CIM SMASH/Server Management API Programming Guide

VMware ESX 4.0

VMware ESXi version 4.0

EN-000101-00

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

© 2007–2009 VMware, Inc. All rights reserved. This product is protected by U.S. and international copyright and intellectual property laws. VMware products are covered by one or more patents listed at http://www.vmware.com/go/patents.

VMware, the VMware "boxes" logo and design, Virtual SMP, and VMotion are registered trademarks or trademarks of VMware, Inc. in the United States and/or other jurisdictions. All other marks and names mentioned herein may be trademarks of their respective companies.

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

Contents

About This Book 5

1 Introduction 7

Platform Product Support 7

Supported Protocols and Versions 7

CIM Version 7

SMASH Version 7

Supported Profiles 8

CIM and SMASH Resources Online 8

2 Developing Client Applications 9

Ports 9

Namespaces 9

Resource URIs 10

Locating a Server with SLP 10

Making a Connection to the CIMOM 11

Listing Registered Profiles 12

Identifying the Base Server Scoping Instance 13

3 Using the CIM Object Space 17

Reporting Manufacturer, Model, and Serial Number 17

Reporting Manufacturer, Model, and Serial Number Using Only the Implementation Namespace 19

Reporting the BIOS Version 20

Monitoring State for All Sensors 21

Monitoring State of All Sensors Using Only the Implementation Namespace 23

Reporting Fan Redundancy 24

Reporting CPU Cores and Threads 26

Reporting Empty Memory Slots Using Only the Implementation Namespace 28

Monitoring RAID Controller State 29

Monitoring State of RAID Connections 31

Reporting Available Storage Extents 33

Working with the System Event Log 34

Rebooting the Managed Server 36

Subscribing to Indications 37

Appendix: Troubleshooting Connections 41
Connections from Client to CIM Server 41
Using SLP 41
Using a Web Browser 41
Using a Command-Line Interface 41
Verifying User Authentication Credentials 41
Rebooting the Server 42
Using Correct Client Samples 42
Using Other CIM Client Libraries 42
Using the WS-Management Library 42
Connections from CIM Server to Indication Consumer 42
Firewall Configuration 42
Opening or Closing Ports in the Firewall 42

Glossary 43

Index 47

About This Book

The *CIM SMASH/Server Management API Programming Guide* provides information about developing applications using the CIM SMASH/Server Management API version 4.

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
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.

Table 1. Revision History

Revision	Description
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 vSphere server hardware with interfaces based on CIM standards.

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 1

VMware ESX/ESXi 4.0 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
- Power off host systems for maintenance

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" on page 7
- "Supported Protocols and Versions" on page 7

Platform Product Support

The VMware CIM SMASH/Server Management API is supported by ESX 4.0 and ESXi 4.0. Hardware compatibility for ESX/ESXi is documented in the hardware compatibility guides, available on the VMware Web site. See http://www.vmware.com/support/pubs/vi_pubs.html.

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. ESX/ESXi 4.0 is compatible with version 2.19.1 (experimental) of the CIM schema.

SMASH Version

The SMASH standard is maintained by the Server Management Working Group (SMWG) of the DMTF. ESX/ESXi 4.0 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 the Host Hardware RAID Controller profile from the SMI specification developed by the Storage Networking Industry Association (SNIA). The implementation uses SMI-S version 1.2.

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 ESX/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 http://www.vmware.com/support/developer/cim-sdk/4.0/profiledoc/.

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	1.0.0
Host Discovered Resources	N/A
Host Hardware RAID Controller	N/A
Host LAN Port	N/A
IP Interface	N/A
Physical Asset	1.0.0
Power State Management	N/A
Power Supply	1.0.0c
Profile Registration	1.0.0a
Record Log	1.0.0
Role Based Authorization	N/A
Sensors	1.0.0
Simple Identity Management	N/A
Software Inventory	1.0.0
System Memory	1.0.0f

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)

Developing Client Applications 2

This chapter provides the details you need to write a basic CIM client that allows you to connect to a CIM server. This chapter presents an outline, divided into several steps. Each step is illustrated with pseudocode. You can build on 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 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 17 for more information about accessing specific objects in the Implementation namespace.

This chapter includes the following topics:

- "Ports" on page 9
- "Namespaces" on page 9
- "Resource URIs" on page 10
- "Locating a Server with SLP" on page 10
- "Making a Connection to the CIMOM" on page 11
- "Listing Registered Profiles" on page 12
- "Identifying the Base Server Scoping Instance" on page 13

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 corresponding 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

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.

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

Table 2-2. ESX and ESXi Namespaces

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

IMPORTANT Prior to ESX 4.0, the Interop namespace for ESX (but not for ESXi) was `root/PG_interop`.

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.

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
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/

Locating a Server with SLP

If you do not know the URL to access the WBEM service of the CIMOM on the ESX/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

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.

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] + ' &lt;host&gt; &lt;user&gt; &lt;password&gt;
               [&lt;namespace&gt;]'
      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 12.

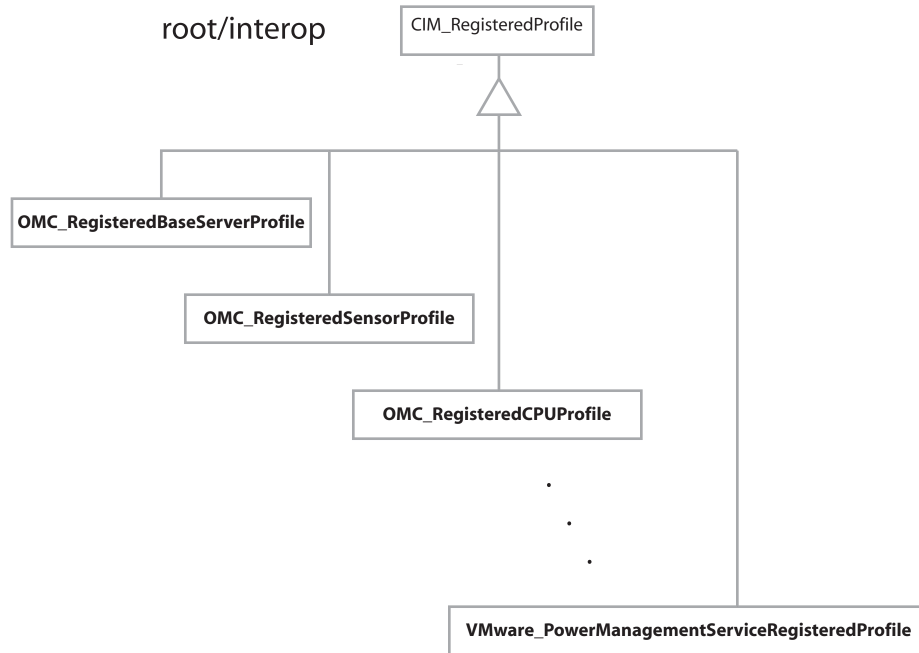
Listing Registered Profiles

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 representing 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 11.

