# Table of Contents

## 0.1 About This Manual

## 0.2 About The Manuals In General

## 0.3 Getting Administrator-Level Support

## 0.4 Getting Developer-Level Support

## 1 Bright Cluster Manager Python API

### 1.1 Installation

#### 1.1.1 Windows Clients

#### 1.1.2 Linux Clients

### 1.2 Examples

#### 1.2.1 First Program

### 1.3 Methods And Properties

#### 1.3.1 Viewing All Properties And Methods

#### 1.3.2 Property Lists

#### 1.3.3 Creating New Objects

#### 1.3.4 List Of Objects

#### 1.3.5 Useful Methods

#### 1.3.6 Useful Example Program

## 2 Metric Collections

### 2.1 Metric Collections Added Using cmsh

### 2.2 Metric Collections Initialization

### 2.3 Metric Collections Output During Regular Use

### 2.4 Metric Collections Error Handling

### 2.5 Metric Collections Consolidator Syntax

### 2.6 Metric Collections Environment Variables

### 2.7 Metric Collections Examples

### 2.8 Metric Collections On iDataPlex And Similar Units
Preface

Welcome to the Developer Manual for Bright Cluster Manager 7.0.

0.1 About This Manual

This manual is aimed at helping developers who would like to program the Bright Cluster Manager in order to enhance or alter its functionality. It is not intended for end users who simply wish to submit jobs that run programs to workload managers, which is discussed in the User Manual.

The developer is expected to be reasonably familiar with the parts of the Administrator Manual that is to be dealt with—primarily CMDaemon, of which cmsh and cmgui are the front ends.

This manual discusses the Python API to CMDaemon, and also covers how to program for metric collections.

0.2 About The Manuals In General

Regularly updated versions of the Bright Cluster Manager 7.0 manuals are available on updated clusters by default at /cm/shared/docs/cm. The latest updates are always online at http://support.brightcomputing.com/manuals.

- The Administrator Manual describes the general management of the cluster.
- The Installation Manual describes installation procedures for a basic cluster.
- The User Manual describes the user environment and how to submit jobs for the end user.
- The Cloudbursting Manual describes how to deploy the cloud capabilities of the cluster.
- The Developer Manual has useful information for developers who would like to program with Bright Cluster Manager.
- The OpenStack Deployment Manual describes how to deploy OpenStack with Bright Cluster Manager.
- The Hadoop Deployment Manual describes how to deploy Hadoop with Bright Cluster Manager.
- The UCS Deployment Manual describes how to deploy the Cisco UCS server with Bright Cluster Manager.

If the manuals are downloaded and kept in one local directory, then in most pdf viewers, clicking on a cross-reference in one manual that refers to a section in another manual opens and displays that section in the second manual. Navigating back and forth between documents is usually possible with keystrokes or mouse clicks.
For example: <Alt>-<Backarrow> in Acrobat Reader, or clicking on the bottom leftmost navigation button of xpdf, both navigate back to the previous document.

The manuals constantly evolve to keep up with the development of the Bright Cluster Manager environment and the addition of new hardware and/or applications. The manuals also regularly incorporate customer feedback. Administrator and user input is greatly valued at Bright Computing. So any comments, suggestions or corrections will be very gratefully accepted at manuals@brightcomputing.com.

0.3 Getting Administrator-Level Support

Unless the Bright Cluster Manager reseller offers support, support is provided by Bright Computing via e-mail at support@brightcomputing.com. Section 10.2 of the Administrator Manual has more details on working with support.

0.4 Getting Developer-Level Support

Developer support is given free, within reason. For more extensive support, Bright Computing can be contacted in order to arrange a support contract.
This chapter introduces the Python API of Bright Cluster Manager. For a head node `bright70`, the API reference documentation for all available objects is available in a default cluster via browser access to the URL:

https://bright70/userportal/downloads/python

The preceding access is via the User Portal (section 9.9 of the Administrator Manual).

The documentation is also available directly on the head node itself at:

file:///cm/local/docs/cmd/python/index.html

### 1.1 Installation

The Python cluster manager bindings are pre-installed on the head node.

#### 1.1.1 Windows Clients

For windows clients, Python version 2.5.X is needed. Newer versions of Python do not work with the API.

