Grid Configuration Monitoring (GCM) Server and Database Documentation

Thomas Low*

March 10, 2009

Abstract

This paper contains a documentation about the *Grid Configuration Monitoring Server and Database* developed at CERN for the *Worldwide LHC Computer Grid* (WLCG). It describes all components in detail, so that it should be easy to maintain, enhance or extend the server and database.

Contents

*Grid Deployment, IT Department, CERN, Switzerland. thomas.low@cern.ch

1 Introduction

The aim of this work is to collect information about the configuration of the WLCG production grid infrastructure. It does this by providing a program that runs with a grid job that gathers information about a worker node and its environment. See the documentation of the *Worker Node Client*. [4]

The server is a python program which retrieves these information, so called *Tests*. It then stores the data in an oracle database using *Django*. Finally a web application [3] visualizes it for statistical purposes or to help to resolve problems. Figure 1 illustrates the whole layout.

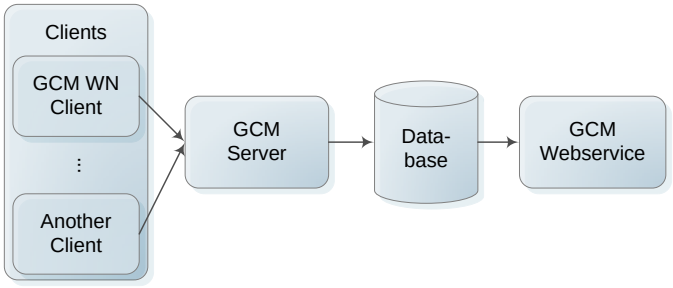

Figure 1: Layout of *Grid Configuration Monitoring* programs

The server is provided as a package which resides in the directory [`server/`] of the package `grid-gcm`.

The database models are stored in [`db/`] and [`tests/`].

2 The Server

The architecture and configuration of the server is identical to the one of the client. Therefore it is advised to read section 2 of the *Worker Node Client* documentation. [4]

The only differences are the used components as described below and the fact that the configuration file is named `grid-cm-server.conf` instead.

2.1 Components

This chapter describes all components which were developed for the server.

2.1.1 TestProducer

PyamqReceiver

The *PyamqReceiver* located in [`server/src/producer.py`] is the counterpart of the *PyamqSender* from the *Worker Node Client*. Instead of sending the *Tests* as messages it receives messages and converts them back into *Tests*.

Configuration The *PyamqReceiver* accepts the following arguments:

- **broker** (mandatory) is a list of connection strings separated by a space. Each connecting string is an *URI* like this:

  `protocoll://host:port/path?key=value&key=value`

  Supported protocolls of *Pyamq* are:

  - **stomp**
  - **http**

  Supported arguments are:

  - **timeout** defines the connection timeout in seconds.
  - **retries** defines how often the server is trying to establish a connection to the broker. (-1 means indefinite)

  An example looks like this:

  `stomp://gridmsg101:6163?timeout=3 http://prod-grid-msg/message`

- **destination** (mandatory) is the topic to which the server will subscribe. For example `/topic/grid.config.workernode`
- **random** (optional) states whether the next connection is established to a broker choosen randomly from the list of brokers.
- **pubkey** (optional) is a path to a public key file (suitable for OpenSSL) which will be used to decrypt *Tests* which have were encrypted by the client.
- **privkey** (optional) is the path to the corresponding private key file.

Note that the list of broker connecting strings is actually retrieved by a query to the *BDII* using the following two commands located at [`common/src/pyamq/ldapquery.py`]:

```
glite-sd-query -e -t msg.broker.stomp
glite-sd-query -e -t msg.broker.rest
```

They will automatically be merged with the ones defined in the configuration file which act as a fail-safe.

Example Consider the following configuration. The `grid-cm-server.conf` contains a section for the *PyamqReceiver* like so:

```
[gcm.server.producer.PyamqReceiver]
broker=stomp://gridmsg101:6163?retries=-1
random=0
destination=/topic/grid.config.workernode
pubkey=%(GCM_HOME)s/lib/gcm/certs/grid-cm-client-wn.pubkey
privkey=%(GCM_HOME)s/lib/gcm/certs/grid-cm-client-wn.privkey
```

Once started the server will run the *PyamqReceiver* which conntects to a broker and subscribes to the topic `/topic/grid.config.workernode`.