To list registered profiles

1 Connect to the server URL, using the Interop namespace.

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

params = cnx.get_params()
params['namespace'] = 'root/interop'
if params is Null
   exit(-1)
connection = cnx.connect_to_host( 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
   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

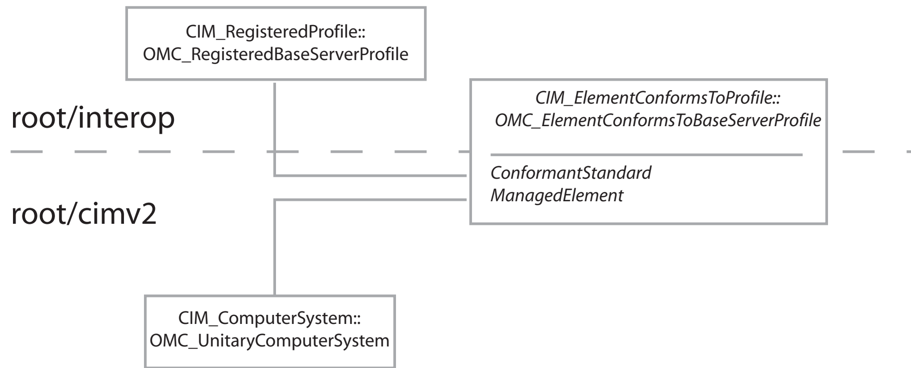
The Scoping Instance of `CIM_ComputerSystem` for the Base Server profile is an object representing the managed server. Various hardware and software components of the managed server are represented by CIM objects associated with this Scoping Instance.

A client can locate CIM objects 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 representing 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 11.

To identify the Base Server Scoping Instance

1 Make a connection to the CIMOM, using the Interop namespace.

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

params = cnx.get_params()
params['namespace'] = 'root/interop'  # Force the namespace.
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_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 corresponding 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 ) )
```

Using the CIM Object Space 3

This chapter presents examples that show how you can use the CIM object space to get information and manage a server that runs VMware ESX/ESXi. 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:

- "Reporting Manufacturer, Model, and Serial Number" on page 17
- "Reporting Manufacturer, Model, and Serial Number Using Only the Implementation Namespace" on page 19
- "Reporting the BIOS Version" on page 20
- "Monitoring State for All Sensors" on page 21
- "Monitoring State of All Sensors Using Only the Implementation Namespace" on page 23
- "Reporting Fan Redundancy" on page 24
- "Reporting CPU Cores and Threads" on page 26
- "Reporting Empty Memory Slots Using Only the Implementation Namespace" on page 28
- "Monitoring RAID Controller State" on page 29
- "Monitoring State of RAID Connections" on page 31
- "Reporting Available Storage Extents" on page 33
- "Working with the System Event Log" on page 34
- "Rebooting the Managed Server" on page 36
- "Subscribing to Indications" on page 37

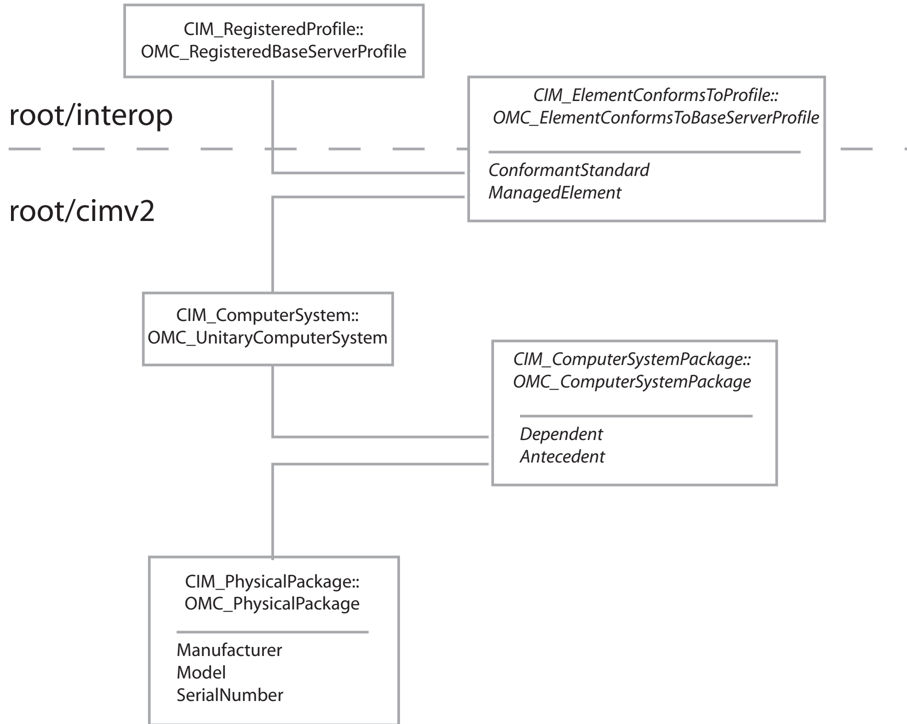
Many of the examples build on the basic steps described in "Developing Client Applications" on page 9.

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 monitoring configuration changes.

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

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



This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 11 and "Identifying the Base Server Scoping Instance" on page 13.

To report manufacturer, model, and serial number

1 Connect to the server URL, using the Interop namespace.

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

params = cnx.get_params()
params['namespace'] = 'root/interop'
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 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_PhysicalPackage` instance that corresponds to the managed server.

```
instance_names = connection.AssociatorNames( scoping_instance_name, \
                            AssocClass = 'CIM_ComputerSystemPackage', \
                            ResultRole = 'Antecedent' )
if len( instance_names ) &gt; 1
   print 'Error:  %d Physical Packages found for Scoping Instance.'  \
         % len ( instance_names )
   sys.exit(-1)
```

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

```
print '\n' + ' [' + instance.classname + '] ='
print '\n' + object_type + ' [' + instance.classname + '] -&gt;'
for prop in [ 'Manufacturer', 'Model', 'SerialNumber' ]
   print ' %30s = %s' % ( prop, instance[prop] )
```

A sample of output looks like the following:

```
Connecting to: server as user: admin
CIM_PhysicalPackage [OMC_Chassis] ->
                   Manufacturer = Dell Inc.
                          Model = PowerEdge 1900
                   SerialNumber = GYZ41D1
```

Reporting Manufacturer, Model, and Serial Number 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, enumerating and filtering `CIM_PhysicalPackage` for the managed server. This approach is convenient when the namespace is known in advance.

This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 11 and "Identifying the Base Server Scoping Instance" on page 13.

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

1 Connect to the server URL, using the Implementation namespace, supplied as a parameter. (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_PhysicalPackage` instances on the server.

```
object_type = 'CIM_PhysicalPackage'
list = connection.EnumerateInstances( object_type )
```

3 Select the instance whose `ElementName` property has the value `"Chassis"`, and print the `Manufacturer`, `Model`, and `SerialNumber` properties of the instance.

