OpenLMI CIM Provider HOWTO

John Dennis jdennis@redhat.com

3/30/2013

Contents

License

Introduction

CIM stands for Common Information Model. CIM is one component of WBEM (Web-Based Enterprise Management). WBEM is a technology suite allowing one to remotely enumerate the computing resources in an enterprise, query their state, modify their configuration and otherwise act upon those resources. People often use the term CIM when they are actually discussing WBEM. Technically CIM is only a schema and specification. The suite of specifications and technologies providing enterprise computer management based on the CIM model is WBEM.

CIM is an open standard under the auspices of DMTF (Distributed Management Task Force). WBEM Management tools based on CIM allow IT administrators to manage diverse computing resources in a heterogeneous environment. In addtion to CIM DMTF also manages many of the specifications related to WBEM.

If you've been asked to write a CIM provider this document will introduce you to the relevant technologies and guide you through the development process.

WBEM Components

Although technically CIM is only a schema definition the common usage of the term CIM is often taken to mean the collection of tools and technologies enabling computer management, i.e. WBEM. Without the overarching technology ecosystem that WBEM provides CIM would just be a paper abstraction. Lets briefly explore how these independent components fit together to form a WBEM management solution.

A manged system runs a service called the CIMOM, sometimes referred to as a broker. CIMOM stands for Common Information Model Object Manager. Typically the CIMOM is connected to the internet so that it can provide remote administration of the computer. The CIMOM will load a set of providers. Each provider is a software module dedicated to managing one type (i.e. class) of resource on the computer, for example network interfaces. There may be multiple instances of that resource class. The provider is responsible for managing all instances of that resource class. The provider in addition to providing information about a resource instance may optionally allow the resource instance to be configured or acted upon. It is the CIMOM which organizes all the providers on the system and grants a CIM client access to those providers. There are many CIMOM implementations available. Since all CIMOM's are supposed to follow the collection of WBEM standards all CIM clients should be able to inter-operate with each CIMOM.

A CIM provider can be loaded into different CIMOM implementations if both the CIMOM and the provider utilize a common programming API. CMPI (Common Manageability Programming Interface) is the standardized API all CIMOM's and provider's should be coded to.

A CIM client is able to connect to a CIMOM and interact with the resources (i.e. objects) being managed by the CIMOM. It is important to note *a CIM client does not interact directly with a provider* running in the CIMOM. Rather the CIMOM will expose to the CIM client the objects made available to the CIMOM via it's set of loaded providers.

Communication between a CIM client and a CIMOM occurs via standardized protocols. At the time of this writing only one protocol is in wide use, CIM Operations over HTTP. This protocol establishes HTTP headers and then passes the CIM payload as an XML document to the CIMOM. Responses from the CIMOM are also encoded in XML. The definition of the XML documents are defined in Representation of CIM in XML. The collection of standards used by CIM clients and CIMOM brokers are known as WBEM (Web-Based Enterprise Management).

In a typical scenario a CIM client will make an authenticated connection to a CIMOM and ask it to enumerate the set of objects it's interested in. The CIM client may choose to enumerate all instances of a specific class or it may utilize a query language to refine the results. CIM objects are often related to one another via Associations. Querying via associations is very similar to performing a join in SQL.

CIM objects have properties and methods. A property is a piece of data and a method is a function you can call. A CIM client may examine the properties of a returned object to determine it's health state, configuration, status, etc. Configuration is an advanced topic explained in greater depth in Resource Configuration.

A lot of information has been presented in a short time, to help clarify here is simple break down of how the components fit together.

1. CIM client makes authenticated connection to a CIMOM.

2. Using the HTTP protocol an XML document is passed from the CIM client to the CIMOM. The XML document describes the requested CIM operation. Formation of the XML document is performed by CIM client API library routines.
3. The CIMOM interprets the XML document and calls routines in it's providers to service the request.
4. The CIMOM communicates with it's providers using the CMPI C language API.
5. The CIMOM coalesces the information supplied by it's providers and forms an XML document which will then be returned as the HTTP response to the CIM client.
6. Library routines in the CIM client parse the XML document and return the information via a CIM client API.

What do I need to know to write a CIM provider?

CIM is an extraordinarily complex topic. Without some guidance one can easily get lost in the wealth of material resulting in spinning your wheels without making a lot of progress. In this section we try to summarize some key aspects of CIM and direct you to information that will help you complete your task while helping you stay clear of material which is not relevant.

Material deemed to be critical will be highlighted

In a single sentence like this.

