

FermiCloud Group, Fermilab

# Technical Report

for work during summer

Siyuan Ma  
8/4/2012



# Table of Contents

- I. Summary ..... 2
- II. Customize Gratia-Web ..... 3
  - 1. GraphTool and Cherryppy ..... 3
  - 2. Gratia-Web Source Structure..... 3
  - 3. Add a customized graph under Bar Graphs section..... 4
  - 4. Add a new database connection..... 5
- III. Reinforce Gratia-Web with interactive charts ..... 8
- IV. Feed Nagios with Gratia RSV Metrics ..... 9
  - 1. Install RSV..... 10
  - 2. Install Nagios..... 10
  - 3. Configure RSV Host for Nagios..... 10
  - 4. Configure Nagios Host to receive RSV Records ..... 11
- V. Automated Software Deployment on FermiCloud ..... 13
  - 1. onecluster ..... 13
  - 2. one\${Type}Cluster..... 14
  - 3. Utilities ..... 14
  - 4. Future Work..... 15
- VI. Acknowledgement ..... 15

## I. Summary

My work during summer mainly involves three projects: Gratia-Web, Nagios, and OpenNebula. The latter is the framework for FermiCloud; Gratia-Web is an OSG(Open Science Grid) project for system monitoring and metrics visualization; Nagios is another widely used monitoring system that gains a considerable popularity in industry. My efforts target at improving and integrating these three systems to simplify the management and use of FermiCloud. These works can be further categorized into four parts:

- 1. Customize Gratia-Web for the use cases of FermiCloud
- 2. Integrate interactive charting into Gratia-Web
- 3. Connect Gratia and Nagios (both latest version)
- 4. Automated software deployment on FermiCloud

The rest of this report details the content of each work.

## II. Customize Gratia-Web

[gratia-web](#) is a package developed upon [GraphTool](#) to visualize database for monitoring in [OSG](#) projects. By modifying xml configure files, [gratia-web](#) is able to

- Connect to specific databases
- Choose figure type (bar, stackedbar, ...) which is supported by [matplotlib](#)
- Write SQL for each figure

Below a short introduction on [GraphTool](#) and [cherryypy](#) is given first. Then I brief the source code structure and installation for [gratia-web](#). Two customization are elaborated at the end of this post.

### 1. GraphTool and Cherryypy

To some extent, [gratia-web](#) is a customized [GraphTool](#), which implements all the core functions used in [gratia-web](#). These functions include

- A XML-based configuration system for ORM(object relational mapping) , dynamic object, and SQL query
- wrappers of [matplotlib](#) functions to generate various graphs
- A embedded webserver using [cherryypy](#) that presents UI to take SQL arguments from browser and display the corresponding figure

As a light weight python web framework, [cherryypy](#) contains an out-of-box [WSGI](#) webserver which provides 1. tree-structure url – object method mapping 2. flexible plugin system to extend its own capability 3. a dictionary-based configuration system to enable easy configuration sharing and modification.

Imaging all urls form a tree with / as the root, we can mount an object to the leaf node of the tree by

```
# My application
app = myApp()
# Mount the application so that CherryPy can serve it
cherryypy.tree.mount(app, '/myApp', os.path.join(self.conf_path, "app.conf"))
```

So if the **myApp** python class has a [exposed method callme](#), it can be accessed through the address

<http://ip:port/myApp/callme>

For configuration and plugin system, please read the [cherryypy tutorial](#).

### 2. Gratia-Web Source Structure

The table below gives a brief description for each subdirectory in the source code folder.

|            |  |
|------------|--|
| templates/ | Use <a href="#">Cheetah</a> as the template language to generate web pages |
|------------|--|

|                                  |   |
|----------------------------------|---|
| graphs/                          | Import <a href="#">GraphTool</a> package to produce figures           |
| database/                        | Import graphtool.database to connect and to query databases           |
| gip/, voms/                      | Related to virtual organizations                                      |
| services/                        | Define resources like Compute Element, Storage Element and Subcluster |
| common/,<br>passct/,<br>summary/ | Related to probes   |
| web/, tools/                     | Related to the webserver, and objects mapped from url                 |

As introduced in previous section, the cherrypy webserver maps an url to a python class method. In gratia-web, there are two types of such python methods.