```
function print_info( instance )
   print '\n' + object_type + ' [' + instance.classname + '] -&gt;'
   for prop in [ 'Manufacturer', 'Model', 'SerialNumber' ]
      print ' %30s = %s' % ( prop, instance[prop] )

for instance in list
   if instance[ 'ElementName' ] == 'Chassis'
      print_info( instance )
      sys.exit(0)
print 'Unable to find Physical Package instance for Chassis.'
```

A sample of the output looks like the following:

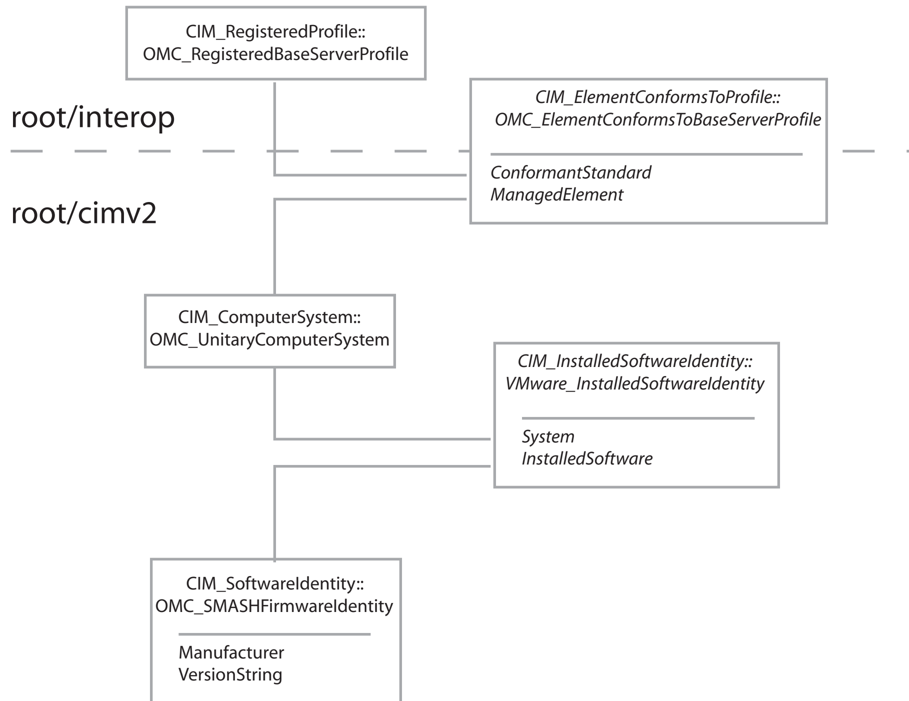
```
Connecting to: server as user: admin
CIM_PhysicalPackage [OMC_Chassis] ->
                  Manufacturer = Dell Inc.
                         Model = PowerEdge 1900
                  SerialNumber = GYZ41D1
```

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. VMware does not implement the `CIM_ElementSoftwareIdentity` association, so you must use `CIM_InstalledSoftwareIdentity` instead. Figure 3-2 shows the relationships of the CIM objects involved.

Figure 3-2. 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 the `MajorVersion` and `MinorVersion` properties.

This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 11 and "Identifying the Base Server Scoping Instance" on page 13.

To report the BIOS version

1 Connect to the server URL, using the Interop namespace.

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

params = cnx.get_params()
params['namespace'] = 'root/interop'
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 Locate the Base Server Scoping Instance representing 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, then print the `Manufacturer` and `VersionString` properties.

```
function print_info( instance )
   print '\n' + ' [' + instance.classname + '] ='
   print '\n' + object_type + ' [' + instance.classname + '] -&gt;'
   for prop in [ 'Manufacturer', 'VersionString' ]
      print ' %30s = %s' % ( prop, instance[prop] )

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

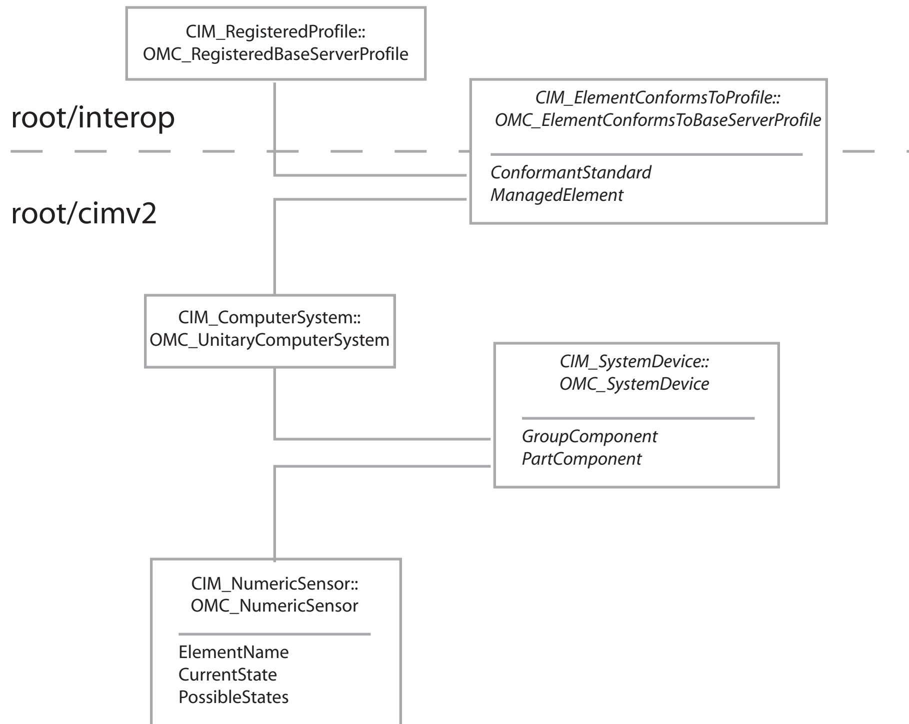
Monitoring State for All Sensors

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 3-3 shows the relationships of the CIM objects involved. For information about getting sensor states using only the Implementation namespace, see “Monitoring State of All Sensors Using Only the Implementation Namespace” on page 23.

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



This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 11 and "Identifying the Base Server Scoping Instance" on page 13.

To report state for all sensors

1 Connect to the server URL, using the Interop namespace.

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

params = cnx.get_params()
params['namespace'] = 'root/interop'
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 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 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 using the standard Interop namespace, see"Monitoring State for All Sensors" on page 21.

This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 11.

To report state of all sensors using only the Implementation namespace

1 Connect to the server URL, using the Implementation namespace, supplied as a parameter. (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`.

```
target_class = 'CIM_NumericSensor'
instances = connection.EnumerateInstances( target_class )
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' + target_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 )
```

A sample of the output looks like the following:

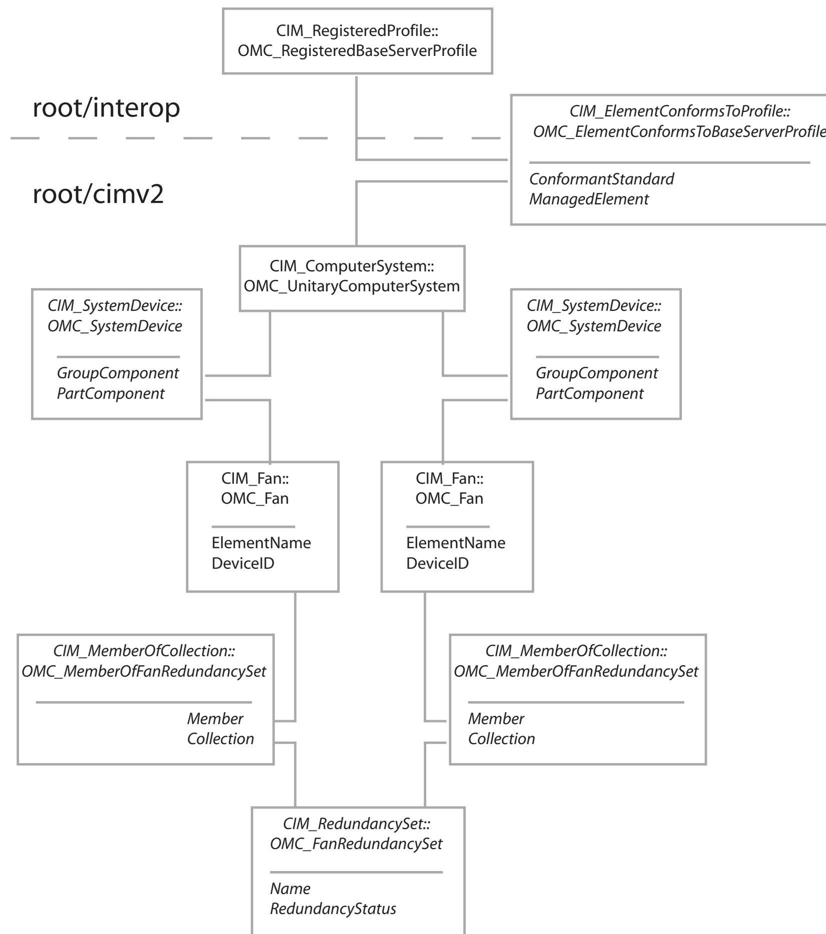
```
CIM_NumericSensor [OMC_NumericSensor] =
                    ElementName = FAN 1 RPM for System Board 1
                   CurrentState = Normal
CIM_NumericSensor [OMC_NumericSensor] =
                    ElementName = Ambient Temp for System Board 1
                   CurrentState = Normal
```

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 3-4 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 3-4. Locating Redundant Fans



This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 11 and "Identifying the Base Server Scoping Instance" on page 13.

To report fan redundancy

1 Connect to the server URL, using the Interop namespace.

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

params = cnx.get_params()
params['namespace'] = 'root/interop'
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 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.

```
target_class = 'CIM_Fan'
fan_instances = connection.Associators( scoping_instance_name, \
                                        AssocClass = 'CIM_SystemDevice', \
                                        ResultClass = target_class )
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' + target_class + ' [' + 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`.

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

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
```

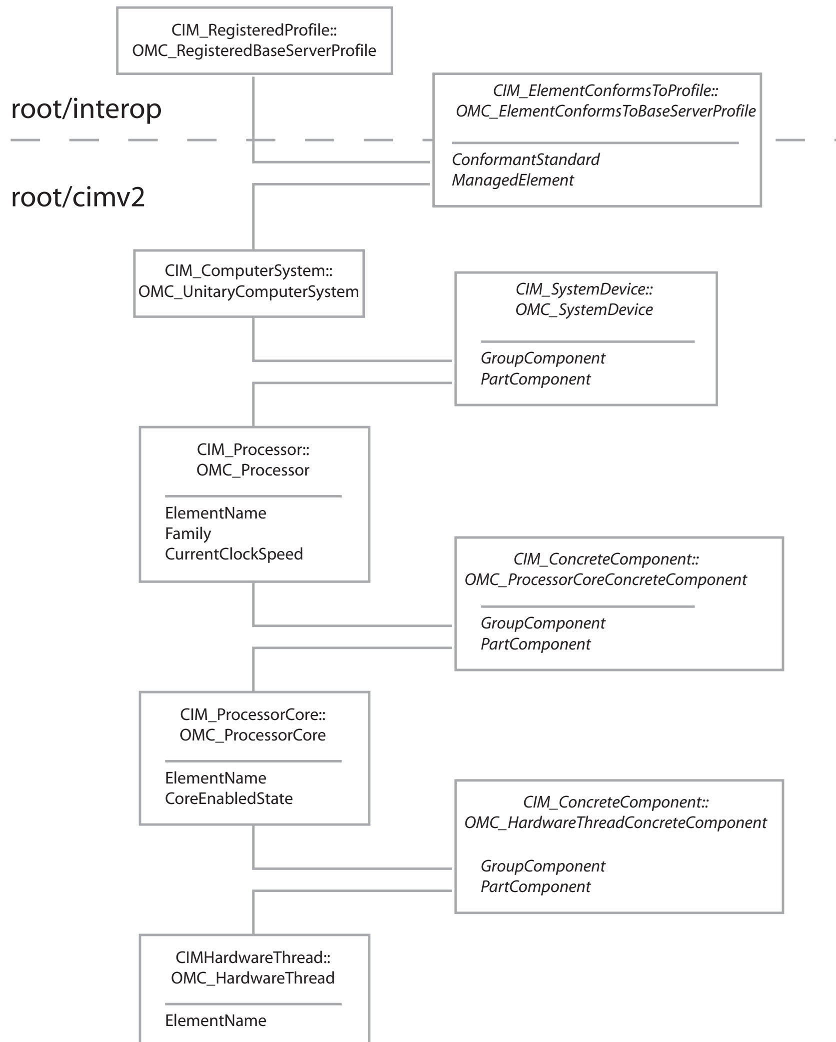
Reporting CPU Cores and Threads

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 3-5 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 3-5. Locating CPU Cores and Hardware Threads



This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 11 and "Identifying the Base Server Scoping Instance" on page 13.

To report CPU cores and threads

1 Connect to the server URL, using the Interop namespace.

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

params = cnx.get_params()
params['namespace'] = 'root/interop'
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 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.

```
target_class = 'CIM_Processor'
proc_instance_names = connection.AssociatorNames( scoping_instance_name, \
                                                  AssocClass = 'CIM_SystemDevice', \
                                                  ResultClass = target_class )
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.

```
target_class = 'CIM_ProcessorCore'
core_instance_names = connection.AssociatorNames( proc_instance_name, \
                                                  AssocClass = 'CIM_ConcreteComponent', \
                                                  ResultClass = target_class )
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.

```
target_class = 'CIM_HardwareThread'
thread_instance_names = connection.AssociatorNames( core_instance_name, \
                                                AssocClass = 'CIM_ConcreteComponent', \
                                                ResultClass = target_class )
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 Using Only the Implementation Namespace

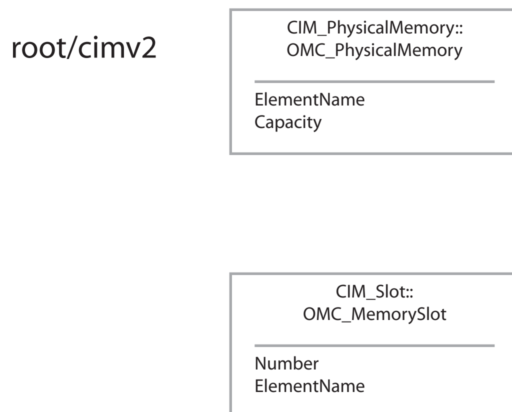
This example describes how to determine the empty slots available for new memory cards. This information is useful to system administrators who want to upgrade the capacity of a managed server.

This example shows how to locate information about the installed memory and available slots using only the objects in the Implementation namespace. Figure 3-6 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 3-6. Locating Physical Memory Slots



This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 11.

To report empty memory slots

1 Connect to the server URL, using the Implementation namespace.