Now it is able to receive messages and convert them to *Tests*. After doing that it will forward each *Test* to all *TestConsumer*.

Finally the `cleanup()` method will wait for a keyboard interrupt (ctrl+c).

2.1.2 TestConsumer

WebAdmin

The *WebAdmin* was created only for debugging purposes. It is a simple web server which displays all *Tests* (which were produced by one execution of the *Worker Node Client*) in a row of a big table. This allows developers to see what is going on in a chronological order.

Each cell and row of the table has a specific color. A row can be grey, if everything during that execution of the *Worker Node Client* was fine, or light yellow when something went wrong. A cell can be grey or light yellow if the *Test* was successful or yellow if it failed. A cell is purple if the name of the *Test* does not appear in the attribute `testlist` of the special *Test* `wn_client` which was produced by the *ClientTester* of the *Worker Node Client*.

The status *successful* or *failed* will be determined by the detailsdata attribute of a *Test*. If it is empty, which means there is no logging information available, then it is considered to be run successful. Otherwise it is considered failed. The *WebAdmin* does not check whether all attributes of a particular *Test* are present.

The label of a cell can be "ok", "dat" (depending on whether detailsdata is empty or not) or the value of the attribute `summary` of a *Test*.

Configuration The *WebAdmin* accepts only one argument:

- **port** (optional) is the port which will be opened to serve web pages. (default 8162)

Example You can open the WebAdmin using:

```
http://your-host:8162/wncm/
```

JobIdSyncer

The *JobIdSyncer* was also created only for debugging purposes. It looks into a specific file which contains a list of job ids and delets the ids of jobs which already send messages to the server.

Therefore it uses the attribute `jobid` of the *Test* `wn_grid_job`.

Configuration The *JobIdSyncer* accepts only one argument:

- **jobfile** (mandatory) is the path to the file which contains the job ids of submitted jobs.

Example At first create a job which runs the *Worker Node Client*. This can be done like so:

1. Install the *Worker Node Client* into a temporary directory using:

   ```
   make install_client_wn DESTDIR=/tmp/gcm
   ```

2. Create a tar ball `grid-cm-client-wn.tar` from this directory. (including the root gcm)

   ```
   cd /tmp
   tar -cf grid-cm-client-wn.tar gcm
   ```

3. Write a shell script which untars and runs `bin/grid-cm-client-wn`. For example:

   ```
   tar -xf grid-cm-client-wn.tar
   gcm=‘pwd‘/gcm
   export GCM_HOME=$gcm
   $gcm/bin/grid-cm-client-wn -v
   rm -rf gcm
   ```

4. Create a JDL file which runs the shell script. For example:

   ```
   Executable = "grid-cm-client-wn.sh";
   InputSandbox = {"grid-cm-client-wn.sh","grid-cm-client-wn.tar"};
   StdOutput = "std.out";
   StdError = "std.err";
   OutputSandbox = {"std.out","std.err"};
   ```

Then submit a job using:

```
glite-wms-job-submit -a -o jobs grid-cm-client-wn.jdl
```

This will create a file `jobs` which contains the job ids. Now if the *JobIdSyncer* is configured to use this file, it will only contain jobs which could not send messages properly.

DatabaseInserter

The *DatabaseInserter* uses *Django* [1] to insert *Tests* into an oracle database.

Basically it looks in the repository of *Django* model classes for a class with the same name as the *Test*. If such a class exists it then calls a special method `insertTest` with the *Test* as the only attribute. If not overwritten this method tries to merge the attributes of the *Test* with the database fields defined in the model class.

A detailed description about this process will be discussed in the next section 3.

Configuration The *DatabaseInserter* accepts no arguments. Everything will be configured by *Djangos* settings file.

3 The Database

The database is mainly maintained by *Django*. It is advised to read *Djangos* tutorials [2] before going on.

3.1 Configuration

As usual *Django* will be configured by a `settings.py` located at [`db/etc/settings.py`]. It contains only settings which affect the database.

You can also customize the configuration by creating a `settings_local.py`. That way you do not have to specify sensible data in a file which is under version control.

For synchronizing the database with *Django* models there is a script called `grid-cm-db` at [`db/bin/grid-cm-db`] which acts exactly like *Djangos* `django-admin.py` except that you do not need to specify `DJANGO_SETTINGS_MODULE`.

