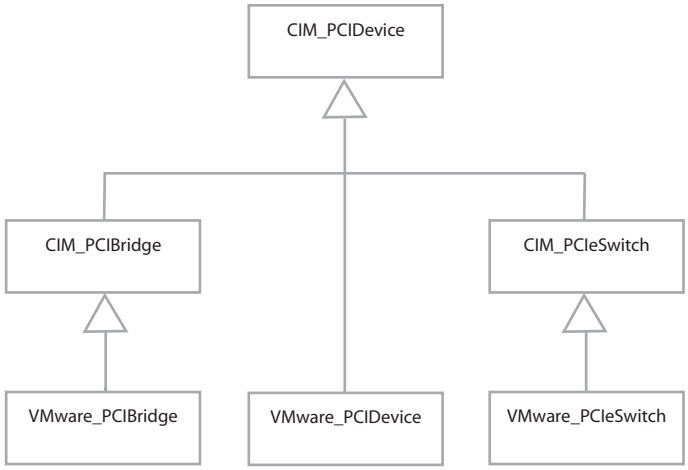
The VMware implementation supports the first two modeling approaches. For an example that uses the second approach to relating PCI devices, see "Reporting the Path to a PCI Device By Using PortGroups" on page 50.

For convenience, the VMware implementation also provides a fourth way to model device connections: `ParentDeviceID`.

The `ParentDeviceID` property relates a PCI device directly to the bridge or switch through which the device accesses the CPU. The value of the property is the value of the `DeviceID` property of that bridge or switch, which can be called its parent device. A CIM client that is aware of the `ParentDeviceID` property can map the hierarchy of PCI devices by using only that property to determine the relationships between devices.

This example shows how you can map the PCI device hierarchy by using the `ParentDeviceID` property. For illustration, this example enumerates PCI device instances by their VMware-specific class names, rather than by a parent class. Alternatively, you could enumerate the `CIM_PCIDevice` class, because all three of the VMware classes derive, directly or indirectly, from that class, as shown in Figure 3-11.

Figure 3-11. Inheritance Relationships of PCI Device Classes



This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 16.

To report the PCI device hierarchy

1 Connect to the server URL.

Specify the Implementation namespace, supplied as a parameter, for the connection.

The actual namespace you will use depends on your implementation.

```
use wbemlib
use sys
use connection renamed cnx
connection = Null

params = cnx.get_params()
if params is Null
   sys.exit(-1)
connection = cnx.connect_to_host( params )
if connection is Null
   print 'Failed to connect to: ' + params['host'] + ' as user: ' + params['user']
   sys.exit(-1)
```

2 Enumerate the `VMware_PCIDevice`, `VMware_PCIBridge`, and `VMware_PCIeSwitch` instances.

Save each instance in an associative array, keyed by its parent's `DeviceID`, or `"none"` if it has no parent. This example saves the children of each parent device as a nested associative array of instances indexed by the device's own ID.

```
dev_entries = {}
enum_devs( 'VMware_PCIDevice' )
enum_devs( 'VMware_PCIBridge' )
enum_devs( 'VMware_PCIeSwitch' )

function enum_devs( class_name )
   dev_instances = connection.EnumerateInstances( class_name )
   for dev in dev_instances
      parent = dev[ 'ParentDeviceID' ]
      if not parent
         parent = 'none'
      id = dev[ 'DeviceID' ]
      if not dev_entries.key( parent )
         dev_entries[ parent ] = {}
      dev_entries[ parent ][ id ] = dev
```

3 Starting with the value `"none"` for devices that have no parent, access the children of each parent.

For each child, print the `DeviceID`, the `BusNumber`, `DeviceNumber`, and `FunctionNumber`, and the `ElementName` properties. Recursively do the same for the children of each child device.

```
parent = 'none'
print_children( '', parent )

function print_children( indent, id )
   if dev_entries.key( id )
      dev_list = dev_entries[ id ]
      for key in dev_list.keys()
         dev = dev_list[ key ]
         print indent, print_dev( dev )
         print_children( indent + '   ', dev[ 'DeviceID' ] )

function print_dev( dev )
   dev_summary = 'ID=%s B/D/F=%s/%s/%s (%s)' % \
      (dev[ 'DeviceID ], dev[ 'BusNumber' ], dev[ 'DeviceNumber' ], \
       dev[ 'FunctionNumber' ], dev[ 'ElementName' ])
   return dev_summary
```

This pseudocode displays an indented representation of the hierarchy of PCI devices. A sample of the output looks like the following:

```
ID=PCI 0:0:1:0 B/D/F=0/1/0 (Plutonic Devices PD-631 PCI-X Bridge)
   ID=PCI 0:2:1:0 B/D/F=2/1/0 (Trans-Oort Networks E-1500 Terabit Ethernet Adapter)
   ID=PCI 0:2:1:1 B/D/F=2/1/1 (Trans-Oort Networks E-1500 Terabit Ethernet Adapter)
ID=PCI 0:0:2:0 B/D/F=0/2/0 (Plutonic Devices PD-631 PCI-X Bridge)
   ID=PCI 0:3:1:0 B/D/F=3/1/0 (Haumea HINA-15K Block Storage Adapter)
   ID=PCI 0:3:1:1 B/D/F=3/1/1 (Haumea HINA-15K Block Storage Adapter)
   ID=PCI 0:3:2:0 B/D/F=3/2/0 (Haumea HINA-15K Block Storage Adapter)
   ID=PCI 0:3:2:1 B/D/F=3/2/1 (Haumea HINA-15K Block Storage Adapter)
   ID=PCI 0:3:3:0 B/D/F=3/3/0 (Plutonic Devices PD-631 PCI-X Bridge)
      ID=PCI 0:4:1:0 B/D/F=4/1/0 (Mercuricity Generic USB OHCI Hub)
      ID=PCI 0:4:1:1 B/D/F=4/1/1 (Mercuricity Generic USB OHCI Hub)
      ID=PCI 0:4:1:2 B/D/F=4/1/2 (Mercuricity Generic USB OHCI Hub)
ID=PCI 0:0:3:0 B/D/F=0/3/0 (Albedo-Kuiper Grafix Super X-Treme Duo)
ID=PCI 0:0:3:1 B/D/F=0/3/1 (Albedo-Kuiper Grafix Super X-Treme Duo)
ID=PCI 0:0:4:0 B/D/F=0/4/0 (vAndromeda FCoW Adapter)
```

Reporting the Path to a PCI Device By Using PortGroups

This example describes a way to discover the path to a PCI device in the managed server by using the portgroup connections. This information is useful to system administrators who want to troubleshoot device problems or upgrade the hardware in a managed server.

The PCI Device profile specification allows flexibility in how the profile is implemented. Designers can apply one of three approaches to modeling PCI device connections, or they can combine these approaches for a more complete implementation. Device connections can be modeled with a combination of the following approaches.

- DeviceConnection associations
- PCIPortGroup instances that express relationships between PCI ports
- Primary and secondary bus numbers that relate PCI devices to bridges and switches

The VMware implementation supports the first two modeling approaches.