CIM Schema, MOF and Profiles

CIM Schema and MOF Syntax

CIM models real world objects and their relationships. Those objects are modeled via CIM classes. A CIM class has properties and methods. The DMTF has defined a set of CIM classes which are meant to be the building blocks for a CIM model, this is the CIM Schema. The CIM Schema is expressed in the MOF Managed Object Format syntax. MOF files are used to define provider interfaces.

You must be fluent in MOF, it is the language of CIM.

Models and Profiles

A model is a collection of schema elements designed to model a computer system component which is to be managed. It defines the schema classes used to represent the managed elements and their relationships. At it's heart a model is pure CIM Schema but schema alone is not sufficient to explain intended usages nor the rules for how the schema elements interact. This expository material is collected into a document called a profile. Profiles follow a standardized format called the Management Profile Specification Template. DMTF has already defined numerous profiles to cover common computer system elements, these are collected in the Management Profiles web page. For some reason the primary CIM page on the DMTF website does not link to the management profiles. This curious omission might cause you to miss the critical aspect of CIM profiles.

Prior to starting a CIM provider peruse the Management Profiles to determine if one or more profiles already exist, if so you should implement that model.

Are you creating a model or implementing an existing profile?

If a profile already exists your job is tremendously simplified. The profile lays out the exact classes you have to implement along with the rules for how they interact. Chapter 9 of the profile is especially useful because it illustrates expected use cases with examples, this can greatly aid your comprehension of the model and it's profile. If a profile exists it is not necessary to understand the breath and depth of the CIM Schema, the necessary schema elements have already been assembled for you. At this point you can move on to the provider implementation tasks at this point using the profile as a recipe.

However if a profile does not yet exist for your provider you must define one. Unfortunately this is a very challenging task, it demands an in-depth understanding the CIM Schema as well as a working knowledge of the existing profiles. You need a familiarity with the existing profiles in order to understand the design patterns of CIM, otherwise your provider will not function as expected.

If you do find yourself in the position of having to author a model/profile then you should read the Using the Schema and Extending the Schema sections in this tutorial section. It will provide the conceptual framework for schema design decisions.

CMPI and KonkretCMPI

OpenLMI encourages the use of KonkretCMPI to aid provider development.

You will want to read the KonkretCMPI Documentation to understand the basic development process with KonkretCMPI.

You might be tempted after reading the KonkretCMPI documentation to dive and begin writing a provider believing KonkretCMPI given you all you need to know to complete the task. But the truth is you're not writing to a KonkretCMPI API, instead you're writing your provider using the CMPI API. Essentially what KonkretCMPI does is insulate you from CMPI. KonkretCMPI gives you a layer over CMPI that provides a nice level of abstraction and other utility support. The other primary advantage of KonkretCMPI is it automatically generates all the necessary stub functions needed to comply with CMPI. After KonkretCMPI runs you need to add your implementation to the function stubs KonkretCMPI generated for you. Overall this simplifies the development process.

However, if you don't have an understanding of CMPI from the outset you'll likely find what KonkretCMPI generates confusing because it will seem to exist in a vacuum. You won't necessarily understand how or why all the code pieces generated by KonkretCMPI fit together. You might find yourself asking were certain initialization is performed or in what order, how to manage life cycle, how are errors handled, etc. All of this is clearly spelled out in the CMPI spec. You'll probably also discover what KonkretCMPI gives to is incomplete, not everything you may need to do in your provider is covered by KonkretCMPI. There is an excellent chance you'll need to call the CMPI API directly for services not provided by KonkretCMPI. It's best to think of KonkretCMPI as a CMPI helper, it is not a CMPI replacement (i.e. wrapper around CMPI).

Therefore you should also read the CMPI Specification

The CMPI specification is long, here is my suggestion for reading it to get started. Read chapters 1-5, those chapters lay out the basic model and groundwork. If you understand those concepts you're in good shape. The remaining chapters list the function tables and detailed descriptions of each function in it's respective table. You don't need to know the details of each function, but it helps to know what is available. I would suggest perusing the beginning of each of these remaining chapters just to see what's available in table, you can skip the per function detail.

The CMPI specification describes the CMPI API at the raw C level. It can be very verbose code which is ripe for simplification through pre-processor macros. Standard macros are defined in `/usr/include/cmpi/cmpimacs.h`. Most code will use those macros and as such those macros are the *effective* CMPI API. You should be using these standard CMPI macros and be familiar with them for those cases where KonkretCMPI does not provide an alternative.