**Static Method** is hard-coded in python. One example is <http://ip:port/gratia/d0>, which is mapped to d0() method [exposed](#) in web/\_\_init\_\_.py.

**Dynamic Method** The strength of gratia-web (or GraphTool) lies in its XML-based configuration system. And one important capability of this system is to convert xml configuration to python objects on the fly.

One example is **query\_xml** object. It is first mounted to gratia/xml in config/website.xml. The object itself is defined in config/test\_queries.xml as a XmlGenerator class. The **query\_xml** object in turn contains a set of query objects (e.g. **GratiaBarQueries**). Therefore, All the urls under <http://ip:port/gratia/xml> are mapped to dynamic methods.

### 3. Add a customized graph under Bar Graphs section

Now we will try to add a new query **osg\_sharing\_vos\_customized** in page <http://ip:port/gratia/xml>. **Insert** below code snippet into config/gratia\_bar\_queries.xml

```
<query name="osg_sharing_vos_customized" base="GratiaGenericQuery.master_summary">
  <inputs>
    <input name="span" type="int" kind="sql">86400</input>
    <input name="starttime" partial="down" type="datetime" kind="sql">2012-05-22
00:00:00</input>
    <input name="endtime" partial="up" type="datetime" kind="sql">2012-06-04 23:59:59</input>
    <input name="includeFailed" kind="sql"> true </input>
    <input name="includeSuccess" kind="sql"> true </input>
    <input name="facility" kind="sql"> .* </input>
    <input name="probe" kind="sql"> .* </input>
```

```

    <input name="user" kind="sql"> .* </input>
    <input name="vo" kind="sql"> .* </input>
    <input name="role" kind="sql"> .* </input>
    <input name="exclude-role" kind="sql"> NONE </input>
    <input name="exclude-vo" kind="sql"> unknown|other </input>
    <input name="exclude-user" kind="sql"> NONE </input>
    <input name="exclude-facility" kind="sql"> NONE|Generic|Obsolete </input>
    <input name="resource-type" kind="sql"> ONEVM.* </input>
  </inputs>
</sql>
  <filler name="group"> VO.VOName, S.SiteName </filler>
  <filler name="column"> sum(Cores*WallDuration)/3600 </filler>
  <filler name="where"> AND WallDuration > 0 </filler>
</sql>
<attribute name="pivot_name">Usage Type</attribute>
<attribute name="title">Opportunistic Wall Hours by VO (Customized)</attribute>
<attribute name="graph_type">GratiaStackedBar</attribute>
<results module="gratia.database.query_handler" function="opportunistic_usage_parser5">
  <inputs>
    <input name="grouping_transform">make_int</input>
    <input name="pivots"> 0,1 </input>
    <input name="grouping"> 2 </input>
    <input name="results"> 3 </input>
  </inputs>
</results>
</query>

```

The **name** attribute of **query** element will appear as part of the url for the new entry, which is `gratia/xml/$name`. The **base** attribute indicates where the query template is defined. GraphTool will merge the content in our query element and what in the query template. We can also customize the SQL query within the **input** element, which equals the [query string](#) generated by filling the query form on webpage. The **title** attribute element will appear as the new entry's name in `gratia/xml`. More tag and attribute usage can be found in [GraphTool Advanced Tutorial](#).

After editing the xml, *reboot the gratia-web server*

```
sudo /etc/init.d/GratiaWeb restart
```

The new entry `osg_sharing_vos_customized` will appear after refreshing the web page.