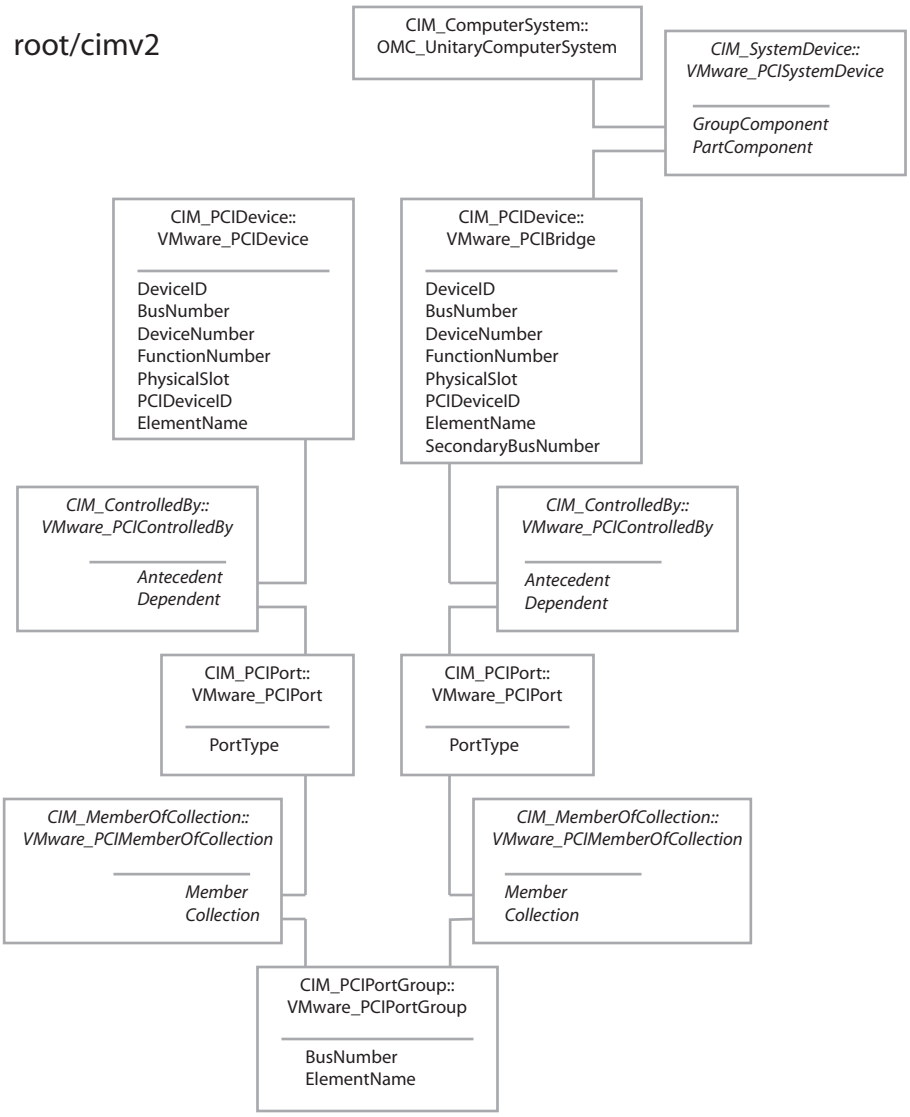
For convenience, the VMware implementation also provides a fourth way to model device connections: `ParentDeviceID`. For an example that uses the `ParentDeviceID` property, see "Reporting the PCI Device Hierarchy By Using Parent DeviceIDs" on page 47. The `ParentDeviceID` property is specific to VMware classes, so it cannot be used in vendor-independent object traversal algorithms.

This example shows how you can trace the path to a PCI device by using the `PCIPortGroup` associations. This way of relating PCI devices depends only on the properties defined in the CIM schema, so it is vendor-independent. Figure 3-12 shows the relationships of the CIM objects involved.

Given a PCI device identified by bus, device, and function numbers (`<bus>:<device>:<function>`), this example identifies and displays all ports, bridges, and switches between the chosen device and the CPU. The PCI Device profile specifies how to model associations between devices and their ports, and between ports and the logical port groups that represent all ports on the same PCI bus.

In Figure 3-12, the SystemDevice association to the managed server is included for reference, but is not used in this example.

Figure 3-12. Tracing the Path to a PCI Device By Using PortGroups



This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 16.

To trace the path to a PCI device

1 Connect to the server URL.

Specify the Implementation namespace, supplied as a parameter, for the connection.

The actual namespace you will use depends on your implementation.

```
use wbemlib
use sys
use connection renamed cnx
connection = Null

params = cnx.get_params()
if params is Null
   sys.exit(-1)
connection = cnx.connect_to_host( params )
if connection is Null
   print 'Failed to connect to: ' + params['host'] + ' as user: ' + params['user']
   sys.exit(-1)
```

2 Enumerate the names of all `CIM_PCIDevice` instances and save each instance name in an array.

```
dev_instance_names = connection.EnumerateInstances( 'CIM_PCIDevice' )
if len( dev_instance_names ) is 0
   print 'Error: No CIM_PCIDevice instances were found.'
   sys.exit(-1)
```

3 Search the array of PCI devices for one that matches the bus number, device number, and function number selected by the command-line parameters.

```
param_bus, param_device, param_function = params['extra_params'][0].split( ':' )
chosen_name = Null
for dev_name in dev_instance_names
   dev = connection.GetInstance( dev_name )
   if (dev['BusNumber'], dev['DeviceNumber'], dev['FunctionNumber']) == \
      (param_bus, param_device, param_function)
      chosen_name = dev_name
      break
if chosen_name is Null
   print 'Error: Chosen device (%s:%s:%s) not found on the managed system.' % \
         (param_bus, param_device, param_function)
   exit(-1)
```

4 Print the `DeviceID`, the `BusNumber`, `DeviceNumber`, and `FunctionNumber`, the `PhysicalSlot`, and the `ElementName` properties of the chosen device.

```
print 'Chosen device:'
print_dev( dev )

function print_dev( dev )
   print 'ID=%s B/D/F=%s/%s/%s Slot=%s Type=%s (%s)' % \
         (dev['DeviceID'], dev['BusNumber'], dev['DeviceNumber'], dev['FunctionNumber'], \
         dev['PhysicalSlot'], dev[ 'ElementName' ])
```

5 Traverse the `CIM_ControlledBy` association to get instance names of the class `CIM_PCIPort`, selecting the instance that has the same `BusNumber` as the chosen instance of `CIM_PCIDevice`.

Print the `PortType` property of the `CIM_PCIPort` instance. This example maps the `PortType` property to the corresponding string value in its `Values` qualifier.

```
port_name = connected_port_on_bus( dev_name, dev[ 'BusNumber' ]
if port_name is Null
   print 'No upstream port found.'
   break
port = connection.GetInstance( port_name )
print_port( port )

function connected_port_on_bus( dev_name, bus_number )
   port_instance_names = connection.AssociatorNames( dev_name, \
                                                     AssocClass = 'CIM_ControlledBy', \
                                                     ResultClass = 'CIM_PCIPort' )
   for port_instance_name in port_instance_names
      port = connection.GetInstance( port_instance_name, \
                                     PropertyList = ['BusNumber', 'PortType'] )
      if port[ 'BusNumber' ] == bus_number
         return port_instance_name
   return Null

use value_mapper renamed mapper
function print_port( port )
   port_type = mapper.map_property_value_to_string( port, 'PortType' )
   print ' (%s port on bus %s)' % (port_type, port[ 'BusNumber' ])
```

6 Traverse the CIM_MemberOfCollection association to the class CIM_PCIPortGroup.

A port can only belong to one portgroup, so the result is a list with one member. Print the ElementName property of the portgroup. If this portgroup has BusNumber 0, stop looping because bus 0 connects to the CPU.

```
portgroup = portgroup_of_port( port_name )
print_portgroup( portgroup )
if (portgroup[ 'BusNumber' ] == 0
   break

function portgroup_of_port( port_name )
   portgroup_instance_names = connection.AssociatorNames( \
                                        port_name, \
                                        AssocClass = 'CIM_MemberOfCollection', \
                                        ResultClass = 'CIM_PCIPortGroup' )
   portgroup_instance_name = portgroup_instance_names[ 0 ]
   return connection.GetInstance( portgroup_instance_name, \
                                  PropertyList = ['BusNumber', 'ElementName']

function print_portgroup( portgroup )
   print '  ', portgroup[ 'ElementName' ]
```

7 Enumerate instances of the CIM_PCIBridge and find one that has the same SecondaryBusNumber as the BusNumber of the instance of CIM_PCIPortGroup.