For example to reset the database the following command can be used:

3.2 Model Layout

Both the server and the web portal uses *Djangos* model layer. That means all database tables and relationships are implemented as python classes.

3.2.1 Grid Topology Models

The grid topology models [`db/src/grid/models.py`] exist to match *Worker Nodes* to a certain *Site*, a *Site* to a certain *Region* and so on. It is mainly static data which will not change very often.

Nevertheless there is a program called `grid-cm-db-sam` [`db/bin/grid-cm-db-sam`] which is able to fill the database with intial data. It does not synchronize changes with the *Test* data. It just adds previously unknown *Sites*, *Regions* and so on. The former is left as an exercise for the reader.

3.2.2 Test Models

The *Test* models are located in [`tests/src`]. Each of them represent a *Test* within the database.

To respect the DRY principle (do not repeat yourself) there are a few abstract classes which do most of the work. A *Test* model class only needs to be inherited from the right class and specify fields which are special for this certain *Test*. Thanks to *Django* it is even possible to define other abstract classes which combine several common fields of some *Tests*. (e.g. *Tests* which collect the version of software)

TestModel

The most general abstract class is `TestModel` located at [`tests/src/models.py`]. It defines just one field: a timestamp which indicates the insertion time of the test. It is not necessary to provide a field for the name of the *Test* since each type of *Test* is represented by exactly one model altough there might be other tables linked to it. (e.g. the *Worker Node* table) It also defines the default insertion behaviour for *Tests*. See section 3.3 for more details.

To mark a model as a *Test* model and to add some meta information about the *Test*, the model class must contain a class called `TestMeta`. Since `TestModel` already adds this class, *Test* models do not need to specifiy it except they want to customize one of these attributes:

- **test_alias** is a list of aliases for a *Test* model. This is used when *Tests* were being renamed.
- **attr_alias** is a dictionary which maps attributes of *Tests* to model field names. This is used when attributes were renamed but not all *Worker Nodes* already have the new version of this test. See section 3.3 for details.
- **description** is a string containing a short description of what the *Test* is doing. The string will be displayed on the web portal.
- **sources** is a list of file paths to the source files of the *Test*. Again these files will be displayed on the web portal.

So if for example the name of a *Test* has changed for some reason, but their underlying attributes are the same, an alias for that *Test* would assure that both old and new results would be stored in the database:

```
class YourNewTestName(TestModel):

    # ... the fields which did not change

    class TestMeta:
        test_alias = ["YourOldTestName"]
```

WN_TestModel

The class `WN_TestModel` [`tests/src/wn/models.py`] is derived from `TestModel` an contains some more specific fields which are tied to the basic operation of the *Worker Node Client*:

- **testId** is an unique identifier for a *Test*.
- **testVersion** is the version of the *Test*.
- **wn** is a foreign key to the *Worker Node* table.
- **pyamqId** is an unique identifier for each execution of the *Worker Node Client*.
- **executionStarttime** is a datetime field which describes the start time of a *Test*.
- **executionEndtime** is a datetime field which describes the end time of a *Test*.
- **executionTime** is a float field which describes the duration of a *Test*.
- **detailsdata** is a text field which contains the log message of a *Test*.

Each *Worker Node Test* model class should be derived from `WN_TestModel`.

A short Example

Although the User Guide [**?**] gives a detailed description about how to add new *Tests*, here now a short example.

Consider the following situation. The *Worker Node Client* sends information about the processor of the machine it is running at. A *Test* might look like this:

```
testName:wn_system_cpu
ProcessorCount:1
ProccesorCores:2
ProcessorClockSpeed:2563.34
ProcessorVendor:GenuineIntel
executionStarttime:1234567890
executionEndtime:1234567891
executionTime:0.1234
...
```

In order to insert that data into the database there has to be a *Test* model class which defines what arguments are of interest. The model could look like this:

```
from django.db import models

class WN_System_CPU(WN_TestModel):
    ProcessorCount = models.IntegerField("Processors",null=1)
    ProcessorCores = models.IntegerField("Cores",null=1)
    ProcessorClockSpeed = models.FloatField("Speed",null=1)
    ProcessorVendor = models.CharField("Vendor",max_length=100,null=1)
```

And that is it. The abstract classes `WN_TestModel` and `TestModel` will automatically add all the other fields like `executionTime` and take care of the necessary actions to insert the data into the database. Of course it is possible to customize the default behaviour. See section 3.3 for more details.