## 4. Add a new database connection

1. Add the new database connection in `/etc/DBParam.xml` under the `gratia ConnectionManager`

```

<connection name="gratia-psacct">
  <attribute name="Interface"> MySQL </attribute>
  <attribute name="Database"> gratia_psacct </attribute>
  <attribute name="Host"> gr-fnal-mysql-collector.fnal.gov </attribute>
  <attribute name="Port">0000</attribute>
  <attribute name="AuthDBUsername"> zzzzz </attribute>

```

```
<attribute name="AuthDBPassword"> zzzzz </attribute>
</connection>
```

2. Since database connections are attached to **SqlQueries** objects. We can create a new **SqlQueries** object in `config/gratia_customized_queries.xml`

```
<graphtool-config>

  <import module="gratia.config" data_file="generic_queries.xml" />

  <class type="SqlQueries" name="GratiaCustomizedQueries">

    <attribute name="display_name"> Customized Graphs </attribute>
    <attribute name="connection_manager"> GratiaConnMan </attribute>
    <aggregate>
      <connection> gratia-psacct </connection>
    </aggregate>

    <query name="osg_sharing_vos_customized_psacct" base="GratiaGenericQuery.master_summary">
      <inputs>
        <input name="span" type="int" kind="sql">86400</input>
        <input name="starttime" partial="down" type="datetime" kind="sql">2012-05-22
00:00:00</input>
        <input name="endtime" partial="up" type="datetime" kind="sql">2012-06-04
23:59:59</input>
        <input name="includeFailed" kind="sql"> true </input>
        <input name="includeSuccess" kind="sql"> true </input>
        <input name="facility" kind="sql"> FermiCloud|OpenNebula </input>
        <input name="probe" kind="sql"> .* </input>
        <input name="user" kind="sql"> .* </input>
        <input name="vo" kind="sql"> .* </input>
        <input name="role" kind="sql"> .* </input>
        <input name="exclude-role" kind="sql"> NONE </input>
        <input name="exclude-vo" kind="sql"> unknown|other </input>
        <input name="exclude-user" kind="sql"> NONE </input>
        <input name="exclude-facility" kind="sql"> NONE|Generic|Obsolete </input>
        <input name="resource-type" kind="sql"> RawCPU </input>
      </inputs>
      <sql>
        <filler name="group"> VO.VOName, S.SiteName </filler>
        <filler name="column"> sum(Cores*WallDuration)/3600 </filler>
        <filler name="where"> AND WallDuration > 0 </filler>
      </sql>
      <attribute name="pivot_name">Usage Type</attribute>
      <attribute name="title">Opportunistic Wall Hours by VO (Customized)</attribute>
      <attribute name="graph_type">GratiaStackedBar</attribute>
      <results module="gratia.database.query_handler" function="opportunistic_usage_parser5">
        <inputs>
          <input name="grouping_transform">make_int</input>
          <input name="pivots"> 0,1 </input>
          <input name="grouping"> 2 </input>
          <input name="results"> 3 </input>
        </inputs>
      </results>
    </query>
  </class>
</graphtool-config>
```

```

    </query>

</class>

</graphtool-config>

```

The new connection is specified in the **connection** element.

### 3. Edit config/text\_queries.xml to display this object as a new section in /grati/xml

```

.....
<import module="gratia.config" data_file="gratia_customized_queries.xml" />
.....

<class type="XmlGenerator" name="query_xml">
  <attribute name="timeout">900</attribute>
  <queryobj> GratiaStatusQueries </queryobj>
  <queryobj> GratiaEventsQueries </queryobj>
  <queryobj> GratiaDataQueries </queryobj>
  <queryobj> GratiaPieQueries </queryobj>
  <queryobj> GratiaGlideinBarQueries </queryobj>
  <queryobj> GratiaCustomizedQueries </queryobj>
  <queryobj> GratiaBarQueries </queryobj>
  <queryobj> GratiaRTQueries </queryobj>
  <queryobj> GratiaTransferQueries </queryobj>
  <queryobj> GratiaCumulativeQueries </queryobj>
  <queryobj> GridScanQueries </queryobj>
  <queryobj> GIPQueries </queryobj>
  <queryobj> RSVQueries </queryobj>
  <queryobj> RSVWLCGQueries </queryobj>
  <queryobj> RSVSummaryQueries </queryobj>
</class>