If no instance of CIM_PCIBridge is found, search for an instance of CIM_PCIeSwitch that has a SecondaryBusNumbers property containing the same BusNumber as the instance of CIM_PCIPortGroup.

```
dev_name = upstream_bridge_or_switch( portgroup[ 'BusNumber' ], 'CIM_PCIBridge' )
if dev_name is Null
   dev_name = upstream_bridge_or_switch( portgroup[ 'BusNumber' ], 'CIM_PCIeSwitch' )
   if dev_name is Null
      print 'No upstream PCI device found.'
      break

function upstream_bridge_or_switch( bus_number, class_name )
   names = connection.EnumerateInstanceNames( class_name )
   for name in names
      instance = connection.GetInstance( name )
      if class_name == 'CIM_PCIBridge' and instance[ 'SecondaryBusNumber' ] == bus_number \
      or class_name == 'CIM_PCIeSwitch' and bus_number in instance[ 'SecondaryBusNumbers' ]
         return name
   return Null
```

8 Working backwards from the bridge or switch, traverse the CIM_ControlledBy association to the class CIM_PCIPort, selecting the instance that has the same BusNumber as the portgroup.

```
port_name = connected_port_on_bus( dev_name, portgroup[ 'BusNumber' ] )
if port_name is Null
   print 'Error: Missing port on downstream side of upstream device.'
   sys.exit(-1)
```

9 Print the PortType property of the CIM_PCIPort.

```
port = connection.GetInstance( port_name )
print_port( port )
```

10 Print the DeviceID, the BusNumber, DeviceNumber, and FunctionNumber, the PhysicalSlot, and the ElementName properties of the upstream bridge or switch.

```
dev = connection.GetInstance( dev_name )
print_dev( dev )
```

11 Repeat from step 4.

A sample of the output looks like the following:

```
Chosen device:
 ID=PCI 0:4:1:0 B/D/F=4/1/0 Slot=0 (Mercuricity Generic USB OHCI Hub)
  (PCI-X port on bus 4)
    PCI port group for bus number 4
  (PCI-X port on bus 4)
 ID=PCI 0:3:3:0 B/D/F=3/3/0 Slot=2 (Plutonic Devices PD-631 PCI-X Bridge)
  (PCI-X port on bus 3)
    PCI port group for bus number 3
  (PCI-X port on bus 3)
 ID=PCI 0:0:1:0 B/D/F=0/1/0 Slot=0 (Plutonic Devices PD-631 PCI-X Bridge)
  (PCI port on bus 0)
```

Monitoring RAID Controller State

RAID controller state is useful to system administrators who need to monitor system health. This example shows how you can report the health state of RAID controllers on the managed server.

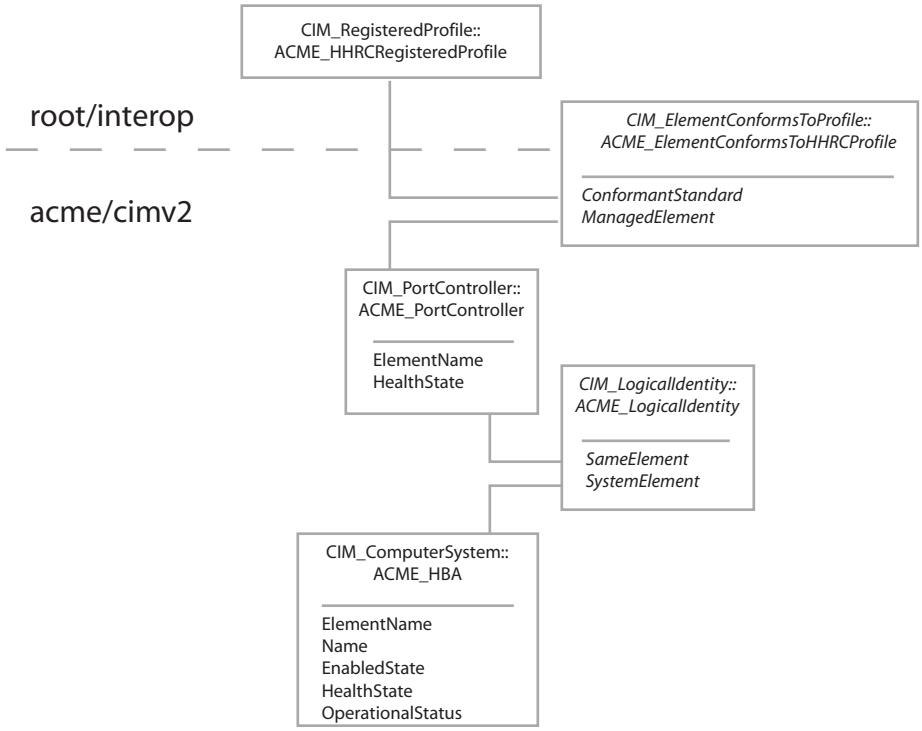
This example assumes you have installed a VIB that contains an implementation of the Host Hardware RAID profile, defined by the SNIA. VMware does not implement this profile, but prominent hardware vendors provide implementations for their storage controllers.

You can enumerate the controllers by starting from the Interop namespace and traversing associations from the Scoping Instance of the profile. Figure 3-13 shows the relationships of the CIM objects involved. Figure 3-13 uses a fictitious namespace and class names that begin with the prefix `ACME_`.

NOTE This example is consistent with versions of SMI-S prior to version 1.4. It is not consistent with version 1.5 or later. Early releases of SMI-S 1.4 are also consistent.

The `CIM_PortController` instance is logically identical to an instance of `CIM_ComputerSystem` subclassed as `ACME_HBA`. The `ACME_HBA` instance is the logical entity that is associated with the controller port objects.

Figure 3-13. Locating RAID Controllers



This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 16 and "Mapping Integer Property Values to Strings" on page 21.