For Windows a redistributable package is supplied in the pythoncm-dist package installed on the cluster. The file at /cm/shared/apps/pythoncm/dist/windows-pythoncm.7.0.r15673.zip—the exact version number may differ—is copied to the Windows PC and unzipped.

A Windows shell (`cmd.exe`) is opened to the directory where the Python bindings are. The `headnodeinfo.py` example supplied with the unzipped files has a line that has the following format:

```python
cluster = clustermanager.addCluster(<parameters>);
```

where `<parameters>` is either:

```
'URL', 'PEMauth1', 'PEMauth2'
```

or

```
'URL', 'PFXauth', '', 'password'
```

The `<parameters>` entry is edited as follows:

© Bright Computing, Inc.
• the correct hostname is set for the <URL> entry. By default it is set to https://localhost:8081

• If PEM key files are to be used for client authentication,
  – <PEMAuth1> is set to path of cert.pem
  – <PEMAuth2> is set to the path of cert.key

• If a PFX file is used for client authentication,
  – <PFXauth> is set to path of admin.pfx
  – <password> is set to the password

To verify everything is working, it can be run as follows:
c:\python25\python headnodeinfo.py

1.1.2 Linux Clients
For Linux clients, a redistributable source package is supplied in the pythoncm-dist package installed on the cluster. The file at /cm/shared/apps/pythoncm/dist/pythoncm-7.0-r18836-src.tar.bz2—the exact version number may differ—is copied and untarred to any directory.

The build.sh script is then run to compile the source. About 4GB of memory is usually needed for compilation, and additional packages may be required for compilation to succeed. A list of packages needed to build Python cluster manager bindings can be found in the README file included with the package.

The headnodeinfo.py example supplied with the untarred files is edited as for in the earlier windows client example, for the clustermanager.addCluster line.

The path to the remote cluster manager library is added:

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:remotecm

To verify everything is working, the following can be run:

python ./headnodeinfo.py

1.2 Examples
A set of examples can be found in /cm/local/examples/cmd/python/ on the head node of the cluster.

1.2.1 First Program
A Python script is told to use the cluster manager bindings by importing pythoncm at the start of the script:

import pythoncm

If not working on the cluster, the administrator needs to set the path where the shared libraries can be found (pythoncm.so in Linux, or python.pyd in windows). This is done by adding the following to the start of the script:

import sys
sys.path.append(".")  # path to pythoncm.so/python.pyd
1.2 Examples

Python cluster manager bindings allow for simultaneous connections to several clusters. For this reason the first thing to do is to create a ClusterManager object:

```python
clustermanager = pythoncm.ClusterManager()
```

A connection to a cluster can now be made. There are two possible ways of connecting.

The first is using the certificate and private key file that cmsh uses by default when it authenticates from the head node.

```python
cluster = clustermanager.addCluster('https://mycluster:8081',
    '/root/.cm/admin.pem', '/root/.cm/admin.key');
```

The second way uses the password protected admin.pfx file, which is generated with the cmd -c command. A Python script could ask for the password and store it in a variable for increased security.

```python
cluster = clustermanager.addCluster('https://mycluster:8081',
    '/root/.cm/cmgui/admin.pfx', '', '<password>');
```

Having defined the cluster, a connection can now be made to it:

```python
isconnected = cluster.connect()
if !isconnected:
    print "Unable to connect"
    print cluster.getLastError()
    exit(1)
```

If a connection cannot be made, the function `cluster.connect()` returns false. The function `cluster.getLastError()` shows details about the problem. The two most likely problems are due to a wrong password setting or a firewall settings issue.

Similar to cmgui and cmsh, the cluster object contains a local cache of all objects. This cache will be filled automatically when the connection is established. All changes to properties will be done on these local copies and will be lost after the Python scripts exits, unless a commit operation is done.

The most common operation is finding specific objects in the cluster.

```python
active = cluster.find('active')
if active == None:
    print "Unable to find active head node"
    exit(1)
else:
    print "Hostname of the active head node is %s" % active.hostname
```

If creating an automated script that runs at certain times, then it is highly recommended to check if objects can be found. During a failover, for instance, there will be a period over a few minutes in which the active head node will not be set.