Loading of Test Models

Usually *Django* requires all models to be in the main `models.py` of your *Django* project. To overcome this limitation the `models.py` [`tests/src/models.py`] dynamically loads other `models.py` files from arbitrary subdirectories into the same namespace. Thus models can be arranged in any directory hierarchy. Note that it is easy to produce naming conflicts when creating new `models.py` files. If a model requires a lot of code everything except the pure model should be sourced out into another module.

The same applies to *Djangos* admin interface except that `admin.py` files will be loaded from all subdirectories. If `admin.py` can not be found in the same directory where `models.py` is located, the server will automatically register a default admin interface for that particular model.

3.3 Inserting a new Test

As soon as a *Test* will be received by the server the *DatabaseInserter* (see section 2.1.2) takes care of it. It then looks for the right model which matches

the name of the *Test* and tries to merge the attributes of the *Test* with the fields of the model class. Finally it tells *Django* to insert a new record into the table of the model.

Looking for the right Model class

In order to find the right model the *DatabaseInserter* iterates over all *Test* models (which are derived from the class `TestModel`) and tries to find one which has the same name as the *Test* or which has the name of *Test* listed in its `test_alias` attribute of its `TestMeta` class.

Since all models are in one namespace (`gcm.tests.models`) there is no special mechanism necessary rather than looping over `dir(gcm.tests.models)`.

The first model class which matches those criterias will be used to insert the *Test*. Therefore the *DatabaseInserter* creates an instance of it and calls its `insertTest` method together with the *Test*.

Merging and Inserting

The default behaviour of the `insertTest` method is to merge attributes of the *Test* with fields of the model and to call its `save()` method afterwards. Therefore it consists of three steps which are implemented as separate python methods within the model:

1. **_preInsertTest** does nothing by default. This is the entry-point which can be used to preprocess the data.

2. **_doInsertTest** merges the attributes of a *Test* with the fields of a model. Therefore it iterates over all *Test* attributes and tries to find a field with the same name. It also checks whether an alias was defined in the `attr_alias` dictionary of the `TestMeta` class for that particular attribute. The first field which matches those criterias will be used for that attribute. Finally it checks the type of the field (integer, char, datetime, ...), converts the value of the *Test* attribute into the right type and assigns the value to the field.

3. **_postInsertTest** tells *Django* to insert a new record by calling the models `save()` method. In order to make things a little faster it does not insert each *Test* one by one. Instead it stores a number of model instances and inserts all of them at once when a limit is reached. Note that this bulk insertion is not implemented on an SQL level but just as a grouping of `save()` calls.

Customizing

There two ways to customize how a *Test* will be inserted into the database.

- By overriding `_preInsertTest` it is possible to preprocess the data which then will be inserted by `_doInsertTest`. Of course it is also possible to query other models or insert new records into them.
- When overriding `_insertTest` there are no limitations.

An Example In order to update the grid topology tables the abstract class `WN_TestModel` overrides the `_preInsertTest` method like so:

```
def _preInsertTest(self, test):
    """Override the default insert behaviour for worker node tests."""
    TestModel._preInsertTest(self, test)
    wn, _ = WN.objects.get_or_create(fp=test.hostid, name=test.hostname)
```

This will insert a new *Worker Node* if it is not already stored in the database and load its representation into `wn`. Note that although in this case it would not be necessary to call the `_preInsertTest` method of `TestModel` it is advised to do so.

Caching

To speed up the customized insertion code as many as possible caches should be used. This is very important because otherwise e.g. a query would be executed everytime a *Test* will be inserted.

Django provides a number of different cache backends. The most simple one is `django.core.cache.backends.lockmem` which uses a dictionary. For an example on how to use it see [`tests/src/wn/grid/infra/models.py`].

4 References

[1] Django. Django - the web framework for perfectionists with deadlines. http://www.djangoproject.com/.

[2] Django. Djangos tutorials. http://docs.djangoproject.com/en/dev/intro/.

[3] T. Low. Grid configuration monitoring - web portal documentation. See https://twiki.cern.ch/twiki/bin/view/EGEE/WorkerNodeConfiguration#Documentation.

[4] T. Low. Grid configuration monitoring - worker node client documentation. See https://twiki.cern.ch/twiki/bin/view/EGEE/WorkerNodeConfiguration#Documentation.