To locate RAID controllers

1 Connect to the server URL.

Specify the Interop namespace, supplied as a parameter, for the connection.

```
use wbemlib
use sys
use connection renamed cnx
connection = Null

params = cnx.get_params()
if params is Null
   sys.exit(-1)
interop_params = params
interop_params['namespace'] = 'root/interop'
connection = cnx.connect_to_host( interop_params )
if connection is Null
   print 'Failed to connect to: ' + params['host'] + ' as user: ' + params['user']
   sys.exit(-1)
```

2 Locate the `CIM_RegisteredProfile` instance for the Host Hardware RAID Controller profile.

```
use registered_profiles renamed prof

profile_instance_name = prof.get_registered_profile_names( connection )
hhrc_instance_name = Null
for instance_name in profile_instance_names
   instance = connection.GetInstance( instance_name )
   if instance[ 'RegisteredName' ] == 'Host Hardware RAID Controller'
      hhrc_instance_name = instance_name
      break
if hhrc_instance_name is Null
   print 'Host Hardware RAID Controller profile not registered.'
   sys.exit(-1)
```

3 Traverse the `CIM_ElementConformsToProfile` association to reach the `CIM_PortController` instances for the Host Hardware RAID Controller profile on the managed server.

```
pc_instance_names = connection.AssociatorNames( hhrc_instance_name, \
                                                AssocClass = 'CIM_ElementConformsToProfile', \
                                                ResultClass = 'CIM_PortController' )
if len( pc_instance_names ) is 0
   print 'Error:  No RAID port controllers found.'
   sys.exit(-1)
```

4 For each port controller instance, traverse the `CIM_LogicalIdentity` association to reach the matching instance of `CIM_ComputerSystem` representing the RAID controller.

The `CIM_LogicalIdentity` mapping is 1:1, so the resulting array has only one element.

```
for pc_instance_name in pc_instance_names
   controller_instance_names = connection.AssociatorNames( pc_instance_name, \
                                                AssocClass = 'CIM_LogicalIdentity', \
                                                ResultClass = 'CIM_ComputerSystem' )
   cs_instance_name = controller_instance_names[ 0 ]
```

5 For the resulting controller instance, print the `ElementName`, `Name`, EnabledState, `HealthState`, and `OperationalStatus` properties.

This pseudocode provides default values for the properties. VMware cannot guarantee that your hardware vendor has implemented all the properties used in this example.

```
use value_mapper renamed map

   instance = connection.GetInstance( cs_instance_name )
   if instance.key( 'ElementName' )
      element_name = instance[ 'ElementName' ]
   else
      element_name = 'ElementName not available'
   if instance.key( 'Name' )
      name = instance[ 'Name' ]
   else
      name = 'Name not available'
   if instance.key( 'EnabledState' )
      enabled_state = map.map_instance_property_to_string( connection, \
                                                           instance, \
                                                           'EnabledState' )
   if not enabled_state
      enabled_state = 'not available'
   if instance.key( 'HealthState' )
      health_state = map.map_instance_property_to_string( connection, \
                                                          instance, \
                                                          'HealthState' )
   if not health_state
      health_state = 'not available'
   if instance.key( 'OperationalStatus' )
      operational_status = map.map_instance_property_to_string( connection, \
                                                                instance, \
                                                                'OperationalStatus' )
   if not operational_status
      operational_status = 'not available'
   print "%s (%s)' % ( element_name, name )
   print ' EnabledState: ' + enabled_state
   print ' HealthState: ' + health_state
   print ' OperationalStatus: ' + operational_status
```

A sample of the output looks like the following:

```
Controller 0 SAS/SATA (1F7D708944192F00)
  EnabledState: Enabled
  HealthState: Minor failure
  OperationalStatus: Degraded
```

Monitoring State of RAID Connections

This example shows how to report the connections of RAID controller initiators to targets on the managed server. RAID connection information is useful to system administrators who need to monitor system health.

This example assumes you have installed a VIB that contains an implementation of the Host Hardware RAID profile, defined by the SNIA. VMware does not implement this profile, but prominent hardware vendors provide implementations for their storage controllers.

This example assumes an implementation that models serial-attached SCSI connections to drives that belong to pooled RAID configurations. This model is similar to the SMI-S Host Hardware RAID Controller profile published by the SNIA. The model might or might not correspond to your hardware vendor's implementation.

Figure 3-14 shows the relationships of the CIM objects involved. Figure 3-14 uses a fictitious namespace and class names that begin with the prefix `ACME_`.

This example enumerates the connections of a controller by starting from the instance of `CIM_ComputerSystem` subclassed as `ACME_HBA` that represents the RAID controller. You must do this procedure for each disk controller that you monitor on the managed server. See "Monitoring RAID Controller State" on page 54 for information about locating the RAID controllers attached to a managed system.

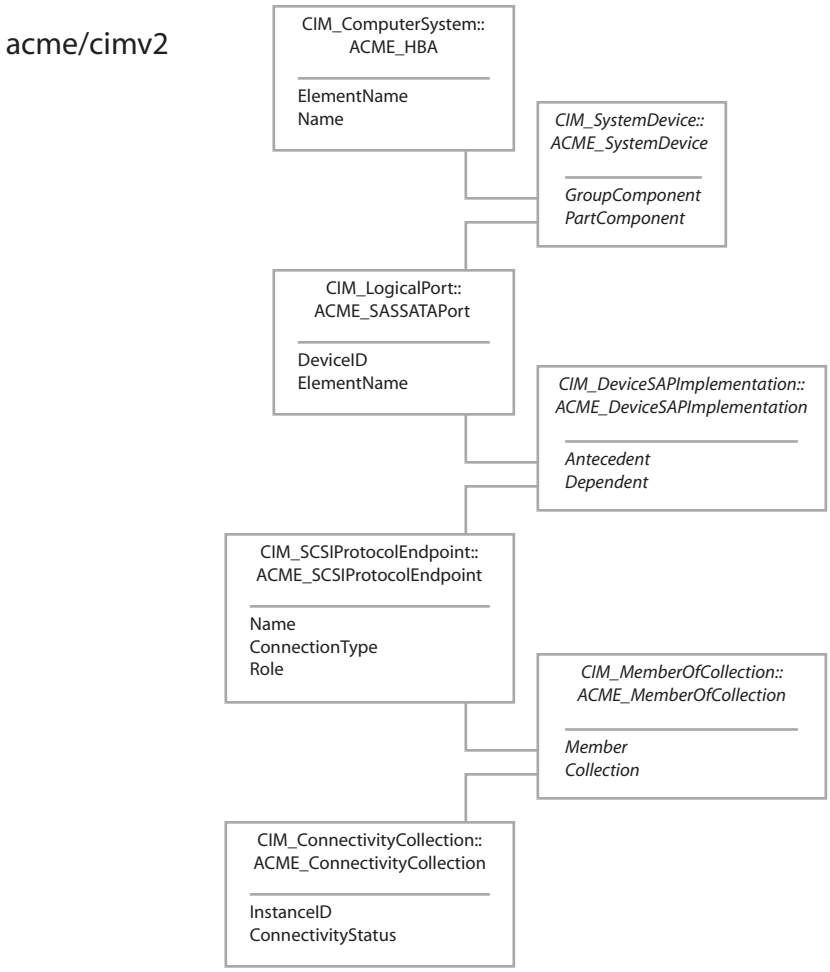
From the `ACME_HBA` instance, you traverse the `CIM_SystemDevice` association to the `CIM_LogicalPort` instances, then traverse the `CIM_DeviceSAPImplementation` association to the `CIM_SCSIProtocolEndpoint` instances.

The SMI-S specifies two different ways to model connections between targets and initiators. This example shows the simpler but less detailed choice.

Your hardware vendor's implementation might not follow this approach. Contact the hardware vendor for more information about the implementation.