```
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. Discard the instances that have matching `ElementName` properties. For others, 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
```

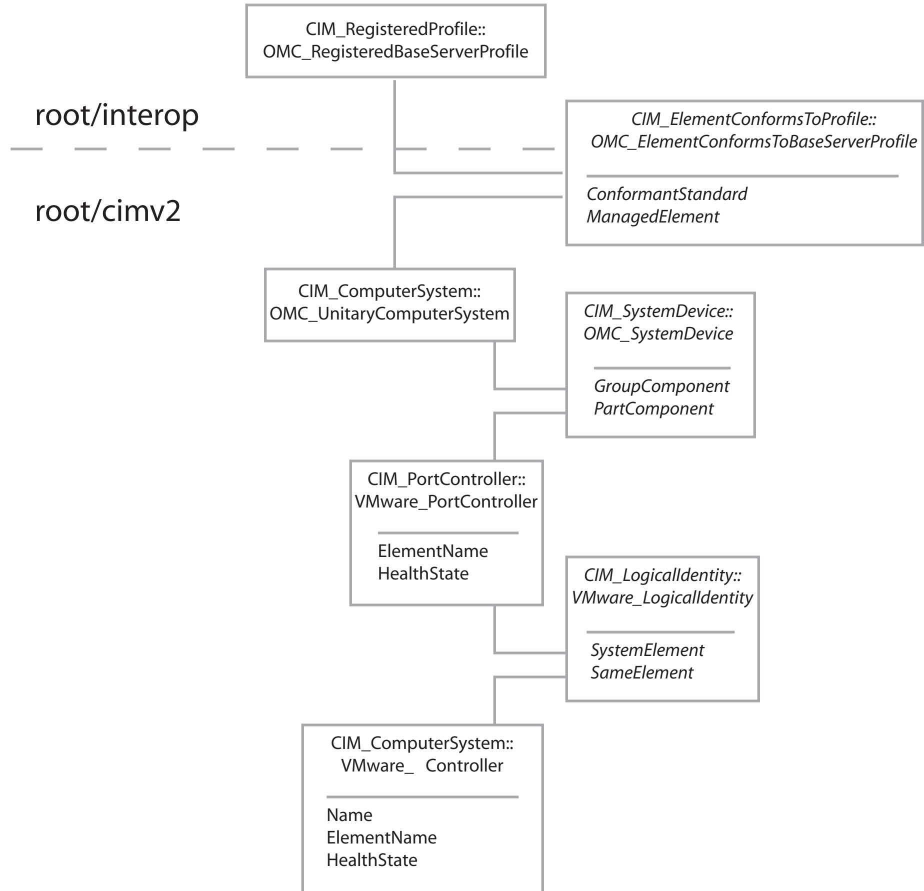
Monitoring RAID Controller State

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.

You can enumerate the controllers 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.

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

Figure 3-7. Locating RAID Controllers



This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 11 and "Identifying the Base Server Scoping Instance" on page 13.

1 Connect to the server URL, using the Interop namespace.

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

params = cnx.get_params()
params['namespace'] = 'root/interop'
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 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_PortController` instances on the managed server. Use the value of the `OtherControllerType` property to distinguish instances of HBA ports from other port types.

```
target_class = 'CIM_PortController'
pc_instance_names = connection.AssociatorNames( scoping_instance_name, \
                                              AssocClass = 'CIM_SystemDevice', \
                                              ResultClass = target_class )
if len( pc_instance_names ) is 0
   print 'Error:  No port controllers associated with server Scoping Instance.'
   sys.exit(-1)
for instance_name in pc_instance_names
   instance = connection.GetInstance( instance_name )
   if ( instance['OtherControllerType'] != 'SAS/SATA' )
      pc_instance_names.delete( instance_name )
```

4 For each port controller instance, traverse the `CIM_LogicalIdentity` association to reach the matching instance of `CIM_ComputerSystem` (subclassed as `VMware_Controller`).

```
for pc_instance_name in pc_instance_names
   target_class = 'CIM_ComputerSystem'
   controller_instance_names = connection.AssociatorNames( scoping_instance_name, \
                                                AssocClass = 'CIM_LogicalIdentity', \
                                                ResultClass = target_class )
```

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

```
for instance_name in controller_instance_names
   instance = connection.GetInstance( instance_name )
   print "%s (%s) health state: %s" % \
         ( instance['ElementName'], \
           instance['Name'], \
           (5==instance['HealthState']) ? 'OK' : 'Degraded' )
```

A sample of the output looks like the following:

```
Controller 0 (SAS5ira) health state: OK
Controller 1 (SAS5ira) health state: OK
```

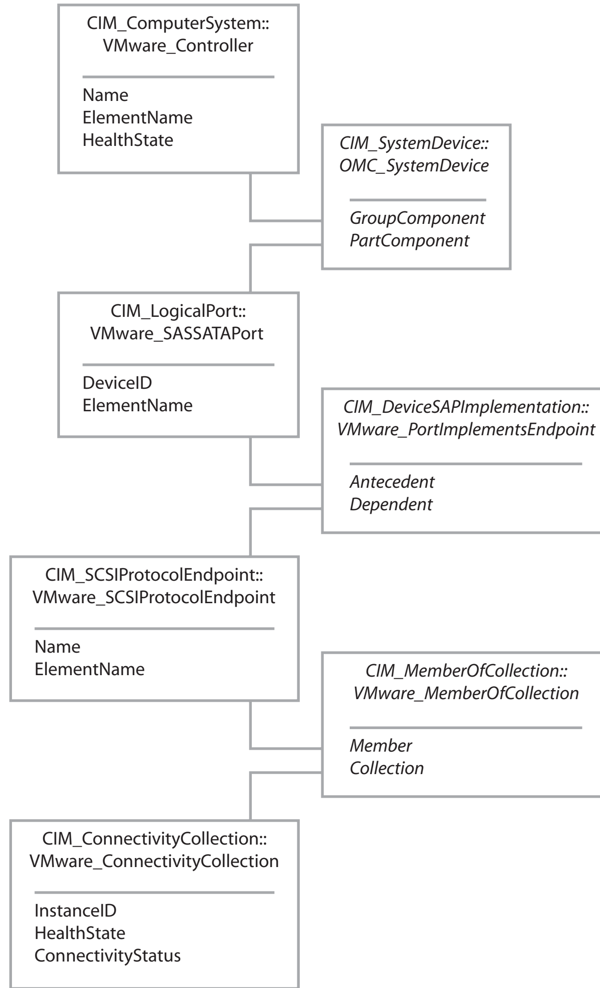
Monitoring State of RAID Connections

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.

The VMware implementation models serial-attached SCSI connections to drives that belong to pooled RAID configurations. The VMware model is similar, but not identical, to the SMI-S Host Hardware RAID Controller profile published by the SNIA.

You can enumerate the connections of a controller by starting from the instance of `CIM_ComputerSystem` subclassed as `VMware_Controller` that represents the RAID controller. From there, you traverse associations to the related `CIM_ConnectivityCollection` instances. Figure 3-8 shows the relationships of the CIM objects involved. You must do this procedure for each disk controller on the managed server. See "Monitoring RAID Controller State" on page 29 for information about locating the RAID controllers attached to a managed system.

Figure 3-8. Locating Connections Between HBA Initiators and Targets



To report state of RAID connections

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