You should be familiar with the contents of `cmpimacs.h`.

Tutorials

At this point you now have enough background information to dive into the DMTF CIM Tutorial. This tutorial will help clarify the concepts already presented and round out material we have glossed over. The tutorial is brief and a bit superficial, it may or may not satisfy your needs for the comprehension of CIM.

Read the DMTF CIM Tutorial.

On the other hand the Learn CIM is much more complete and goes into much more depth. Not everyone will need the material here but many will find it helpful. A good strategy is to skim the tutorial making note of the material it covers and then later when confronted with a gap in your comprehension return to the tutorial.

Tools and Packages

CIMOM's

- OpenPegasus OpenPegasus is an open-source implementation of the DMTF CIM and WBEM standards. OpenPegasus is written in C++ and is designed to be portable. It builds and runs on most versions of UNIX, Linux, OpenVMS, and Microsoft Windows.

  RPM package name: tog-pegasus

  You may find the OpenPegasus Administrator's Guide useful for topics concerning installation, configuration, authentication, etc. of OpenPegasus.

- SFCB Small Footprint CIM Broker. SFCB is a CIM server for resource-constrained and embedded environments. It is written in C and designed to be modular and lightweight.

  RPM package name: sblim-sfcb

Development Tools

- KonkretCMPI is used to generate C source code for providers from a mof specification.

  RPM package name: konkretcmpi

- CMake is the required build tool. Various aspects of OpenLMI provider development and deployment are automatically handled via custom CMake macros provided by OpenLMI.

  RPM package name: cmake

- PyWBEM PyWBEM is a Python library supporting CIM operations over HTTP using the WBEM CIM-XML protocol. It is easy to use and master. PyWBEM also provides a Python provider interface suitable for rapid development of CIM providers.

  RPM package name: pywbem

- OpenLMI OpenLMI maintains numerous development tools to ease the task of CIM provider development, deployement and WBEM operations.

  RPM package name: openlmi-providers-devel

Client Tools

- PyWBEM can be used for client scripting in Python.
- YAWN Yet Another WBEM Navigator. YAWN runs in the Apache web server, it is Python based and utilizes Apache's mod-python. It is a CIM client tool in that it provides a way to browse CIM Schema and exercise CIM providers from within your local web browser.

  RPM package name: yawn

- OpenLMI OpenLMI maintain several client tools

  RPM package name: openlmi-tools

Writing a CIM provider for OpenLMI

OpenLMI Development Conventions

OpenLMI has established a number of development conventions which you will want to observe.

- The preferred CIMOM is OpenPegasus
- The preferred source code language for providers is C.
- KonkretCMPI is used to generate C source code for providers.
- CMake is the build tool. Various aspects of provider development and deployment are automatically handled via custom CMake macros provided by OpenLMI.
- PyWBEM is used for client scripting in Python and implementing providers when Python is the language of choice.

Set-Up Your Environment

This assumes you are working on a Fedora/RHEL/CentOS RPM based Linux distribution, although the basic concepts remain the same some package names and commands may differ slightly for other Linux distributions.

Install CIMOM

Install the OpenPegasus package

```
sudo yum install tog-pegasus
```

Make sure OpenPegasus is running. **Note:** provider registration only works when OpenPegasus is running.

```
sudo systemctl start tog-pegasus.service
```

If you want the OpenPegasus CIMOM service to automatically start after a reboot:

```
sudo systemctl enable tog-pegasus.service
```

Tip: pegasus can easily be controlled via the `cimserver` command, in fact the systemd service control mechanisms simply calls the `cimserver` command. During development you may find it easier to start, stop, and check the status of the OpenPegasus CIMOM service via `cimserver` rather than by the `systemctl` infrastructure.

Install YAWN

YAWN is not a requirement but being able to browse the CIM class hierarchy and invoke your provider from within your web browser is so handy most developers will want this. YAWN runs in the Apache web server therefore you will need to have Apache (e.g. httpd) installed and running to be able to browse at the following URL http://host/yawn where host is the host name you've installed YAWN on.

```
sudo yum install httpd yawn
```

Then make sure the Apache httpd service is running.

```
sudo systemctl start httpd.service
```

Optionally enable Apache httpd to start after booting.

```
sudo systemctl enable httpd.service
```

See the section on OpenPegasus authentication to understand the username and password prompts required by YAWN when you first connect.