This example traverses the `CIM_MemberOfCollection` association from the `CIM_SCSIProtocolEndpoint` to the `CIM_ConnectivityCollection` instance that represents a connection to a SCSI target. If your vendor's hardware implementation models the connection with the `CIM_SCSIInitiatortargetLogicalUnitPath` association, you can find connection status in that association instead of in the `CIM_ConnectivityCollection` instance.

Figure 3-14. Locating Connections Between HBA Initiators and Targets



This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 16 and "Mapping Integer Property Values to Strings" on page 21.

To report state of RAID connections

1 From a given instance of `CIM_ComputerSystem` that represents a SCSI controller, traverse the `CIM_SystemDevice` association to reach the `CIM_LogicalPort` instances on the managed server.

```
port_instance_names = connection.AssociatorNames( controller_instance_name, \
                                    AssocClass = 'CIM_SystemDevice', \
                                    ResultClass = 'CIM_LogicalPort' )
if len( port_instance_names ) is 0
   print 'Error:  No ports associated with controller.'
   sys.exit(-1)
```

2 For each logical port instance, traverse the `CIM_DeviceSAPImplementation` association to reach the matching instance of `CIM_SCSIProtocolEndpoint.`

```
for port_instance_name in port_instance_names
   init_instance_names = connection.AssociatorNames( port_instance_name, \
                                       AssocClass = 'CIM_DeviceSAPImplementation', \
                                       ResultClass = 'CIM_SCSIProtocolEndpoint' )
```

3 From the instance of `CIM_SCSIProtocolEndpoint`, traverse the `CIM_MemberOfCollection` association to reach the instance of `CIM_ConnectivityCollection` that represents the connection between initiator and target.

```
   for init_instance_name in init_instance_names
      conn_instance_names = connection.AssociatorNames( init_instance_name, \
                                        AssocClass = 'CIM_MemberOfCollection', \
                                        ResultClass = 'CIM_ConnectivityCollection' )
```

4 For the resulting instance of `CIM_ConnectivityCollection`, print the `InstanceID` and `ConnectivityStatus` properties.

```
      for instance_name in conn_instance_names
         print_scsi_connection_instance( connection, instance_name )

use value_mapper renamed map

function print_scsi_connection_instance( connection, instance_name
   health_state = connectivity_status = ''
   instance = connection.GetInstance( instance_name )
   if instance.key( 'InstanceID' )
      instance_id = instance[ 'InstanceID' ]
   else
      instance_id = 'InstanceID not available'
   if instance.key( 'ConnectivityStatus' )
      connectivity_status = map.map_instance_property_to_string( connection, \
                                                                 instance, \
                                                                 'ConnectivityStatus' )
   if not connectivity_status
      connectivity_status = 'not available'
   print '   Port connection ' + instance_id
   print '     ConnectivityStatus: ' + connectivity status
```

Reporting Accessible Storage Extents

This example shows how to report the disk storage extents that are accessible to a given SCSI controller. The information can be useful for configuring the managed servers in a datacenter.

This example assumes you have already located an instance of `CIM_ComputerSystem` subclassed as `ACME_Controller` that represents the RAID controller. See "Monitoring RAID Controller State" on page 54 for information about locating the RAID controllers attached to a managed system.

This example is based on the assumption that you have already installed a VIB that contains an implementation of the Host Hardware RAID profile, defined by the SNIA. VMware does not implement this profile, but prominent hardware vendors provide implementations for their storage controllers.

This example is based on the assumption that the implementation on the managed server models serial-attached SCSI connections to drives that belong to pooled RAID configurations. This model is similar to the SMI-S Host Hardware RAID Controller profile published by the SNIA.

The model might or might not correspond to your hardware vendor's implementation. Contact the hardware vendor for more information about the implementation.

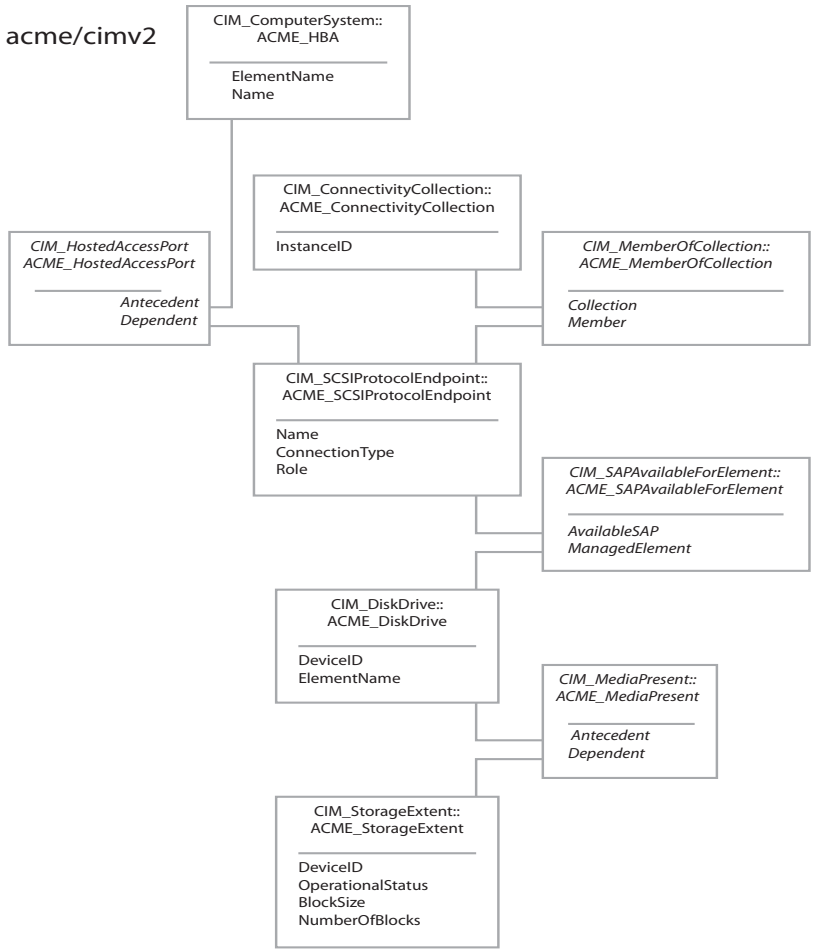
Figure 3-15 shows the relationships of the CIM objects involved. Figure 3-15 uses a fictitious namespace and class names that begin with the prefix `ACME_`.

The SMI-S specifies two different ways to model connections between targets and initiators. If your hardware vendor's implementation uses the `CIM_SCSIInitiatortargetLogicalUnitPath` association, you can follow the `LogicalUnit` reference of that association to get to the LUN directly.

Another way to locate disk storage extents is to start from each instance of `CIM_ConnectivityCollection` connected to the controller and to follow a series of associations to the disk media attached to the target endpoint. This procedure begins with the reverse of the last step in "Monitoring State of RAID Connections" on page 56, except that you need to filter on the value of the `Role` property to retrieve only targets, not initiators.

This example bypasses the issue of implementation choice by going from the SCSI controller to the target endpoints in one step by using the `CIM_HostedAccessPort` association. With this approach, the hardware vendor's choice of SMI-S implementation does not matter.

Figure 3-15. Locating Storage Extents Attached to SCSI Targets



This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 16 and "Mapping Integer Property Values to Strings" on page 21.

To report available storage extents

1 From a given instance of `CIM_ComputerSystem` subclassed as `ACME_HBA`, traverse the `CIM_HostedAccessPoint` association to reach the `CIM_SCSIProtocolEndpoint` instances on the managed server.