```

### 4. Edit config/gratia\_graphs.xml to associate the SqlQueries object with a grapher

```

.....
<import module="gratia.config" data_file="gratia_customized_queries.xml" />
.....
<class type="Grapher" name="gratia_customized_grapher">
  <attribute name="display_name"> Customized Graphs </attribute>
  <queryobj> GratiaCustomizedQueries </queryobj>
</class>

```

### 5. Mount the new grapher in config/website.xml

```

<class name="web" type="WebHost">
  <mount location="/gratia/bar_graphs" content="image/png"> <instance name="gratia_bar_grapher" />
</mount>
  <mount location="/gratia/customized_graphs" content="image/png"> <instance
name="gratia_customized_grapher" /> </mount>
  .....
  <instance name="static" location="/gratia/static" />

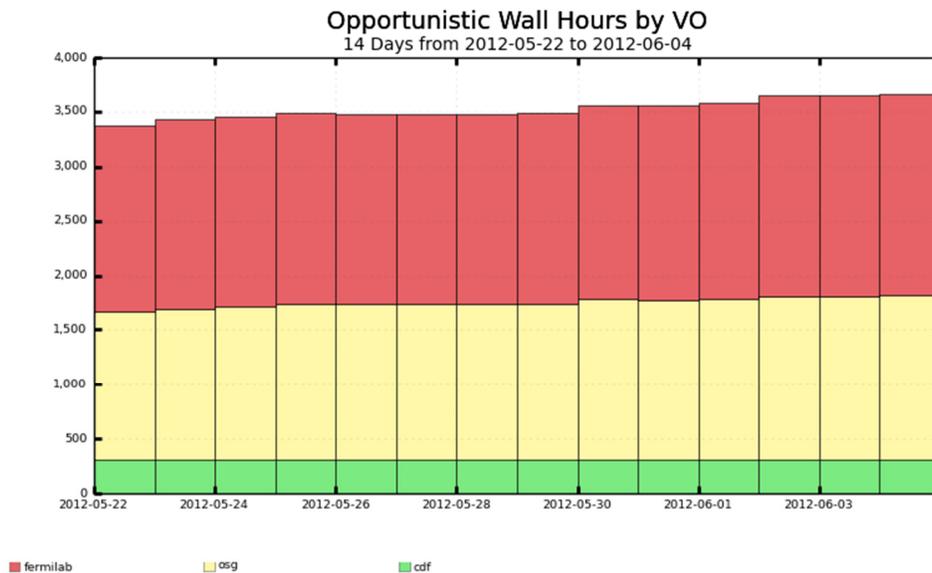
```

```
<config module="gratia.config">prod.conf</config>
<instance name="GratiaWeb" location="/gratia"/>
</class>
```

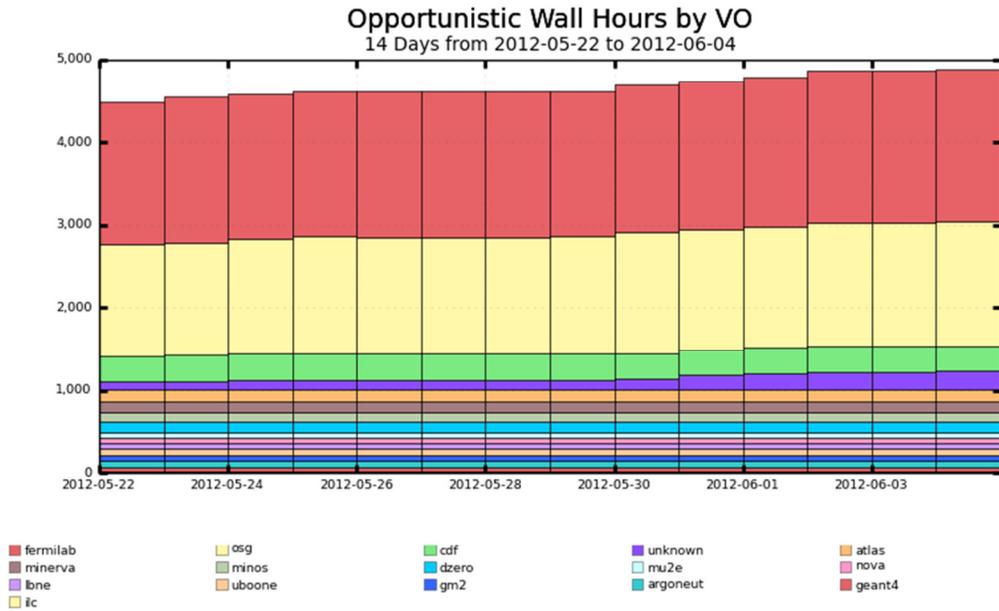
6. At last, reboot the server and refresh the /gratia/xml page.