It is good practice to disconnect from the cluster at the end of the script.

```python
cluster.disconnect()
```

When connecting to a cluster with a failover setup, it is the shared IP address that should be connected to, and not the fixed IP address of either of the head nodes.
1.3 Methods And Properties

1.3.1 Viewing All Properties And Methods

All properties visible in cmsh and cmgui are also accessible from Python cluster manager bindings. The easiest way to get an overview of the methods and properties of an object is to define the following function:

```python
import re

def dump(obj):
    print "--- DUMP ---"
    for attr in dir(obj):
        p = re.compile('^__.*__$')
        if not p.match(attr):
            print "%s = %s" % (attr, getattr(obj, attr))
```

An overview of all properties and methods for the active head node can be obtained with:

```python
active = cluster.find('active')
dump(active)
```

1.3.2 Property Lists

Most properties are straightforward and their names are almost identical to the cmsh equivalent.

For instance:

```python
node.mac = '00:00:00:00:00:00'
category.softwareimage = cluster.find('testimage')
```

Properties that contain lists, like `node.roles`, `node.interfaces`, `category.fsmounts` and several others, are trickier to deal with. While iterating over a list property is simple enough:

```python
for role in node.roles:
    print role.name
```

because of an implementation restriction, adding a new role requires that a local copy of the roles list be made:

```python
roles = node.roles
provisioningrole = pythoncm.ProvisioningRole() # Create a new provisioning role object
roles.append(provisioningrole)
node.roles = roles # This will update the internal roles list with the local copy
```

1.3.3 Creating New Objects

Creating a new node can be done with:

```python
node = pythoncm.Node()
```

This is valid command, but fairly useless because a node has to be a MasterNode, PhysicalNode or VirtualSMPNode. So to create a normal compute or login node, the object is created as follows:

```python
node = pythoncm.PhysicalNode()
```

The first thing to do after creating a new object is to add it to a cluster.
cluster.add(node)

It is impossible to add one node to more than one cluster.

After the node has been added its properties can be set. In cmsh and cmgui this is semi-automated, but in Python cluster manager bindings it has to be done by hand.

    node.hostname = 'node001'
    node.partition = cluster.find('base')
    node.category = cluster.find('default')

Similar to the node object, a NetworkInterface object has several subtypes: NetworkPhysicalInterface, NetworkVLANInterface, NetworkAliasInterface, NetworkBondInterface, and NetworkIPMIInterface.

    interface = pythoncm.NetworkPhysicalInterface()
    interface.name = 'eth0'
    interface.ip = '10.141.0.1'
    interface.network = cluster.find('internalnet')
    node.interfaces = [interface]
    node.provisioningInterface = interface

Having set the properties of the new node, it can now be committed.

    cr = node.commit()

If a commit fails for some reason, the reason can be found:

    if not cr.result:
        print "Commit of %s failed:" % node.resolveName()
        for j in range(cr.count):
            print cr.getValidation(j).msg

1.3.4 List Of Objects

In the following lists of objects:

- Objects marked with (*) cannot be used
- Trees marked with (+) denote inheritance

Roles
    Role (*)
    + BackupRole
    + BootRole
    + DatabaseRole
    + EthernetSwitch
    + LoginRole
    + LSFClientRole
    + LSFServerRole
    + MasterRole
    + PbsProClientRole
    + PbsProServerRole
    + ProvisioningRole
    + SGECClientRole
    + SGESequenceRole
    + SlurmClientRole
    + SlurmServerRole
    + SubnetManagerRole
    + TorqueClientRole
    + TorqueServerRole

© Bright Computing, Inc.
Devices
Device (*)
+ Chassis
+ GpuUnit
+ GenericDevice
+ PowerDistributionUnit
+ Switch (*)
  + EthernetSwitch
  + IBSwitch
  + MyrinetSwitch
Node (*)
+ FSExport
+ FSMount
+ MasterNode
+ SlaveNode (*)
  + PhysicalNode
  + VirtualSMPNode

Network Interfaces
NetworkInterface (*)
+ NetworkAliasInterface
+ NetworkBondInterface
+ NetworkIpmiInterface
+ NetworkPhysicalInterface
+ NetworkVLANInterface

Information Objects
ClusterSetup
GuiClusterOverview
GuiCephOverview
GuiHadoopHDFSOverview
GuiOpenStackOverview
GuiOpenStackTenantOverview
GuiGpuUnitOverview
GuiNodeOverview
GuiNodeStatus
LicenseInfo
SysInfoCollector
VersionInfo

Monitoring Configuration Objects
MonConf
ConsolidatorConf
MonHealthConf
HealthCheck
MonMetricConf
ThreshActionConf
ThreshAction
Threshold