Use the value of the `Role` property to distinguish the target endpoints from the initiator endpoints. Values of 3 or 4 indicate that the endpoint functions as a target.

```
targ_instance_names = connection.AssociatorNames( controller_instance_name, \
                                    AssocClass = 'CIM_HostedAccessPoint', \
                                    ResultClass = 'CIM_SCSIProtocolEndpoint' )
if len( targ_instance_names ) is 0
   print 'Error:  No targets associated with SCSI controller instance.'
   sys.exit(-1)
for instance_name in targ_instance_names
   instance = connection.GetInstance( instance_name )
   if ( not ( instance['Role'] in [3, 4] ) )
      targ_instance_names.delete( instance_name )
```

2 For each target instance, traverse the `CIM_SAPAvailableForElement` association to reach the disk drive for the target.

```
for targ_instance_name in targ_instance_names
   disk_instance_names = connection.AssociatorNames( targ_instance_name, \
                                     AssocClass = 'CIM_SAPAvailableForElement', \
                                     ResultClass = 'CIM_DiskDrive' )
```

3 From `CIM_DiskDrive`, traverse the `CIM_MediaPresent` association to reach the storage extents that belong to that drive.

```
   for disk_instance_name in disk_instance_names
      ext_instance_names = connection.AssociatorNames( disk_instance_name, \
                                      AssocClass = 'CIM_MediaPresent', \
                                      ResultClass = 'CIM_StorageExtent' )
```

4 For each instance of `CIM_StorageExtent`, print the `DeviceID` and `OperationalStatus` properties. Also print the computed extent size (`BlockSize * NumberOfBlocks`), if those properties are available.

```
      for ext_instance_name in ext_instance_names
         print_extent( connection, ext_instance_name )

use value_mapper renamed mapper
function print_extent( connection, instance_name )
   instance = connection.GetInstance( instance_name )
   device_id = instance[ 'DeviceID' ]
   operational_status = ''
   status_codes = instance[ 'OperationalStatus' ]
   for status_code in status_codes
      value = mapper.map_instance_property_to_string( connection, \
                                                      instance, \
                                                      'OperationalStatus' )
      operational_status = operational_status + ' ' + value
   if instance.key( 'BlockSize' )
      block_size = instance[ 'BlockSize' ]
   else
      block_size = 0
   if instance.key( 'NumberOfBlocks' )
      num_blocks = instance[ 'NumberOfBlocks' ]
   else
      num_blocks = 0
   print 'Disk extent: ' + device_id
   print '   Operational status: ' + operational_status
   size = num_blocks * block_size
   if size
      print '   Size: " + size
```

Reporting Storage Extents Without Third-Party Storage Provider

This example shows how to report the disk storage extents that are available to a managed server, in the absence of a dedicated storage provider supplied by a storage vendor. Information about the storage extents is limited when a dedicated storage provider is not installed. The limited information can still be useful for configuring the managed servers in a datacenter.

You can locate disk storage extents by enumerating instances of `VMware_HypervisorStorageExtent` in the Implementation namespace. The pseudocode in this topic depends on the pseudocode in "Making a Connection to the CIMOM" on page 16.

To report storage extents

1 Connect to the server URL.

Specify the Implementation namespace, supplied as a parameter, for the connection.

The actual namespace you will use depends on your installation.

```
use wbemlib
use sys
use connection renamed cnx
connection = Null

params = cnx.get_params()
if params is Null
   sys.exit( -1 )
connection = cnx.connect_to_host( params )
if connection is Null
   print 'Failed to connect to: ' + params['host'] + ' as user: ' + params['user']
   sys.exit( -1 )
```

2 Enumerate instance names of `VMware_HypervisorStorageExtent`.

Select the instances where the `OtherIdentifyingInfo` property begins with `'/vmfs/devices/disks'`. For each such instance, print the `ElementName`, `OtherIdentifyingInfo`, and `OperationalStatus` properties.

```
use value_mapper renamed mapper
instances = connection.EnumerateInstances( 'VMware_HypervisorStorageExtent' )
for instance in instances
   if instance[ 'OtherIdentifyingInfo' ][ 0 ] begins '/vmfs/devices/disks'
      status = mapper.map_instance_property_to_string( connection, \
                                                       instance, \
                                                       'OperationalStatus' )
      print ' Storage Extent = ' + instance[ 'ElementName' ]
      print '   Other Info: ' + instance[ 'OtherIdentifyingInfo' ]
      print '   OperationalStatus: ' + status
```

A sample of the output looks like the following:

```
Storage Extent = Local Disk (naa.7001e4e041d08f00119991caf9fd2aaf)
  Other Info: /vmfs/devices/disks/naa.7001e4e041d08f00119991caf9fd2aaf
  OperationalStatus: OK
```

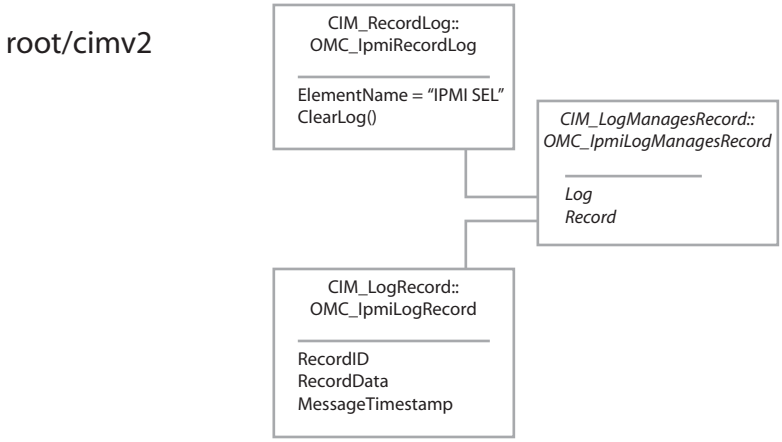
Working with the System Event Log

This example shows how to list the records in the system event log (SEL) of a managed server. This example also shows how to clear the records from the SEL. Clearing the log entries can save on disk space and reduce clutter from old records in the SEL.

You can locate the instance of `CIM_RecordLog` that represents the SEL by enumerating all instances of `CIM_RecordLog` and filtering out other logs by name. The log records are associated to the `CIM_RecordLog` instance. Figure 3-16 shows the relationships of the CIM objects involved.

NOTE This discussion assumes that the managed server is a single-node system.

Figure 3-16. Listing Records of the System Event Log



This example shows how to get the log entries from the Implementation namespace, assuming you already know its name. The pseudocode in this topic depends on the pseudocode in "Making a Connection to the CIMOM" on page 16.

To list and clear the System Event Log

1 Connect to the server URL.

Specify the Implementation namespace, supplied as a parameter, for the connection.