### III. Reinforce Gratia-Web with interactive charts

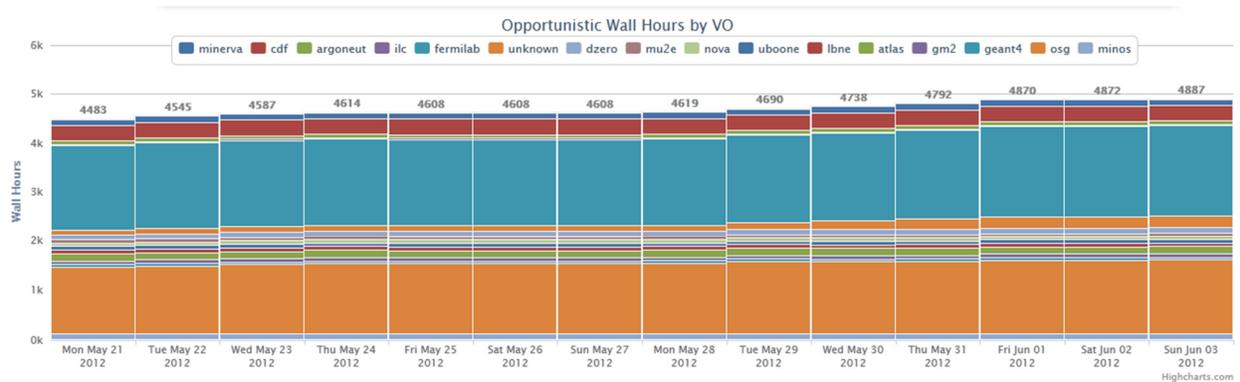
As mentioned before, Gratia-Web connects to database for site info and visualize fetched metrics with matplotlib. Figure below gives a typical result of database query generated by Gratia-Web.



Such static graphs generated by Gratia-Web have two major drawbacks. One is the shortage of features. Basic features like Tooltip still require the support of javascript and passing all the related data to the client side. Another issue emerges when data set becomes complicated. As the purpose of graphing is to make data intuitive, complex figure should be avoided. While in our case, the complicated data sets do yield complex figures, which limit the intuition we desire to build.



For the reasons above, we integrate Highcharts, a javascript lib, into Gratia-Web, to visualize data in a more interactive way. In addition to Tootip, Highcharts provide many fancy features including clickable legends and zoomability. It also alleviate the burden on server and network since data is now processed in the client browser.



#### IV. Feed Nagios with Gratia RSV Metrics

Nagios and RSV(Resource and Service Validation) are two powerful open source monitoring systems nowadays.

[RSV](#) is developed by the [VDI](#)(Virtual Data Toolkit) team, which is mainly committed to the simple deployment, maintenance, and usage of [OSG](#)(Open Source Grid) software. Although the latest [OSG](#) is now using RPM for distribution, [RSV](#) still serves as the recommended monitoring infrastructure for any OSG site admin.

Unlike that RSV finds favors mostly from [OSG](#), [Nagios](#) gains more popularity among enterprises as [a commercial product and full-fledged monitoring solution](#). It therefore has a larger [community](#) and more [company users](#) including Yahoo!, McAfee, and DHL.