LDAP Objects
User
Group

Category Objects
Category
FSExport
FSMount
### 1.3 Methods And Properties

**Miscellaneous Objects**

- SoftwareImage
- KernelModule
- Network
- NodeGroup
- Partition

+ BurnConfig

**1.3.5 Useful Methods**

**For The Cluster Object:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>find(&lt;name&gt;)</td>
<td>Find the object with a given name, &lt;name&gt;</td>
</tr>
<tr>
<td>find(&lt;name&gt;, &lt;type&gt;)</td>
<td>Because it is possible to give a category and node the same name, sometimes the type &lt;type&gt; of the object needs to be specified too</td>
</tr>
<tr>
<td>getAll(&lt;type&gt;)</td>
<td>Get a list of all objects of a given type: e.g. device, category</td>
</tr>
<tr>
<td>activeMaster()</td>
<td>Get the active master object</td>
</tr>
<tr>
<td>passiveMaster()</td>
<td>Get the active master object</td>
</tr>
<tr>
<td>overview()</td>
<td>Get all the data shown in the cmgui cluster overview</td>
</tr>
<tr>
<td>add(&lt;object&gt;)</td>
<td>Add a newly created object &lt;object&gt; to the cluster. Only after an object is added can it be used</td>
</tr>
<tr>
<td>pexec(&lt;nodes&gt;, &lt;command&gt;)</td>
<td>Execute a command &lt;command&gt; on one or more nodes</td>
</tr>
</tbody>
</table>

**For Any Object:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>commit()</td>
<td>Save changes to the cluster</td>
</tr>
<tr>
<td>refresh()</td>
<td>Undo all changes and restore the object to its last saved state</td>
</tr>
<tr>
<td>remove()</td>
<td>Remove an object from the cluster</td>
</tr>
<tr>
<td>clone()</td>
<td>Make an identical copy. The newly created object is not added to a cluster yet</td>
</tr>
</tbody>
</table>
For Any Device:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>close()</td>
<td>Close a device</td>
</tr>
<tr>
<td>open()</td>
<td>Open a device</td>
</tr>
<tr>
<td>powerOn()</td>
<td>Power on a device</td>
</tr>
<tr>
<td>powerOff()</td>
<td>Power off a device</td>
</tr>
<tr>
<td>powerReset()</td>
<td>Power reset a device</td>
</tr>
<tr>
<td>latestMonitoringData()</td>
<td>Return a list of the most recent monitoring data</td>
</tr>
</tbody>
</table>

For Any Node:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>overview()</td>
<td>Get the data displayed in the cmgui node overview tab</td>
</tr>
<tr>
<td>sysinfo()</td>
<td>Get the data displayed in the cmgui node system information tab</td>
</tr>
<tr>
<td>pexec(&lt;command&gt;)</td>
<td>Execute a command</td>
</tr>
</tbody>
</table>

1.3.6 Useful Example Program

In the directory `/cm/local/examples/cmd/python` are some example programs using the python API. One of these is `printall.py`. It displays values for objects in an easily viewed way. With `all` as the argument, it displays resource objects defined in a list in the program. The objects are ‘Partition’, ‘MasterNode’, ‘SlaveNode’, ‘Category’, ‘SoftwareImage’, ‘Network’, ‘NodeGroup’. The output is displayed something like (some output elided):

Example

```
[root@bright70 ~]# cd /cm/local/examples/cmd/python
[root@bright70 python]# ./printall all
Partition base
  +- revision ......................
    | name .......................... base
    | clusterName .................... Bright 7.0 Cluster
...
| burnConfigs
  | +- revision ......................
    | | name ........................ default
    | | description .................. Standard burn test.
    | | configuration ................ < 2780 bytes >
    | +- revision ......................
    | | name ........................ long-hpl
...
| provisioningInterface ........ None
| fsmounts ........................ < none >
| fsexports                         |
| +- revision ......................
    | | name ........................ /cm/shared@internalnet
```
The values of a particular resource-level object, such as `softwareimage`, can be viewed by specifying it as the argument:

Example

```
[root@bright70 python]# ./printall.py softwareimage
softwareimage default-image
    +- revision ......................
    | | name .......................... default-image
    | | path .......................... /cm/images/default-image
    | | originalImage ................... 0
    | | kernelVersion ................... 2.6.32-431.11.2.el6.x86_64
    | | kernelParameters ................. rdblacklist=nouveau
    | | creationTime .................... 1398679806
    | | modules
    | |    +- revision ..................
    | |    | | name ...................... xen-netfront
    | |    | |    +- revision .............
    | |    | |    | | name .................... hpilo
    | |    | |    | | parameters ...........
    | |    | |    | | enableSOL .................... False
    | |    | |    | | SOLPort ................... ttyS1
    | |    | |    | | SOLSpeed ................... 115200
    | |    | |    | | SOLFlowControl ............... True
    | |    | |    | | notes ......................
    | |    | |    | | fspart ..................... 98784247812
    | |    | |    | | bootfspart ................. 98784247813
```

[root@bright70 python]#
This chapter gives details on metric collections. Section 9.4.4 of the Administrator Manual introduces metric collections, and describes how to add a metric collections script with cmgui. This chapter covers how to add a metric collections script with cmsh. It also describes the output specification of a metric collections script, along with example outputs, so that a metric collections script can be made by the administrator.

2.1 Metric Collections Added Using cmsh

A metric collections script, responsiveness, is added in the monitoring metrics mode just like any other metric.

Example

[...][monitoring->metrics]% add responsiveness
[...][monitoring->metrics]% set command /cm/local/apps/cmd/scripts/metrics/sample_responsiveness
[...][monitoring->metrics]% set classofmetric prototype; commit

For classofmetric, the value prototype is the class used to distinguish metric collections from normal metrics.

2.2 Metric Collections Initialization

When a metric collections script is added to CMDaemon for the first time, CMDaemon implicitly runs it with the --initialize flag. The output is used to define the collections table header structure. The structure is composed of the component metrics in the collections script, and the resulting structure is placed in the CMDaemon monitoring database. After the initialization step, data values can be added to the collections table during regular use of the script.

The displayed output of a metric collections script when using the --initialize flag is a list of available metrics and their parameter values. The format of each line in the list is:

```
metric <name[::parameter]> <unit> <class> "<description>" <cumulative> <min> <max>
```

© Bright Computing, Inc.
where the items in the line are:

- **metric**: A bare word.
- `<name[:parameter]>`: The name of the metric, with for certain metrics a parameter value. For example, the metric `AlertLevel` can have the parameter `sum` assigned to it with the `:` character.
- `<unit>`: The unit of measurement that the metric uses.
- `<class>`: Any of: misc cpu disk memory network environmental operating system internal workload cluster.
- `<description>`: This can contain spaces, but should be enclosed with quotes.
- `<cumulative>`: Either yes or no. This indicates whether the metric increases monotonically (e.g., bytes received) or not (e.g., temperature).
- `<min>` and `<max>`: The minimum and maximum numeric values of this metric are determined dynamically based on the values so far.

**Example**

```
[root@myheadnode metrics]# ./sample_responsiveness --initialize
metric util_sda % internal "Percentage of CPU time during which I/O requests were issued to device sda" no 0 100
metric await_sda ms internal "The average time (in milliseconds) for I/O requests issued to device sda to be served" no 0 500
```

**2.3 Metric Collections Output During Regular Use**

The output of a metric collection script without a flag is a list of outputs from the available metrics. The format of each line in the list is:

```
metric <name[:parameter]> <value> [infomessage]
```

where the parameters to the `metric` bare word are:

- `<name[:parameter]>`: The name of the metric, with optional parameter for some metrics.
- `<value>`: The numeric value of the measurement.

**Example**

```
[root@myheadnode metrics]# ./sample_responsiveness
metric await_sda 0.00
metric util_sda 0.00
[root@myheadnode metrics]#
```
2.4 Metric Collections Error Handling

If the output has more metrics than that suggested by when the --initialize flag is used, then the extra sampled data is discarded. If the output has fewer metrics, then the metrics are set to NaN (not a number) for the sample.

A metric or health check inside a metric collection appears as a check when viewing metrics or healthcheck lists. Attempting to remove such a check specifically using cmsh or cmgui only succeeds until the node is updated or rebooted. It is the metric collection itself that should have the check removed from within it, in order to remove the check from the list of checks permanently.

Setting a node that is UP to a CLOSED state, and then bringing it out of that state with the open command (section 5.5.4 of the Administrator Manual) also has CMDaemon run the metric collections script with the --initialize flag. This is useful for allowing CMDaemon to re-check what metrics in the collections can be sampled, and then re-configure them.