The actual namespace you will use depends on your installation.

```
use wbemlib
use sys
use connection renamed cnx
connection = Null

params = cnx.get_params()
if params is Null
   sys.exit( -1 )
connection = cnx.connect_to_host( params )
if connection is Null
   print 'Failed to connect to: ' + params['host'] + ' as user: ' + params['user']
   sys.exit( -1 )
```

2 Enumerate instance names of `CIM_RecordLog`.

```
instance_names = connection.EnumerateInstanceNames( 'CIM_RecordLog' )
if len( instance_names ) is 0
   print 'Error:  No logs found on managed server.'
   sys.exit( -1 )
```

3 Iterate over the log instances, rejecting all log instances that are not named `"IPMI SEL"`.

```
for instance_name in instance_names
   instance = connection.GetInstance( instance_name )
   if instance['ElementName'] is 'IPMI SEL'
      print_log_entries( instance_name )
      clear_log_entries( instance_name )
```

4 From the log instance that represents the SEL, traverse the `CIM_LogManagesRecord` association to reach the entries that belong to the log.

```
function print_log_entries( instance_name )
   instances = connection.Associators( instance_name,
                                       AssocClass = 'CIM_LogManagesRecord' )
   for instance in instances
      for prop in [ 'MessageTimestamp', 'RecordData' ]
         print ' %28s %s' % ( prop, instance[prop] )
```

5 On the log instance that represents the SEL, invoke the `ClearLog()` method with no parameters.

```
function clear_log_entries( instance_name )
   method_params = { }
   ( error_return, output ) = connection.InvokeMethod( 'ClearLog', \
                                                       instance_name, \
                                                       **method_params )
   if error_return is 0
      print 'Log entries cleared.'
   else
      print 'Failed to clear log entries; error = %s' % error_return
```

A sample of the output looks like the following:

```
Log contains 5 entries:
  MessageTimestamp 20090408014645.000000+000
    RecordData *81.0.32*1 0*2*5 2 220 73*32 0*4*16*81*false*111*2*255*255*1*
  MessageTimestamp 20090408014807.000000+000
    RecordData *3.0.32*2 0*2*87 2 220 73*32 0*4*1*3*false*1*87*149*129*1*
  MessageTimestamp 20090408015617.000000+000
    RecordData *3.0.32*3 0*2*65 4 220 73*32 0*4*1*3*false*1*89*149*129*1*
  MessageTimestamp 20090408020052.000000+000
    RecordData *3.0.32*4 0*2*84 5 220 73*32 0*4*1*3*false*1*89*149*129*1*
  MessageTimestamp 20090408020807.000000+000
    RecordData *3.0.32*5 0*2*7 7 220 73*32 0*4*1*3*false*1*89*150*129*1*
Log entries cleared.
```

Subscribing to Indications

ESXi 5.5 supports the following types of indications.

Table 3-2. Indications Supported by ESXi

Indication	Description
`OMC_IpmiAlertIndication`	Sent whenever entries are added to the IPMI System Event Log, and whenever a sensor's `HealthState` property becomes less healthy than previously seen.
`OMC_BatteryIpmiAlertIndication`	Specializes `OMC_IpmiAlertIndication`.
`OMC_BIOSIpmiAlertIndication`	Specializes `OMC_IpmiAlertIndication`.
`OMC_ChassisIpmiAlertIndication`	Specializes `OMC_IpmiAlertIndication`.
`OMC_CoolingUnitIpmiAlertIndication`	Specializes `OMC_IpmiAlertIndication`.
`OMC_DiskIpmiAlertIndication`	Specializes `OMC_IpmiAlertIndication`.
`OMC_MemoryIpmiAlertIndication`	Specializes `OMC_IpmiAlertIndication`.
`OMC_PowerIpmiAlertIndication`	Specializes `OMC_IpmiAlertIndication`.
`OMC_ProcessorIpmiAlertIndication`	Specializes `OMC_IpmiAlertIndication`.
`VMware_ConcreteJobCreation`	Notifies a listener when a new `VMware_ConcreteJob` has been created to monitor an asynchronous operation initiated by an extrinsic method.
`VMware_ConcreteJobModification`	Reports when the status of a `VMware_ConcreteJob` has changed. A change to a job indicates progress or completion, or that an error occurred during the asynchronous operation.
`VMware_ConcreteJobDeletion`	Notifies a listener when a `VMware_ConcreteJob` has been deleted by the provider for that job.
`VMware_KernelIPChangedIndication`	This indication is sent whenever the ESXi kernel IP address for the host has changed.

To receive CIM indications, you must have a running process that accepts indication messages and logs them or otherwise acts on them, depending on your application. You can use a commercial CIM indication consumer to do this. If you choose to implement your own indication consumer, see the following documents:

- DMTF's CIM Event Model White Paper at http://www.dmtf.org/standards/documents/CIM/DSP0107.pdf
- DMTF's Indications Profile specification at http://www.dmtf.org/standards/published_documents/DSP1054.pdf
- CIM indication specifications from your server supplier that are specific to the server model

The indication consumer must operate with a known URL. This URL is used when instantiating the IndicationHandler object.

Similarly, you must know which indication class to monitor. This information is used when instantiating the IndicationFilter object.

This example shows how to instantiate the objects needed to register for indications.

This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 16.

To subscribe to indications

1 Connect to the server URL.

Specify the Interop namespace for the connection.

```
use wbemlib
use sys
use connection renamed cnx
connection = Null

params = cnx.get_params()
if params is Null
   exit(-1)
interop_params = params
interop_params['namespace'] = 'root/interop'
connection = cnx.connect_to_host( interop_params )
if connection is Null
   print 'Failed to connect to: ' + params['host'] + ' as user: ' + params['user']
```

2 Build the URL for the indication consumer.

```
destination = 'http://' + params['consumer_host'] \
             + ':' + params['consumerPort'] + '/indications'
```

3 Create the IndicationHandler instance to represent the consumer.

```
handlerBindings = { \
   'SystemCreationClassName' : 'OMC_UnitaryComputerSystem', \
   'SystemName' : clientHost, \
   'Name':  'Org:Local', \
   'CreationClassName' : 'CIM_IndicationHandlerCIMXML' \
   }

handlerName = wbemlib.CIMInstanceName( \
   'CIM_IndicationHandlerCIMXML', \
   keybindings=handlerBindings, \
   namespace='root/interop' )

handlerInst = wbemlib.CIMInstance( \
   'CIM_IndicationHandlerCIMXML', \
   properties = handlerBindings, \
   path = handlerName )
handlerInst['Destination'] = destination

chandlerName = connection.CreateInstance( handlerInst )
```

Use a globally unique organization identifier in place of `Org`, and use an organizationally unique identifier in place of `Local`.

4 Create the IndicationFilter instance to specify the indication class (such as `CIM_AlertIndication`).

The `SourceNamespace` property of the filter must match the Implementation namespace of the indication provider. In this pseudocode, the namespace is `root/cimv2` but a third-party indication provider might use a different namespace.

```
filterBindings = { \
   'SystemCreationClassName' : 'OMC_UnitaryComputerSystem', \
   'SystemName' : clientHost, \
   'Name':  'Org:Local', \
   'CreationClassName' : 'CIM_IndicationFilter' \
   }

filterName = wbemlib.CIMInstanceName( \
   'CIM_IndicationFilter', \
   keybindings=filterBindings, \
   namespace='root/interop' )

filterInst = wbemlib.CIMInstance( \
   'CIM_IndicationFilter', \
   properties = filterBindings, \
   path = filterName )
   filterInst['SourceNamespace'] = 'root/cimv2'
   filterInst['Query'] = 'SELECT * FROM ' + params['className']
   filterInst['QueryLanguage'] = 'WQL'

   cfilterName = connection.CreateInstance( filterInst )
```

5 Create the IndicationSubscription association to link the filter with the handler.

```
subBindings = { 'Filter': cfilterName, \
                'Handler' : chandlerName }

   subName = wbemlib.CIMInstanceName( \
      'CIM_IndicationSubscription', \
      keybindings = subBindings, \
      namespace = 'root/interop' )

   subInst = wbemlib.CIMInstance( 'CIM_IndicationSubscription', \
                                  path = subName )
   subInst['Filter'] = cfilterName
   subInst['Handler'] = chandlerName


rsubName = connection.CreateInstance( subInst )
```

Troubleshooting CIM Connections 4

If you have trouble with connections between a CIM client and a CIM server, or between a CIM server and a process that consumes indications, you can try to diagnose and correct the trouble using this information.