In the rest of this post, we brief how to push [RSV](#) probe's result to a remote [Nagios](#) monitor. The installation and configuration are well documented. While the [document](#) for connecting these two systems is somewhat outdated. So this post is largely an effort to update this part of knowledge. Notice that we adopt [RSV](#) 3.7 and [Nagios](#) 3.4.1.

## 1. Install RSV

It is easy to [install RSV from RPM](#).

## 2. Install Nagios

Let's follow the [tutorial for Fedora](#). Since the tutorial starts with tarball, It works for most Linux Distribution. Notice that [Nagios](#) can be installed on a host different from the host for RSV. In addition, the [Nagios](#) user **nagiosadmin** created in this step will be referenced soon.

## 3. Configure RSV Host for Nagios

First make sure that [RSV sends records to Nagios](#). A consumer process, **nagios-consumer**, is launched for this routine. There are several things worth to mention:

- There is no need to add “—send-nsc”. And this component will not come with the default setting, and adding this option would cause some silent errors.
- Your **rsv-nagios.conf** will look like

```
[RSV]
## short_hostname known to Nagios server; usually the short RSV monitoring hostname

## Your Nagios server coordinates
NAGIOS_URL: http://*****/nagios/cgi-bin/cmd.cgi
NAGIOS_USERNAME: nagiosadmin
NAGIOS_PASSWORD: *****
```

It implies that RSV will send the probe records to **cmd.cgi**. You will need to make sure the cgi is reachable on Nagios host. Also, **nagiosadmin** is the user created in last section.

- Check **nagios-consumer.output** and **nagios-consumer.err** under **/var/log/rsv/consumers** for any possible errors. If RSV is sending records to Nagios host, the **nagios-consumer.output** would look like:

```
2012-06-28 13:47:38: nagios-consumer initializing.
2012-06-28 13:47:38: Processing 440 files
2012-06-28 13:52:40: nagios-consumer initializing.
2012-06-28 13:52:40: Processing 40 files
2012-06-28 14:04:07: nagios-consumer initializing.
2012-06-28 14:04:07: Processing 95 files
2012-06-28 14:10:04: nagios-consumer initializing.
2012-06-28 14:10:04: Processing 48 files
```

## 4. Configure Nagios Host to receive RSV Records

[Nagios](#) uses a simple template language to [define](#) host, command, service, and many other [objects](#). An object appears as a collection of directives. An example for a host object is:

```
define host{

    use      generic-host      ; Inherit from a template

    host_name      remotehost

    alias      Some Remote Host

    address 192.168.1.50

    hostgroups      allhosts

}
```

Where **define** follows the object type, and each line within the **define** scope is a directive. There are two special directives:

**host\_name**: specify the ID of the host object. The object can be referenced by other objects using this ID. Also, the keyword in this ID declaration depends on the corresponding object types. For instance, **host\_name** for **host** object; **service\_name** for **service** object; **command\_name** for **command** object. Or in general, **\_\${OBJ\_TYPE}\_name** for **\_\${OBJ\_TYPE}\_** object.

**use**: it follows the name of a template. The object will then include all the directives within the template. Directives in the object itself having a higher priority, so they will overwrite the template's directives were the keyword the same (e.g. **address**, **hostgroups**).

```
define service {
    name rsv-probe      ; Template name
    use  generic-service ; Inherit from another template
    active_checks_enabled      0
    check_freshness            1
    register                    0
}
```

Above is an example of template in Nagios. A template is almost identical to other objects except:

- Its ID declaration uses only **name** as the keyword, no matter what is the **define** type.
- It has a special directive **register 0**, which informs the system not to register itself as an object

Objects and Templates are generally defined in **.cfg** files. If you add your own **.cfg** file, you will need to declare it in **nagios.cfg** before nagios launch.