```
target_class = 'CIM_LogicalPort'
port_instance_names = connection.AssociatorNames( controller_instance_name, \
                                     AssocClass = 'CIM_SystemDevice', \
                                     ResultClass = target_class )
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
   target_class = 'CIM_SCSIProtocolEndpoint'
   init_instance_names = connection.AssociatorNames( port_instance_name, \
                                        AssocClass = 'CIM_DeviceSAPImplementation', \
                                        ResultClass = target_class )
```

3 From the instance of `CIM_SCSIProtocolEndpoint`, traverse the `CIM_MemberOfCollection` association to reach the instances of `CIM_ConnectivityCollection` that represent the connection between the initiator and its targets.

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

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

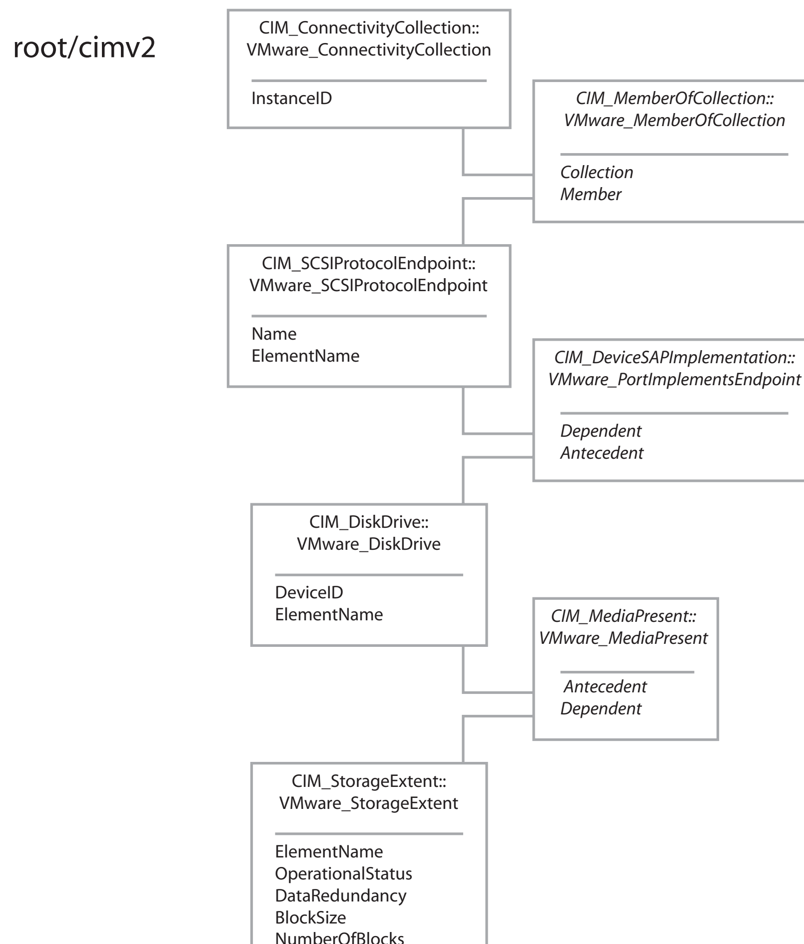
```
for instance_name in conn_instance_names
   instance = connection.GetInstance( instance_name )
   print "Port connection %s  health state: %s  connectivity status: %s" % \
      ( instance['InstanceID'], \
        (5==instance['HealthState']) ? 'OK' : '??', \
        (2==instance['ConnectivityStatus']) ? 'Up' : '??' )
```

Reporting Available Storage Extents

This example shows how to report the unused disk storage extents that are available to a managed server. Unused storage extents are those that do not belong to a storage pool implementing a RAID configuration. The information can be useful for configuring the managed servers in a datacenter.

You can locate disk storage extents by starting from each instance of `CIM_ConnectivityCollection` and following associations to the disk media attached to the target endpoint. Figure 3-9 shows the relationships of the CIM objects involved.

Figure 3-9. Locating Storage Extents Attached to SCSI Targets



This procedure assumes you have already located the SAS or SATA connections on a managed server. See "Monitoring State of RAID Connections" on page 31 for information about locating the connections of a RAID controller to targets on the managed system

To report available storage extents

1 From a given instance of `CIM_ConnectivityCollection`, traverse the `CIM_MemberOfCollection` association to reach the `CIM_SCSIProtocolEndpoint` instances on the managed server. Use the value of the `ElementName` property to distinguish the target endpoints from the initiator endpoints.

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

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

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

3 From `CIM_DiskDrive`, traverse the `CIM_MediaPresent` association to reach the storage extents that belong to that drive. Use the value of the `ElementName` property to identify the storage extents that already belong to a storage pool.

```
for disk_instance_name in disk_instance_names
   target_class = 'CIM_StorageExtent'
   ext_instance_names = connection.AssociatorNames( disk_instance_name, \
                                     AssocClass = 'CIM_MediaPresent', \
                                     ResultClass = target_class )
   for instance_name in ext_instance_names
      instance = connection.GetInstance( instance_name )
      if ( instance['ElementName'] not ends 'ONLINE' )
         ext_instance_names.delete( instance_name )
```

4 For each instance of `CIM_StorageExtent`, print the `ElementName`, `OperationalStatus`, and `DataRedundancy` properties. Also print the computed extent size (`BlockSize * NumberOfBlocks`).

```
for ext_instance_name in ext_instance_names
   instance = connection.GetInstance( ext_instance_name )
   print 'Disk extent: %s' % (instance['ElementName'])
   print '   Operational status: %s' % (instance['OperationalStatus'])
   print '   Redundancy: %s' % (instance['DataRedundancy'])
   print '   Size: %s" % (instance['BlockSize'] * instance['NumberOfBlocks'])
```

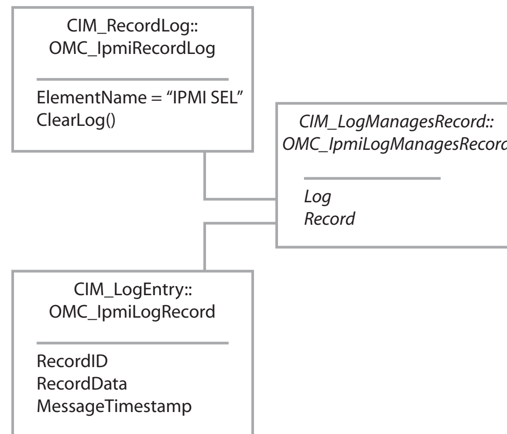
Working with the System Event Log

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 3-10 shows the relationships of the CIM objects involved.

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

Figure 3-10. 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 11.

To list and clear the System Event Log

1 Connect to the server URL, using the Implementation namespace, supplied as a parameter. (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`.