This material is organized into two sections. One section applies to connections initiated by the client. The other section applies to connections initiated by the server when delivering indications.

- See "Connections from Client to CIM Server" on page 67 if your CIM client is unable to connect to the CIM server.
- See "Connections from CIM Server to Indication Consumer" on page 68 if your CIM client is able to connect to the CIM server and subscribe to indications, but the indications are not delivered.

Connections from Client to CIM Server

If your client fails to complete a connection to a CIM server, use these suggestions to help verify the connection parameters and the health of the CIM server.

Using SLP

Check the connection parameters using an SLP client (available on the Web). Run the SLP client on the same subnetwork as the managed server. Verify that the managed server advertises the expected CIM service and the correct URL.

Using a Web Browser

To verify that you can reach the CIM service at the advertised location, connect to the managed server with a Web browser. Use a URL of the form https://<cim-server.mydomain.com>:5989/ (substituting the name of the actual server), and verify that the server is responding on the expected port. Port 5989 is the default port for CIM-XML connections, but your installation might be different.

Using a Command-Line Interface

Using a command-line interface allows you to bypass any issues related to the correct invocation of the interface functions in a programmatic client.

For convenient interactive access to a CIM server, install `wbemcli`, available from http://sourceforge.net/project/showfiles.php?group_id=128809. Using `wbemcli`, you can invoke CIM operations from a shell.

To access a CIM server using the WS-Management protocol, install the `wsmancli` package, available from http://sourceforge.net/projects/openwsman/. Using `the wsman` command-line interface, you can invoke CIM operations from a shell.

Verifying User Authentication Credentials

If you are certain that the connection parameters are correct, verify the authentication parameters. To complete a connection, you must authenticate as a user that is known to the managed server.

Connect to the managed server through the console and check that your root password is correct. Then use that password to authenticate as the root user from your client.

NOTE If the managed server is in lockdown mode, you must authenticate using a CIM ticket obtained from vCenter Server. See *CIM Authentication for Lockdown Mode* for more information about using a CIM ticket to authenticate.

Rebooting the Server

If all your connection parameters are correct and you are certain of your authentication credentials but you still cannot complete a connection, reboot the managed server or restart the management agents on the server.

Using Correct Client Samples

If you are using sample clients supplied by VMware, check the documentation to be sure that the samples are intended to work with the CIM server to which you are trying to connect. The samples might hard-code parameters, such as the port and namespace, that affect the connection.

For example, the C++ code in the *CIM Storage Management API Programming Guide* connects to the CIM server included with ESX Server 3.0, but does not connect to the CIM server included with ESX 4 or later.

Using Other CIM Client Libraries

VMware does not test all available CIM client libraries with ESXi. If your CIM client cannot connect to the CIM server, try writing a test client for a different library. For example, http://.sourceforge.net has a number of CIM client libraries available.

Using the WS-Management Library

If you are unable to find a satisfactory client library to make a WBEM connection, use WS-Management. ESXi 6.5 includes a WS-Man server to support CIM operations.

VMware recommends using the Web Services for Management Perl Library for WS-Man clients. This library is included with the VMware vSphere SDK for Perl version 1.6 or higher. See http://www.vmware.com/support/pubs/sdk_pubs.html for more information about the vSphere SDK for Perl.

Connections from CIM Server to Indication Consumer

If your client can connect to a CIM server and subscribe to indications, but cannot receive indications, use these suggestions to try to resolve the problem.

Firewall Configuration

ESXi ships with a software firewall that is configured by default to block outgoing connection requests. When an indication is triggered, the producer cannot open a connection to the consumer unless the target port is opened in the firewall.

When you create an indication subscription, the CIMOM opens the corresponding port in the firewall for you. To check the firewall configuration, use these commands:

- `esxcli network firewall get`

  tells you whether the firewall is enabled.

- `esxcli network firewall ruleset list`

  tells you which specific services are enabled.

To disable or enable the firewall, use these commands:

- `esxcli network firewall set –e false`

  disables the firewall.

- `esxcli network firewall set –e true`

  enables the firewall.

It is also possible to create rulesets to open or close firewall ports manually. For information about manual firewall configuration for ESXi, see the *vSphere Security Guide*.

For information about the `esxcli` command set, see the manual *Getting Started with vSphere Command-Line Interfaces.*

System Event Log

Alert indications for a managed server rely primarily on the contents of the System Event Log (SEL). If the SEL is disabled, or if it is full and cannot accept new log entries, you will not receive most alert indications for new events.

If the SEL is full, system status is shown correctly in response to CIM queries, regardless of indication delivery. To receive indications when the SEL is not accepting new entries, you have the following options.

- Consult your hardware vendor's system documentation for instructions to clear the SEL.
- You can clear the SEL from a vSphere Client connected to vCenter Server. On the **Hardware** tab, choose **System Event Log** from the View menu and click **Reset event log**.

Creating Offline Bundles 5

Offline bundles contain a combination of VIBs and metadata used to update ESXi host software. Offline bundles are similar to depots, with the difference that offline bundles are available from a local file system rather than from a web server.

You can create an offline bundle from a depot using VMware vSphere PowerCLI.

Creating an Offline Bundle With VMware vSphere PowerCLI

Before you can create an offline bundle, you must install the PowerCLI software. vSphere PowerCLI 5.0 requires:

- .NET 2.0 Service Pack 1
- Windows PowerShell 1.0 or Windows PowerShell 2.0 RTM

You can download vSphere PowerCLI 5.0 from the VMware vSphere 5.0 web site.

To create an offline bundle using vSphere PowerCLI

1 Run vSphere PowerCLI.

Choose **Start > Programs > VMware > VMware vSphere PowerCLI > VMware vSphere PowerCLI**.

2 Select a software depot from which to create an offline bundle.

```
Add-ESXSoftwareDepot http://depot-server/build-123456/ESX
```

NOTE If you previously added a different software depot during this session, first remove it from the array of default software depots. Repeat the following commands until the `$DefaultSoftwareDepots` array is empty. Then select a software depot using the `Add-ESXSoftwareDepot` command.

```
Remove-ESXSoftwareDepot $DefaultSoftwareDepots[0]
$DefaultSoftwareDepots
```

3 Display a list of the array image profiles in the depot.

```
$profiles=Get-ESXImageProfile
$profiles
```

4 Find the array index of the Standard image profile and export it to an offline bundle.

```
Export-ESXImageProfile -ImageProfile $profiles[index] ‘
                       -ExportToBundle -FilePath “C:\ESX_bundle.zip”
```

For information about using the offline bundle to upgrade ESXi host software, see “Creating an Offline Bundle With VMware vSphere PowerCLI” on page 71.

For more information about using vSphere PowerCLI with image profiles, see *vSphere Installation and Setup*.

Index

V

W

X