Install Client Tools

OpenLMI Tools provides a CIM shell and a few other handy utilities.

```
sudo yum install openlmi-tools
```

Various WBEM command line utilities are provided by sblim-wbemcli

```
sudo yum install sblim-wbemcli
```

Python libraries which allow you to write simple Python scripts for WBEM operations are provided by pywbem.

```
sudo yum install pywbem
```

Install Provider Development tools

You will need KonkretCMPI, CMake and tools provided by OpenLMI.

```
sudo yum install cmake konkretcmpi openlmi-providers-devel
```

Begin Provider Development (C Language)

During this discussion we're going to use `XXX` as the name of your provider, you will need to substitute `XXX` for your provider name.

You must use CMake to take advantage of the OpenLMI development support. It expects the following structure in your development tree.

```
CMakeLists.txt
mof/LMI_XXX.mof
```

CMake will read the contents of CMakeLists.txt and produce a set of native Makefiles. CMake macros provided by openlmi-providers-devel will also set things up to invoke konkretcmpi to translate your LMI_XXX.mof file into a set of C source code files.

The defaults for CMake do not correspond to the defaults when CMake is invoked when producing OpenLMI RPM's. The goal here is not so much to match RPM but rather to produce a build that matches the expected system conventions and the OpenLMI conventions.

Since you probably won't be producing an RPM for your provider initially it's best to make sure CMake is configured to generate Makefiles that will build using the same conventions for building and installing as is done in RPM. There are various ways to do this but one simple technique is to define a shell script which invokes CMake with the matching RPM configuration, for example:

```
#!/bin/sh

/usr/bin/cmake -DCMAKE_VERBOSE_MAKEFILE=ON \
               -DCMAKE_INSTALL_PREFIX:PATH=/usr \
               -DINCLUDE_INSTALL_DIR:PATH=/usr/include \
               -DLIB_INSTALL_DIR:PATH=/usr/lib \
               -DSYSCONF_INSTALL_DIR:PATH=/etc \
               -DSHARE_INSTALL_PREFIX:PATH=/usr/share \
               -DBUILD_SHARED_LIBS:BOOL=ON .

if [ $? -eq 0 ]; then
    make
fi
```

During development it is useful to turn on debugging symbols and turn off optimization which complicate debugging using `gdb`. If you add

```
export CFLAGS='-g -O0'
```

to the above script before invoking cmake it will add these options to the Makefile. CMake has other mechanisms for producing debug builds (i.e. build targets). If you prefer those CMake mechanisms that's fine too but I found the above to be simple and expedient.

You will need a valid CMakeLists.txt file that follows the OpenLMI CMake conventions. Since OpenLMI is under active development the contents of the example CMakeLists.txt file may evolve or a CMakeLists.txt template may be included in the future but for the time being the example CMakeLists.txt represents a viable starting point.

Note: If you aren't familiar with CMake it can be difficult to figure out how to do a few simple things. For example if your provider is dependent upon another library how do you get things set up such that the include files are found and the right libraries are added when linking? In the following CMakeLists.txt example I've added a dependency on `glib2` only for the purposes of illustration, if your provider does not use `glib2` you won't need any of the items in the CMakeLists.txt file with the string "glib" in it. But you can use "glib" items as a model for what needs to be added for any dependency you do have. Also note that `glib2` installs pkgconfig files which define how to compile and link against `glib2`. Most library packages ship with pkgconfig files, the `glib2` example assumes pkgconfig files are available for the dependency, if your dependency does not provide pkgconfig files you'll have to adjust accordingly.

```
cmake_minimum_required(VERSION 2.6)
include(OpenLMIMacros)
find_package(CMPI REQUIRED)
find_package(KonkretCMPI REQUIRED)

find_package(PkgConfig QUIET)
pkg_check_modules(GLIB2  glib-2.0 REQUIRED)

add_subdirectory(mof)

set(PROVIDER_NAME XXX)
set(LIBRARY_NAME cmpiLMI_${PROVIDER_NAME})
set(MOF LMI_XXX.mof)


# Add all your .c source files here
set(provider_SRCS
    LMI_XXXProvider.c
)

set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -Wall")

konkretcmpi_generate(${MOF}
                     CIM_PROVIDERS
                     CIM_HEADERS
)

add_library(${LIBRARY_NAME} SHARED
            ${provider_SRCS}
            ${CIM_PROVIDERS}
            ${CIM_HEADERS}
)
```

```
# FIXME - /usr/include/openlmi shouldn't be hardcoded, needed for globals.h
# OpenLMI should provide a pkgconfig file
include_directories(${CMAKE_CURRENT_BINARY_DIR}
                    ${CMPI_INCLUDE_DIR}
                    ${GLIB2_INCLUDE_DIRS}
                    /usr/include/openlmi
                    )

target_link_libraries(${LIBRARY_NAME}
                      openlmicommon
                      ${KONKRETCMPI_LIBRARIES}
                      ${GLIB2_LIBRARIES}
                      )

# Create registration file
cim_registration(${PROVIDER_NAME} ${LIBRARY_NAME} ${MOF} share/openlmi-providers)

install(TARGETS ${LIBRARY_NAME} DESTINATION lib${LIB_SUFFIX}/cmpi/)
```