With the above introduction, we can then start to configure Nagios for RSV:

- 1) Add [rsv.cfg and modify nagios.cfg](#). Probably need to remove the rsv member in the contact group, or create a rsv Nagios user on the Nagios Host.
- 2) Add `rsv_service.cfg`

On the RSV host, if `/etc/rsv` is:

```
consumers          if-gridftp-minerva.fnal.gov.conf  meta
consumers.conf    if-gridftp-minos.fnal.gov.conf    metrics
if-gridftp-argoneut.fnal.gov.conf  if-gridftp-muze.fnal.gov.conf     rsv.conf
if-gridftp-gm2.fnal.gov.conf       if-gridftp-nova.fnal.gov.conf     rsv-nagios.conf
if-gridftp-lbne.fnal.gov.conf      if-gridftp-uboone.fnal.gov.conf
```

The `conf` files `_${host}.conf` implies that `_${host}` is currently monitored by RSV. For example, `if-gridftp-argoneut.fnal.gov.conf` implies the RSV host is collecting info from site `if-gridftp-argoneut.fnal.gov`. And the `if-gridftp-argoneut.fnal.gov.conf` defines the info to collect

```
[if-gridftp-argoneut.fnal.gov]
org.osg.general.ping-host = 1
org.osg.globus.gridftp-simple = 1
```

Which is two metrics, `org.osg.general.ping-host` and `org.osg.globus.gridftp-simple`. By the configuration in last section, all the metrics in all the `conf` files will be sent to the specified Nagios system.

Therefore, we need to define corresponding services on Nagios host. To figure out what the definition looks like, we can first take a look at `/var/log/httpd/access_log` on Nagios host. If the RSV host is configured correctly, we will see the record below every several minutes:

```
127.0.0.1 - - [28/Jun/2012:14:31:03 -0500] "
GET / HTTP/1.0" 200 14 "-" "check_http/v1.4.15 (nagios-plugins 1.4.15)"
***** - - [28/Jun/2012:14:31:29 -0500] "
  GET /nagios/cgi-bin/cmd.cgi?
    cmd_typ=30&
    cmd_mod=2&
    service=org.osg.general.ping-host&
    host=if-gridftp-nova.fnal.gov&
    plugin_state=0&
    plugin_output=Host%2Bif-gridftp-
nova.fnal.gov%2Bis%2Balive%2Band%2Bresponding%2Bto%2Bpings.&
    btnSubmit=Committed HTTP/1.1"
401 491 "-" "Python-urllib/2.4"
```

In the record, we notice that `host` is the site appeared in `/etc/rsv`; and `service` is the metric found in `_${host}.conf`. This observation inspires us to write `rsv_services.cfg` as below.

```

define host {
    use                linux-server
    host_name          if-gridftp-nova.fnal.gov
    address             ***
}

define host {
    use                linux-server
    host_name          if-gridftp-argoneut.fnal.gov
    address             ***
}

define service{
    use                rsv-probe          ; Name of service template to use
    host_name          if-gridftp-argoneut.fnal.gov,if-gridftp-nova.fnal.gov
    service_description org.osg.general.ping-host
}

```

Note that the **service\_description** must be the name of the metrics, and **host\_name** need to match **`\${host}.conf** in `/etc/rsv`.

- 3) verify nagios.cfg and restart the service

## V. Automated Software Deployment on FermiCloud

The backbone of FermiCloud, OpenNebula 3.2, provides a sizable group of command line tools that ease the use and management of our cloud system. While we also notice the absence of large scale deployment tools in current OpenNebula distribution. We therefore propose a set of command line tools that could perform automated software deployment based on pre-cooked images. The target of these tools is to facilitate the management of virtual clusters, hence we name the entry command `onecluster`. There are three types of commands in our toolkit:

### 1. `onecluster`

The entry commnad of our toolkit performs general operations like

- launch and configure a virtual cluster of certain type
- shutdown a virtual cluster
- add or remove resources from a virtual cluster without breaking the state consistency of distributed software running upon (incomplete)
- monitor cluster state (incomplete)

In our design, `onecluster` is unaware of the detailed knowledge of software stack. Its only assumption is that the virtual cluster is a collection of virtual machines connected by (virtual) network, hence it could use general OpenNebula tools to perform vm operations or collect information. The usage convention of `onecluster` is:

```
onecluster <command> -t Type -n ClusterName -f Conf
```

Where

**command** is the operation we are going to take. Two operations are available for now, create and shutdown.

**Type** specifies the script we are going to use for image instantiation and cluster configuration. The corresponding script name is one\${Type}Cluster.