2.4 Metric Collections Error Handling

If the exit code of the script is 0, CMDaemon assumes that there is no error. So, with the --initialize flag active, despite no numeric value output, the script does not exit with an error.

If the exit code of the script is non-zero, the output of the script is assumed to be a diagnostic message and is passed to the head node. This shows up as an event in cmsh or cmgui.

For example, the sample_ipmi script uses the ipmi-sensors binary internally. Calling the binary directly returns an error code if the device has no IPMI configured. However, the sample_ipmi script in this case simply returns 0, and no output. The rationale here being that the administrator is aware of this situation and would not expect data from that IPMI anyway, let alone an error.

2.5 Metric Collections Consolidator Syntax

Metric collections can have a consolidator format defined per metric. The consolidator definition must be placed as an output in the line immediately preceding the corresponding metric initialization output line. The consolidator definition line can take the following forms:

consolidators default
consolidators none
consolidators CONSOLIDATORNAME FORMAT SPECIFICATION

The meanings of the texts after consolidators are as follows:

• default: The metrics follow the default consolidator names and interval values (section 9.7.4, page 386 of the Administrator Manual). That is, consolidator names take the value of Hour, Day, Week, while the interval values are the corresponding durations in seconds.

• consolidators none: No consolidation is done, only raw data values are collected for the metrics.
• **CONSOLIDATORNAME FORMAT SPECIFICATION:** This has the form:

\[
\text{name: interval[ :kind[ :tablelength]]}
\]

  - **name:** the consolidator name. A special feature here is that it can also define a new consolidator if the name does not already exist. Multiple consolidators can be defined in each consolidator definition line, with **name** separated from any preceding definition on the same line by a space.
  
  - **interval:** the duration in seconds, between consolidation, for the consolidator.
  
  - **kind:** an optional value of min, max, or average. By default it is average.
  
  - **tablelength:** an optional value for the length of the table, if **kind** has been specified. By default it is 1000.

### 2.6 Metric Collections Environment Variables

The following environment variables are available for a metric collection script, as well as for custom scripts, running from CMDaemon:

**On all devices:**

CMD\_HOSTNAME: name of the device. For example:

CMD\_HOSTNAME=myheadnode

**Only on non-node devices:**

CMD\_IP: IP address of the device. For example:

CMD\_IP=192.168.1.33

**Only on node devices:**

Because these devices generally have multiple interfaces, the single environment variable CMD\_IP is often not enough to express these. Multiple interfaces are therefore represented by these environment variables:

- CMD\_INTERFACES: list of names of the interfaces attached to the node. For example:

  CMD\_INTERFACES=eth0 eth1 ipmi0 BOOTIF

- CMD\_INTERFACE\_<interface>\_IP: IP address of the interface with the name \<interface\>. For example:

  CMD\_INTERFACE\_eth0\_IP=10.141.255.254
  CMD\_INTERFACE\_eth1\_IP=0.0.0.0

- CMD\_INTERFACE\_<interface>\_TYPE: type of interface with the name \<interface\>. For example:
CMD_INTERFACE_eth1_TYPE=NetworkPhysicalInterface
CMD_INTERFACE_ipmi0_TYPE=NetworkBmcInterface

Possible values are:
- NetworkBmcInterface
- NetworkPhysicalInterface
- NetworkVLANInterface
- NetworkAliasInterface
- NetworkBondInterface
- NetworkBridgeInterface
- NetworkTunnelInterface
- NetworkNetMapInterface

- CMD_BMCUSERNAME: username for the BMC device at this node (if available).
- CMD_BMCPASSWORD: password for the BMC device at this node (if available).

To parse the above information to get the BMC IP address of the node for which this script samples, one could use (in Perl):

```perl
my $ip;
my $interfaces = $ENV{'CMD_INTERFACES'};
foreach my $interface ( split( " ", $interfaces ) ) {
    if( $ENV{'CMD_INTERFACE_' . $interface . '_TYPE'} eq "NetworkBmcInterface" ) {
        $ip = $ENV{'CMD_INTERFACE_' . $interface . '_IP'};
        last;
    }
}
# $ip holds the bmc ip
```

A list of environment variables available under the CMDae- mon environment can be found by running a script under CMDae- mon and exporting the environment variables to a file for viewing. For example, the `/cm/local/apps/cmd/scripts/healthchecks/ testhealthcheck` script can be modified and run to sample on the head node, with the added line: `set>/tmp/environment`. The resulting file `/tmp/environment` that is generated as part of the healthcheck run then includes the CMD_* environment variables.