Perform a Build

Your mof/LMI_XXX.mof file will need to have been populated with something to start with. At this point you may want to refer to the KonkretCMPI Documentation to understand the basic development process with KonkretCMPI and follow it's example tutorial.

Understanding KonkretCMPI Behavior

The role of KonkretCMPI is to read a MOF file and generate C code definitions and code stubs necessary to implement your provider. KonkretCMPI generates 3 distinct sets of files:

- .h C header files
- .c C implementation files
- .reg CIMOM registration files

Anytime your mof file changes KonkretCMPI needs to run again to make sure the generated files are in sync with the mof file contents. If KonkretCMPI sees that you are missing any of the generated .c files it will generate the .c files for you. However if KonkretCMPI finds an existing .c file it will **not overwrite** the .c file. This is because you will have likely added your custom provider implementation code to these files and you don't want to lose your work.

The header files are directly tied to the definitions found in the mof file. It is essential those definitions be correct and reflect the current contents of the mof file. Therefore KonkretCMPI **always overwrites generated .h files**.

Tip: Do not add any custom content to any of the generated .h files, your content will be lost every time KonkretCMPI runs due to a mof file change. If you did add any custom content to a generated .h file you'll have to merge it back in, this is a pain. A better approach is to put custom header style information in an independent header file and include it in your .c file.

Note: Because KonkretCMPI will not overwrite an existing .c file you may need to merge function prototype information back into your .c file. You typically only need to do this when a CIM method signature is modified in the mof file or you've added a new CIM method. It's much easier to merge function prototypes back into your .c file than it is to move the .c file aside to allow KonkretCMPI to regenerate the .c file and then subsequently needing to merge all your custom code back into the newly generated .c file.

Tip: It's easier to do your provider development if your mof file is complete and exactly as you want it. This way you'll have fewer issues with KonkretCMPI regenerating files, needing to unregister and re-register your provider, or wondering why you can't see your updated mof (because you forget to redo the registration). But sometimes it's easier to develop your mof incrementally, you'll have to decide which development strategy better suits your style and situation.

Installing and Registering Your Provider

If all has gone well after typing `make` you will have compiled your .c files and linked them into a provider module. Now you need to install your provider so your CIMOM can load it.

In order for your CIMOM to expose your provider it must be registered with your CIMOM. This requires the following 3 files to have been installed in the expected location in your file system.

- provider module (i.e. your provider .so file)
- provider mof file
- provider registration file

The following command will install these files

```
sudo make install
```

Note: This is one reason it's critical to invoke CMake with the overridden defaults otherwise the installation directories will not default to the system defaults.

Now that the files are installed you must register your provider. OpenLMI provides a utility script `openlmi-mof-register` to simplify this task. **Note:** the `openlmi-mof-register` script is also used to unregister a provider, more on that in a moment.

Substitute `XXX` for the name of your provider.

```
sudo openlmi-mof-register register \
                          /usr/share/openlmi-providers/XXX.mof \
                          /usr/share/openlmi-providers/XXX.reg
```

Note: OpenPegasus must be running when you register or unregister a provider.

Tip: During your provider development cycle of edit/build/test you do not need to re-register your provider if your mof file did not change. It's sufficient to just install your updated provider .so (e.g. `sudo make install`) and restart the CIMOM. However if your mof file changed your CIMOM won't know about the mof changes because one of the things registration does is import the mof definitions into the CIMOM. After editing your mof you will need to unregister your old provider and re-register it again. In summary only after a mof modification you will need to do the following:

```
sudo openlmi-mof-register unregister \
                          /usr/share/openlmi-providers/XXX.mof \
                          /usr/share/openlmi-providers/XXX.reg
sudo make install
sudo openlmi-mof-register register \
                          /usr/share/openlmi-providers/XXX.mof \
                          /usr/share/openlmi-providers/XXX.reg
```

Testing Your Provider

Once your provider is installed and registered you will want to exercise it. Here are some simple ways you can do that, which one you choose is up to you, all boil down to invoking a CIM client against your CIMOM.