**ClusterName** is the unique ID for the entire cluster to store and fetch cluster states.

**Conf** is the configuration file taken by onecluster and one\${Type}Cluster. It defines both general configuration and cluster type specific configurations.

```
#[general configuration]
CONF_FUNC=Hadoop; #Configuration Function
CLUSTER_INFO_PATH=`dirname $0`;
USER=root;

#[type specific configuration]
TID=11; #TEMPLATE ID
HADOOP_HOME=/home/hadoop/etc/hadoop-1.0.3;
XPATH=/cloud/login/siyuan/exec/bin/orc-xonf.py;
HADOOP_TMP_DIR=/home/hadoop/storage;
SNUM=1;
```

Within the above configuration, **CONF\_FUNC** specifies function to execute in one\${Type}Cluster; **CLUSTER\_INFO\_PATH** gives path to store cluster information; **TID** is the template ID used to instantiate vm instances; **SNUM** represents the number of slave nodes to launch in a Hadoop cluster.

## 2. one\${Type}Cluster

For each type of cluster, configuration procedures vary from case to case, while they can generally be decomposed into four steps:

- Collect global information like IP addresses of VMs
- Configure each VM with the information collected
- Execute local configuration script on each VM
- Start software in accordance to the requirement of distributed software stack

For example, oneHadoopCluster would launch a Hadoop cluster in following steps:

- Collect all VM IP addresses
- Generate **masters** and **slaves** file for all VM instances by collected IP addresses; modify configuration files in each VM for current master node
- Format NameNode; set up storage space for each VM
- Start all hadoop processes

## 3. Utilities

Utilities are tools for special configuration purpose, like changing an xml configuration file. Since such modification has a strong dependency on the chosen software stack, such utility may not be designed for general purpose. During deployment, utility will be first copied to the local storage of each VM, and then get executed.

In the case above, XPATH specifies a utility to update Hadoop configuration files.

## 4. Future Work

One important concern is fault tolerance. As cloud scales up and software stack complexity increases, faults and errors will become inevitable during deployment. Admittedly, it is always a solution to delete all VM instances and to redo the deployment. The cost of such recovery is simply unaffordable when hundreds or thousands of instances involved. Therefore, we need a mechanism to

- Detect faults and errors immediately
- Perform small step undo when possible
- Perform small step redo when possible
- Guarantee **Final Consistency** (software stack is consistent at final stage)

Our primitive idea is to break each deployment into **READ, UPDATE, LOCAL-UPDATE** operations and cluster, local vm **STATES**.

- **READ** fetch information from all VM instances
- **UPDATE** push the processed information fetched by **READ** to all VMs
- **LOCAL-UPDATE** do not need any information from remote VMs. We distinguish UPDATE and LOCAL-UPDATE for their complexity and cost differences in regard to undo and redo.
- local vm **STATE** describes the state of one vm in the deployment procedure. It can be affected by UPDATE or LOCAL-UPDATE
- cluster **STATE** is the collection of all vm STATES. A cluster **STATE** is final consistent if it can still progress to the consistent final stage in current deployment.

To detect faults and errors, each definition of STATE needs to come with a check script; to perform undo, each UPDATE operation must accompany with a reverse operation; it is simple to redo a LOCAL-UPDATE.

## VI. Acknowledgement

Special thanks are given to Steven Timm, Tanya Levshina, Gabriele Garzoglio, Hyunwoo Kim, and all the other colleagues I have been working with in the fermilab. Your care and help are important to me, and also make fermilab a very pleasing place for work.