**Example**

CMD_BMCPASSWORD
CMD_BMCUSERNAME
CMD_CLUSTERNAME
CMD.CmdSTARTEDTIME
CMD_DEVICE_TYPE
CMD_EXPORTS
CMD_FSEXPORT__SLASH_cm_SLASH_shared_AT_internalnet_ALLOWWRITE
CMD_FSEXPORT__SLASH_cm_SLASH_shared_AT_internalnet_HOSTS
CMD_FSEXPORT__SLASH_cm_SLASH_shared_AT_internalnet_PATH
CMD_FSEXPORT__SLASH_home_AT_internalnet_ALLOWWRITE

© Bright Computing, Inc.
2.7 Metric Collections Examples

Bright Cluster Manager has several scripts in the
/cm/local/apps/cmd/scripts/metrics directory. Among
them are the metric collections scripts testmetriccollection
and sample_responsiveness. A glance through them while reading
this chapter may be helpful.

2.8 Metric Collections On iDataPlex And Similar Units

IBM’s iDataPlex is a specially engineered dual node rack unit. When the
term iDataPlex is used in the following text in this section, it also implies
any other dual node units that show similar behavior.

This section gives details on configuring an iDataPlex if IPMI metrics
retrieval seems to skip most IPMI values from one of the nodes in the unit.

When carrying out metrics collections on an iDataPlex unit, Bright Cluster Manager should work without any issues. However, it may be that due to the special paired node design of an iDataPlex unit, most IPMI metrics of one member of the pair are undetectable by the `sample_ipmi` script sampling on that particular node. The missing IPMI metrics can instead be retrieved from the second member in the pair (along with the IPMI metrics of the second member).

The output may thus look something like:

**Example**

```
[root@master01 ~]# cmsh
[master01]$ device latestmetricdata node181 | grep Domain
Metric          Value
---------------------
Domain_A_FP_Temp  23
Domain_A_Temp1    39
Domain_A_Temp2    37
Domain_Avg_Power  140
Domain_B_FP_Temp  24
Domain_B_Temp1    40
Domain_B_Temp2    37
[master01]$ device latestmetricdata node182 | grep Domain
Metric          Value
---------------------
Domain_A_FP_Temp  no data
Domain_A_Temp1    no data
Domain_A_Temp2    no data
Domain_Avg_Power  170
Domain_B_FP_Temp  no data
Domain_B_Temp1    no data
Domain_B_Temp2    no data
```

Because there are usually many iDataplex units in the rack, the metrics retrieval response of each node pair in a unit should be checked for this behavior.

The issue can be dealt with by Bright Cluster Manager by modifying the configuration file for the `sample_ipmi` script in `/cm/local/apps/cmd/scripts/metrics/configfiles/sample_ipmi.conf`. Two parameters that can be configured there are `chassisContainsLeadNode` and `chassisContainsLeadNodeRegex`.

- Setting `chassisContainsLeadNode` to `on` forces the `sample_ipmi` script to treat the unit as an iDataPlex unit.

In particular:

- `auto` (recommended) means the unit is checked by the IPMI metric sample collection script for whether it behaves like an iDataPlex unit.
- `on` means the unit is treated as an iDataplex node pair, with one node being a lead node that has all the IPMI metrics.
off means the unit is treated as a non-iDataPlex node pair, with each node having normal behavior when retrieving IPMI metrics. This setting may need to be used in case the default value of auto ever falsely detects a node as part of an iDataPlex pair.

• The value of `chassisContainsLeadNodeRegex` can be set to a regular expression pattern that matches the system information pattern for the name, as obtained by CMDaemon for an iDataPlex unit (or similar clone unit). The pattern that it is matched against is the output of:

```bash
cmsh -c 'device ; sysinfo master | grep "^System Name"'
```

If the pattern matches, then the IPMI sample collection script assumes the unit behaves like an iDataPlex dual node pair. The missing IPMI data values are then looked for on the lead node.

The value of `chassisContainsLeadNodeRegex` is set to `iDataPlex` by default.