- Use YAWN in your web browser, open a URL (e.g. 'http://localhost/yawn) and navigate to one of your provider classes.
- Write a Python script using PyWBEM and run that script.
- Use the OpenLMI shell

Development Debugging Tricks and Techniques

Starting, Stopping and Controling OpenPegasus

Don't use the `systemctl` service command to stop and start OpenPegasus, instead use the `cimserver` command, it's much easier, plus you can specify one time configuration parameters useful for debugging (see below).

Run OpenPegasus In The Foreground

You can insert printf debugging statements in your code but you won't see them on the console unless you run OpenPegasus in a special way. Also, you will want to run OpenPegasus in the foreground, not allow it to fork a daemon process or spawn child processes. Running OpenPegasus in the following way is much friendlier during the development cycle.

```
sudo cimserver daemon=false forceProviderProcesses=false
```

When run this way you'll see any printf's you've added, they will appear in your console. When you're done with the current testing cycle simply control-c and OpenPegasus will exit. A rapid development cycle might look like this:

```
# Edit your provider source code
make
sudo make install
sudo cimserver daemon=false forceProviderProcesses=false
# Test via YAWN or a script
```

Using the Debugger

It's easy to set a breakpoint in your provider and have `gdb` break there for you. Of course you'll want to have compiled with -g to turn debugging symbols on and you'll probably also want to disable optimization with -O0 so that single stepping in the debugger follows your source code instead of jumping around.

Create a .gdbinit file in your local directory. Let's say you want to break on LMI_foobar

```
set breakpoint pending on
b LMI_foobar
r daemon=false forceProviderProcesses=false
```

For security reasons current versions of `gdb` require you to enable reading the the local .gdbinit file, one solution is to add this to your `~/.gdbinit` file:

```
add-auto-load-safe-path .
set auto-load local-gdbinit on
```

Then run OpenPegasus under `gdb`

```
sudo gdb /usr/sbin/cimserver
```

Exercise your provide in your preferred fashion and you should break. Then debug in `gdb` as you would normally do.

Of course like most things in life there are multiple ways of doing things, the above is just one suggestion, you could use gdb to attach to the running cimserver process or any number of other mechanisms, use your programming skills and knowledge to find a methodology that works best for you.

Controlling OpenPegasus Behavior

Advanced startup properties for CIMOM provides useful information about available options to control OpenPegasus behavior.

OpenPegasus Logging and Tracing

OpenPegasus has a tracing facility. You can utilize the CMPI logging commands to record debug and/or informational messages to the OpenPegasus trace file instead of printf statements. This is a cleaner solution once your provider more stable and you can dispense with temporary printf statements.

The CMPI logging and trace functions are `CMLogMessage` and `CMTraceMessage` and are defined in `/usr/include/cmpi/cmpimacs.h`, refer to that file for their usage.

But more importantly when you're baffled about what OpenPegasus is doing it can be invaluable to have full logging and trace information at your disposal to peruse. The trace file is:

```
/var/lib/Pegasus/cache/trace/cimserver.trc
```

To ratchet up the verbosity of the trace information you may want to run OpenPegasus like this:

```
sudo cimserver daemon=false forceProviderProcesses=false \
               logLevel=TRACE traceLevel=5 traceFacility=File \
               traceComponents=All
```

Detailed information about OpenPegasus tracing can be found in OpenPegasus Tracing User Guide

Dumping Method Parameters To the Console

Sometimes it's nice to be able to see the CIM method parameters being passed to your CIM method by dumping them to the console (of course you could also run under `gdb` too and break on the method). KonkretCMPI generates a Args_Print function in the generated .h file you can call.

To print method YYY args in konkret:

In XXX_DispatchMethod() in XXX.h

Add

XXX_YYY_Args_Print(&args, stdout);

after XXX_YYY_Args_InitFromArgs (otherwise args won't be initialized.)