```
target_class = 'CIM_RecordLog'
instance_names = connection.EnumerateInstanceNames( target_class )
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 representing 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 representing 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.
```

Rebooting the Managed Server

Rebooting the managed server can be useful when you want to replace or upgrade hardware in the managed server.

This pseudocode depends on the pseudocode in "Making a Connection to the CIMOM" on page 11.

To reboot the managed server

1 Connect to the server URL, using the Interop namespace.

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

params = cnx.get_params()
params['namespace'] = 'root/interop'
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 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_AssociatedPowerManagementService` association to reach the `CIM_PowerManagementService` instance on the managed server.

```
instance_names = connection.AssociatorNames( scoping_instance_name, \
                 AssocClass = 'CIM_AssociatedPowerManagementService', \
                 ResultRole = 'ServiceProvided' )
if len( instance_names ) != 1
   print 'Error:  Unable to find PowerManagementService.'
   sys.exit(-1)
service_name = instance_names.pop()
```

4 Invoke the `RequestPowerStateChange()` method to request power state 5 (Power Cycle).

```
requested_state = 5
method_params = { 'PowerState' : requested_state, \
                  'ManagedElement' : scoping_instance_name }
( error_return, output ) = connection.InvokeMethod( \
                             'RequestPowerStateChange', \
                             service_name, \
                             **method_params )
```

The managed server might be left in maintenance mode after power cycling.

Subscribing to Indications

ESX/ESXi 4.0 supports the following types of indications.

Table 3-1. Indications Supported by ESX/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.
`VMware_AlertIndication`	Sent for events detected on the LSI Host Hardware RAID controller. All AEN data generates indications.
`VMware_KernelIPChangedIndication`	This indication is sent whenever the ESX/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 11.

To subscribe to indications

1 Connect to the server URL, using the Interop namespace.

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

params = cnx.get_params()
params['namespace'] = 'root/interop'
if params is Null
   exit(-1)
connection = cnx.connect_to_host(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' : 'IndicationHandlerName', \
   '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 = client.CreateInstance( handlerInst )
```

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

```
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 = client.CreateInstance( filterInst )
```

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

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 = client.CreateInstance( subInst )
```

Appendix: Troubleshooting Connections

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.

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.

Rebooting the Server

If all your connection parameters are correct, 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/ESXi 4.0.

Using Other CIM Client Libraries

VMware does not test all available CIM client libraries with ESX/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. ESX/ESXi 4.0 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

ESX 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.

This problem does not apply to ESXi, which does not have a firewall for outgoing connections.

Opening or Closing Ports in the Firewall

To open an outgoing port in the firewall

In the service console, use the following command to open a port for an HTTP connection between the indication producer and the indication consumer:

```
esxcfg-firewall -o <port_number>,tcp,out,http
```

To close an outgoing port in the firewall

In the service console, use the following command to close an outgoing HTTP port in the firewall:

```
esxcfg-firewall -c <port-number>,tcp,out,http
```

Glossary

A

API (application programming interface)

A set of functions that allows you to access a service programmatically.

B

BIOS (basic input/output system)

Firmware that controls machine startup and manages communication between the CPU and other devices, such as the keyboard, monitor, printers, and disk drives.

C

CIM (Common Information Model)

A collection of standards created by the DMTF to provide a shared model for managing systems. A set of object-oriented schemas, defined by the DMTF (Distributed Management Task Force), that provides a shared model for managing systems. The CIM schemas are not bound to any particular implementation. CIM defines how managed elements in a networked environment are represented as a common set of objects and relationships that multiple users can view, share, and control.

CIMOM (CIM Object Manager)

A service that provides standard CIM management functions over a WBEM connection to a CIM client. A CIMOM stores class definitions and populates requests for CIM operations with information returned from specific data providers. *See also* CIM (Common Information Model).

CIM-XML

A WBEM protocol defined by the DMTF for transporting XML over HTTP. *See also* DMTF (Distributed Management Task Force).

D

DMTF (Distributed Management Task Force)

An industry-wide consortium of hardware and software vendors with a mission to define interoperability standards. *See also* CIM (Common Information Model)

E

extrinsic method

A CIM method that is defined only for a specific CIM class. *See also* intrinsic method.

F

FRU (field replaceable unit)

A component that can be removed and replaced at the installed site.

I

IDL (interface definition language)

A human-readable syntax used to specify an API. An API described in the IDL can be compiled into stubs on a client machine. See also API (application programming interface), stub.

Implementation namespace

A logical location that contains most of the CIM classes and objects. The name is generally specific to the organization that supplies the CIM implementation, such as `acme/cimv2`. The name of the Implementation namespace can be discovered from associations with the Interop namespace.

indication
In the context of a system modeled by CIM, an asynchronous notification of an event, such as a state change or the deletion of an object monitored on behalf of a CIM client.

indication consumer
A process that receives CIM indication messages on a known port. In some systems, the consumer might be the same client process that subscribed to the indications. *See also* indication.

indication producer
A process that creates and sends CIM indication messages to report events for which one or more CIM clients have subscribed. *See also* indication.

initiator
In the context of storage subsystems, a host controller that communicates with devices on a storage network. *See also* target.

Interop namespace
A logical location containing a few key CIM classes and objects that have associations to the larger Implementation namespace. The Interop namespace generally has a well-known name, such as `root/interop`.

intrinsic method
A CIM method that can apply to any CIM class or object. *See also* extrinsic method.

IPMI (Intelligent Platform Management Interface)
A specification for system management interfaces that operate independently of system software. IPMI controllers in a managed system can interact with CIM providers to make system management functions available to CIM clients.

M

MOF
A file format for the CIM IDL that describes model classes. *See also* CIM (Common Information Model), IDL (interface definition language).

P

profile
A standardized collection of CIM classes selected and organized to model a particular management area.

provider
A module associated with a CIMOM that provides information in response to CIM client requests. The CIMOM dispatches service requests to one or more providers and returns an aggregated response to a CIM client.

S

SAN (storage area network)
A large-capacity network of storage devices that can be shared by a number of servers.

Scoping Instance
In CIM implementations, an instance that serves the key role in associations with other objects in a CIM profile.

SAS (Serial-attached SCSI)
A high-performance storage communication technology.

SATA (Serial Advanced Technology Attachment)
A standard, based on serial signaling technology, for connecting computers and hard drives. Also called "Serial ATA."

SEL (system event log)
An ongoing record of operational events occurring on a managed system.

SLP (Service Location Protocol)
A standard for advertising services available to a network.

SMASH (Systems Management Architecture for Server Hardware)
A collection of CIM profiles and communication protocols selected for datacenter management.

SMBIOS (System Management BIOS)
A DMTF standard for application access to system BIOS information. *See also* DMTF (Distributed Management Task Force), BIOS (basic input/output system).

SMI-S (Storage Management Initiative Specification)
A standard defined by SNIA for using a CIM client to manage a SAN. *See also* CIM (Common Information Model), SNIA (Storage Networking Industry Association).

SMWG (Server Management Working Group)
A DMTF committee responsible for the SMASH standard. *See also* DMTF (Distributed Management Task Force), SMASH (Systems Management Architecture for Server Hardware).

SNIA (Storage Networking Industry Association)
The trade organization responsible for the SMI-S profiles. *See also* SMI-S (Storage Management Initiative Specification).

SOAP (Simple Object Access Protocol)
A lightweight XML-based communication protocol that provides the messaging framework for Web services. SOAP specifies a standard way to encode parameters and return values in XML, and standard ways to pass them using common network protocols like HTTP (Web) and SMTP (email).

stub
A local procedure that implements the client side of a remote procedure call. The client calls the stub to perform a task. The stub packages the parameters. sends them over the network to the server, and returns the results to the client.

T

target
In the context of storage subsystems, a device controller that responds to commands on a storage network. *See also* initiator.

W

WBEM (Web-Based Enterprise Management)
A standard for message-based system management over HTTP.

WS-Man (Web Services-Management)
A standard for system management using SOAP over HTTP. *See also* SOAP (Simple Object Access Protocol).

Index

A

associations
- CIM_AssociatedPowerManagementService **37**
- CIM_ComputerSystemPackage **18**
- CIM_ConcreteComponent **27**, **28**
- CIM_DeviceSAPImplementation **32**, **34**
- CIM_ElementConformsToProfile **15**
- CIM_ElementSoftwareIdentity **20**
- CIM_IndicationFilter **38**
- CIM_IndicationSubscription **39**
- CIM_InstalledSoftwareIdentity **20**, **21**
- CIM_LogicalIdentity **31**
- CIM_MediaPresent **34**
- CIM_MemberOfCollection **25**, **33**, **34**
- CIM_SystemDevice **22**, **25**, **27**, **31**, **32**

authentication credentials **41**

B

Base Server profile **10**, **14**
BIOS version **20**, **21**

C

CIM object space **17**
CIM server **41**
CIM version **7**
CIMOM **8**, **10**
CIM-XML **7**
client applications, developing **9**
client libraries, CIM **42**
connection object, client **12**
connections
- CIM client to CIM server **41**
- CIM server to indication consumer **42**
- network **41**

connections, troubleshooting **41**
console access, managed server **41**
controllers, RAID **29**, **31**, **37**
cores
- *See* processor cores

CPU cores
- *See* processor cores

D

diagnosing connections **41**
DMTF **7**, **8**

E

esxcfg-firewall utility **42**
extents
- storage **33**, **34**

F

fans
- instances **25**
- redundancy **24**

firewall
- ports **42**
- software **42**

H

hardware threads **26**

I

Implementation namespace **9**, **10**, **12**, **13**, **19**
indication consumer **37**, **42**
indication producer **42**
indications **37**
initiators **31**, **33**, **34**
instances
- CIM_AlertIndication **38**
- CIM_ComputerSystem **22**, **25**, **27**, **29**, **30**, **31**, **32**, **36**
- CIM_ConnectivityCollection **31**, **33**, **34**
- CIM_DiskDrive **34**
- CIM_Fan **25**
- CIM_HardwareThread **26**, **28**
- CIM_LogicalPort **32**
- CIM_NumericSensor **21**, **22**, **23**
- CIM_PhysicalPackage **18**, **19**
- CIM_PortController **29**, **31**
- CIM_PowerManagementService **37**
- CIM_Processor **27**
- CIM_ProcessorCore **26**, **27**, **28**
- CIM_RecordLog **35**
- CIM_RedundancySet **24**, **25**
- CIM_RegisteredProfile **10**, **13**
- CIM_SCSIProtocolEndpoint **32**, **33**, **34**
- CIM_Sensor **21**, **23**, **24**
- CIM_SoftwareIdentity **20**, **21**

- CIM_StorageExtent **34**
- OMC_MemorySlot **28**, **29**
- OMC_PhysicalMemory **28**, **29**
- ProcessorCapabilities **26**
- VMware_Controller **29**, **31**

Interop namespace **10**, **11**, **12**, **14**
inventory, datacenter **17**

M

maintenance mode **37**
managed server **9**, **13**, **17**, **18**
- rebooting **36**–**37**

management agents **42**
manufacturer **17**, **19**
memory **28**
methods, extrinsic
- RequestPowerStateChange() **37**

model number **17**, **19**

N

namespace, XML **10**
namespaces, CIM **9**
- Implementation **9**, **10**, **12**, **13**, **19**
- Interop **10**, **11**, **12**, **14**

O

OMC **7**
online resources, CIM and SMASH **8**

P

platform product support **7**
ports
- CIM server **41**
- controller **29**, **31**
- network **10**, **42**

processor cores **26**
profiles
- Base Server **8**, **10**, **14**
- CPU **8**
- Ethernet Port **8**
- Fan **8**
- IP Interface **8**
- Power State Management **8**
- Power Supply **8**
- Profile Registration **8**
- Record Log **8**
- registered **10**, **12**
- Role Based Authorization **8**
- Sensors **8**
- Simple Identity Management **8**
- SMASH **8**, **10**
- Software Inventory **8**
- System Memory **8**
- versions **8**

properties
- BlockSize **34**
- ConectivityStatus **33**
- CoreEnabledState **27**
- CurrentClockSpeed **27**
- CurrentState **23**, **24**
- DataRedundancy **34**
- DeviceID **25**
- ElementName **19**, **23**, **24**, **25**, **27**, **28**, **29**, **31**, **34**
- Family **27**
- HealthState **31**, **33**, **37**
- InstanceID **33**
- MajorVersion **20**
- Manufacturer **19**, **21**
- MinorVersion **20**
- Model **19**
- Name **31**
- NumberOfBlocks **34**
- OperationalStatus **34**
- OtherControllerType **31**
- PossibleStates **23**
- RegisteredName **13**
- RegisteredVersion **13**
- SerialNumber **19**
- VersionString **20**, **21**

protocol and version support **7**

R

RAID configuration **33**
RAID controllers **29**, **31**, **37**
rebooting **36**, **42**
redundancy, fans **24**
registered profiles **10**, **12**
- listing **12**

resource URIs **10**

S

sample clients **42**
SAS **34**
SATA **34**
schema definitions **10**
Scoping Instance, Base Server **10**, **13**, **15**, **21**
SEL
- *See* System Event Log

sensors **21**, **23**
serial number **17**, **19**
server, managed **9**, **13**, **17**, **18**
- rebooting **36**–**37**

service console **42**
Service URL **10**

shell operations **41**
SLP **7**, **10**, **41**
SMASH profiles **8**
SMASH version **7**
SMI-S **8**, **31**
SMWG **7**, **8**
SNIA **8**, **31**
state
    RAID connections **31**
    RAID controller **29**
    sensors **21**, **23**
storage extents **33**, **34**
subnetwork **41**
subscribing to indications **37**
System Event Log **37**

T

targets **31**, **33**, **34**
technical support resources **6**
threads
    See hardware threads
troubleshooting connections **41**

U

URIs, resource **10**
URL
    CIM server **10**, **13**, **41**
    indication consumer **37**, **38**
    Service **10**

V

version
    BIOS **20**
    CIM **7**
    profiles **8**
vSphere SDK for Perl **42**

W

WBEM **10**
wbemcli utility **41**
Web Services for Management Perl Library **42**
WS-Management **7**

X

XML namespace **10**