Provider Development Tips

MOF Development Issues

Structures and Array of Structures as CIM Method Parameters

There is no way to pass a complex object in a CIM method call (i.e. a structure or class). CIM arrays are limited to simple scalar base types (int, string, etc.). Thus there is no way to pass things like (key,value) pairs directly. Instead one needs to define an array for the key names, and an array for the values (of a specific base type). To find the value of a key look up it's value at the same index in the as it appears in the key array. The same holds true for any array of structures, you have to decompose the structure members into individual arrays and recombine them back together by indexing into each array using the same index. Don't forget you'll need to declare the array with the `ArrayType ( "Indexed" )` qualifier in the MOF file. This is very reminiscent of programming in FORTRAN, ugh!

KonkretCMPI Oddities

KonkretCMPI doc uses this example invocation (note KonkretCMPI invocation is normally done via CMake macros)

```
konkret -s KC_Widget -m Widget.mof KC_Widget=Widget
```

But there is no man page describing what the args do in detail and the -h option is very terse and omits describing the final arg and for a long it was not clear to me what that arg was doing. The CMake macro konkretcmpi_generate does not use the same arg list as what is documented above which is also confusing.

Apparently the form used in the Konkret doc of `KC_Widget=Widget` is an alias mechanism which modifies the class name as found in the MOF file (lhs) to an alternate name (rhs) used in the generated C code. The type names, function names, generated file names etc. will all use the rhs alias, otherwise they will use the class name as found in the MOF.

OpenPegasus Authentication

The OpenPegasus Administrator's Guide gives a brief overview of how OpenPegasus handles authentication. But the following document which is installed along with the tog-pegasus package on Red Hat systems gives a more comprehensive overview.

```
/usr/share/doc/tog-pegasus-*/README.RedHat.Security
```

The short story is root user authentication works for local connections but is denied for network connections. If you've installed YAWN then the user authentication prompt issued by YAWN appears to OpenPegasus as a local user and root will work. However this is quite insecure and should be avoided. Root authentication is possibly justified in constrained cases such as during development where the target machine is on an isolated local network (i.e. virtual machines used for test and development).

The preferred mechanism is to use the `pegasus` user account which is created when tog-pegasus is installed. However there is no password established for the `pegasus` user during install (this is a security precaution) and you will need to set the `pegasus` user password (requires root privileges)

```
sudo passwd pegasus XXX
```

where XXX is the `pegasus` password. After this is done you can authenticate to OpenPegasus with the username `pegasus` and the password you created.

Advanced CIM Topics

Resource Configuration

How one handles configuration of CIM elements is a surprisingly complex topic and woefully under documented. If you're developing your own profile you'll need to understand these topics. The best way to learn about configuration approaches is to study the existing CIM profiles and see how they are handled in the example profiles.

One naive approach would be to provide a CIM method to set configuration parameters on your CIM object. This is a traditional approach in many programming API's. However the CIM Schema and existing models often take a different approach utilizing the following CIM classes and associations. This is probably worth entire tutorial on it's own. Here is a brief introduction:

The following classes are used as base classes to contain configuration parameters.

- `CIM_SettingData`
- `CIM_Capabilities`

The following *association* classes are used to form links between the `SettingData` and `Capability` derived classes.

- `CIM_ElementSettingData`
- `CIM_ElementCapabilities`
- `CIM_SettingsDefineCapabilities`
- `CIM_SettingsDefineState`

The way these can be combined is many fold. Naively you may assume you would have only one `SettingData` instance where the entirety of the configuration parameters are stored. But in fact you may have many such instances joined in a web by associations, some indicating current values, defaults values to be applied next, minimum values, maximum values, etc.

CIM_SettingData

ChangeableType Has the following possible values

- Not Changeable - Persistent
- Changeable - Transient
- Changeable - Persistent
- Not Changeable - Transient

CIM_SettingData is linked via CIM_SettingsDefineState and CIM_SettingsDefineCapabilities associations.

CIM_ElementSettingData CIM_ElementSettingData has the following properties:

IsDefault Has the following possible values

- Unknown
- Is Default
- Is Not Default

IsCurrent Has the following possible values

- Unknown
- Is Current
- Is Not Current

IsNext Has the following possible values

- Unknown
- Is Next
- Is Not Next
- Is Next For Single Use

CIM_ElementCapabilities

Characteristics[] Has the following possible simultaneous values

- Default
- Current

CIM_SettingsDefineCapabilities

PropertyPolicy Has the following possible values

- Independent
- Correlated

ValueRole Has the following possible values

- Default
- Optimal
- Mean
- Supported

ValueRange Has the following possible values

- Point
- Minimums
- Maximums
- Increments

Tying the Configuration Classes Together

If you're looking at a CIM_ElementSettingData association the `IsDefault` property will tell you if the group of configuration parameters pointed by the SettingData reference are the default values. Likewise the `IsCurrent` property tells you if the configuration parameters pointed by the SettingData reference are current values or not. The `IsNext` property tells you if the the configuration parameters pointed by the SettingData reference will be applied the next time configuration is applied and whether those parameters will permanently persist.

The CIM_ElementCapabilities association tells you if the CIM_Capabilities pointed to by the association for a CIM_ManagedElement (i.e. an object) are the defaults or the current values.

The CIM_SettingsDefineCapabilities association tells you how to interpret the SettingData being pointed to. There may be many SettingData objects needed to fully specify the configuration. The `PropertyPolicy` property tells you if you have to correlate the SettingData values or if you can treat them independently. The `ValueRole` property tells you what role the pointed to SettingData plays, i.e. defaults, optimal, average, etc. The `ValueRange` property tells you if the pointed to SettingData are a single set of values, just the minimum values, just the maximum values, or represent the increments each property value can be stepped by.

In practice what the `ValueRole` property does is force you to have many SettingData objects to specify the configuration for an element. Let's say your CIM element has some properties that can only be specified within a minimum and maximum range. You would then create a SettingData containing the valid minimums and point to it via a CIM_SettingsDefineCapabilities association. Likewise you would create a SettingData containing the valid maximums and point to it via a CIM_SettingsDefineCapabilities association. To ascertain the valid range you have to query for CIM_SettingsDefineCapabilities where the `ValueRange` property is Minimums, query for the Maximums and then follow the association pointers to each respective Capabilities to form the min/max range. By the same token a CIM_SettingsDefineCapabilities whose `ValueRange` property is Point indicates a single set of values rather than a range. Ultimately you have to find all the CIM_SettingsDefineCapabilities objects bound to the element you want to configure and interpret them.

Are you confused yet? It's very convoluted and the possible combinations are large. Don't you wish you could just call a method and set the configuration parameters or query them? The best way to wrap your head around all this is to study the various profiles utilizing these classes, especially study the use case examples in Chapter 9 of the profile, that will help solidify your understanding.

FAQ

Q: How do I make a CIM method a class method as opposed to a instance method?

A: An instance method is bound to the instance it is called from, in object oriented languages the instance is often called "self" or "this". This is the default method binding in CIM. However you can specify class methods as well which are not bound to an instance, to do this add the `Static` qualifier to the list of qualifiers belonging to the method.

Vocabulary

CIM Common Information Model is schema and associated specification which details how to represent the elements of a computer system in order to manage those elements. This yeilds a common and portable mechanism by which IT administrators can manage their computing resources. CIM is defined by the DMTF.

CIM Schema The CIM Schema is a collection of predefined CIM classes which forms the building blocks for modeling in CIM. The CIM Schema is expressed in MOF (Managed Object Format) syntax.

CIMOM Common Information Model Object Manager. Sometimes referred to as a broker the CIMOM is a network connected service running on a managed computer which grants access to the CIM providers on the managed computer. A CIM client connects to the CIMOM in order to manage a specific resource on the managed computer. Those resource instances are made available to the CIMOM by the providers loaded by the CIMOM.

CMPI Common Manageability Programming Interface. CMPI is an open standard defined by the Open Group which defines the programming API between a CIMOM and a CIM provider. In the absence of CMPI each CIM provider would need to be coded to the API of the CIMOM it was loaded into. CMPI allows a CIM provider to be written once and utilized by different CIMOM implementations.

DMTF Distributed Management Task Force is an industry consortium defining open standards for computer system management.

KonkretCMPI A tool used to aid development of CIM providers. KonkretCMPI reads a MOF specification file and generates a set of C header files, C program files and provider registration files. The primary purpose of KonkretCMPI is to insulate a provider author from the CMPI API by providing all the necessary “glue code” needed to adhere to the CMPI specification. This allows the programmer to focus on the particulars of the provider.

MOF Managed Object Format is the syntax used to describe the CIM Schema. MOF files are used to define provider interfaces.

Provider A software module which is loaded by the CIMOM broker running locally on a managed system which provides information about a type (i.e. class) of resource, for example network interfaces. There may be multiple instances of that resource class. The Provider is responsible for managing all instances of that resource class. The Provider in addition to providing information about a resource instance may optionally allow the resource instance to be configured or acted upon.

WBEM Web-Based Enterprise Management. A collection of standardized technologies providing unified management of distributed computing environments based on CIM concepts.