Grid Configuration Monitoring (GCM) Worker Node Client Documentation

Thomas Low[*]

March 10, 2009

Abstract

This paper contains a documentation about the *Grid Configuration Monitoring Worker Node Client* 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 client.

Contents

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

1 Introduction

Aim

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.

It shall collect information about:

- Versions of **Installed Software** like *Java, Python, Perl, gLite*,
- **Operating System** like its name, version or architecture,
- **The Grid Job** like its id, submitter and his virtual organization,
- **Infrastructure** like the site, computing element or queue to which a worker node belongs.

Due to the heterogeneous nature of grids the program has to work on a wide range of systems. It also must not disturb the regular grid operation. This implies that it should be platform independent, fast and reliable.

Additionally it should provide a mechanism which allows people to easily access information which were not yet gathered by the system without waiting for the deployment of the next release.

This problem has been tackled previously inside WLCG, but the solutions did not result in a system which was able to handle these tasks in production. Our solution tries to overcome the problems of its predecessors by using modern techniques like *Python* [6] or *ActiveMQ* [1].

Approach

The *Worker Node Client* is provided as a package which is loaded onto worker nodes. It resides in the directory [`clients/wn/`] of the package `grid-gcm`. It is a python program which gathers information, so called *Tests*, by running a set of scripts or executables within a certain amout of time. After that it sends all results to a server using *ActiveMQ* and exits.

The output of a *Test* is collected by a server located at CERN so they can be visualized with a web interface for statistical purposes or to help to resolve problems. Figure 1 illustrates the whole layout.

General Definitions

Test

A *Test* [`common/src/test.py`] is a list of key-value pairs, which are nothing but simple strings as described in the *Grid Monitoring Probes Specification*

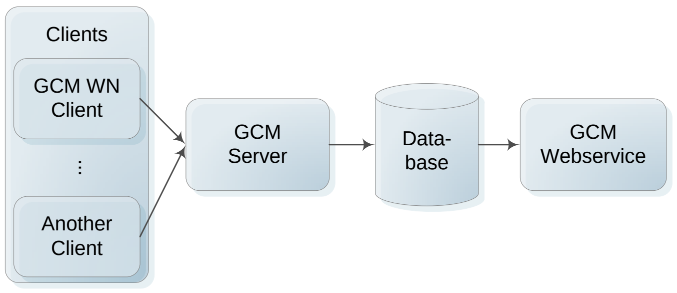

Figure 1: Layout of *Grid Configuration Monitoring* programs

[2]. Additionally a *Test* has to have a `testName` key which should be an unique identifier for each kind of a *Test*. Note that a *Test* name must only consist of alphanumeric characters and underscores. The attribute `detailsdata` is predetermined for error messages and can contain a multi line value.

A typical string representation of a *Test* looks like this:

```
testName:wn_system_os
testVersion:1.0
executionCommand:system_os
executionStarttime:1234567
executionEndtime:1234568
summary:error
detailsdata:Operating System could not be gathered.
Error 1234. Timeout exceeded.
```

Note that altough a *Test* does not need to have certain tuples, there are a couple of common key-value pairs which will be created at different points in the life time of a *Test*. Some of these are:

- **testId** is an unique id for each test. (e.g. 7dd4984b03115b34f7993439055f622f)
- **testVersion** is the version of the test. (e.g. 1.0)
- **hostname** is the name of the host. (e.g. lxbra2108.cern.ch)
- **hostid** is a unique id for each host. (e.g. d965877f592d3fdf85d43a28e9dc8380)
- **pyamqId** is a unique id for tests which were being sent by *Pyamq* at the same time. See section **??** for details. (e.g. 30ff8d0689a5b23b0024bcf9746ceeb7)
- **executionCommand** is a string describing what was being executed to get the *Test*. (e.g. /path/to/executable arg1 arg2)
- **executionStarttime**, **executionEndtime** are unix timestamps which state when a *Test* was started and finished. (e.g. 1234567890.123)

- **executionTime** is the difference between the start and end time. (e.g. 0.012345)

2 The Client

2.1 Operation

The *Worker Node Client* [`clients/wn/src/client.py`] is shaped by a basic scheme. It consists of two parts: *Test* producers and consumers. Each *TestProducer* does something which creates one or more *Tests*. Meanwhile each *TestConsumer* retrieves *Tests* one by one and processes them somehow. After all *TestProducer* are finished and each *Test* was processed by all *TestConsumer* the program stops.

As you can see in Figure 2, the common coordinator is a simple event manager called *TestManager*, which calls each *TestConsumer* with a specific *Test* which was given to it by a *TestProducer*.

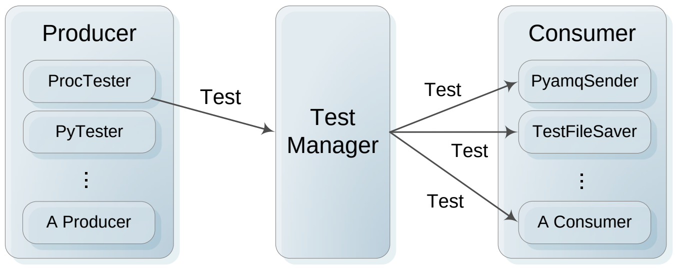

Figure 2: Basic layout of the *Worker Node Client*. Each *Test* of a *TestProducer* will be sent to all *TestConsumer*.

This should happen asynchroniously so that input/output intensive components (a *TestProducer* or *TestConsumer*) do not disturb processor intensive ones. That means each component has to be thread-safe.

The execution of a component consists of 3 phases: setup, run and cleanup. These phases are executed in a specific order to ensure that none of the *Tests* get lost.

Setup

At first all *TestConsumer* and after that all *TestProducer* will be configured by executing their `setup()` function. This means they are now ready to retrieve and create *Tests*.

Run

Then each *TestProducer* will be run one by one using their `run()` function. If a *TestProducer* requires a lot of time to complete, it should start a new thread, so it will not slow down the scheduling of other tests. From now on a *TestProducer* can produce *Tests* which will be send to all *TestConsumer*.

Cleanup

Finally all *TestProducer* and after that all *TestConsumer* will be cleaned up in the reverse order as they were being set up using their `cleanup()` function. As soon as all *TestProducer* were cleaned up, no new *Test* will be created.

After the cleanup is complete the program exits.

Figure 3 demonstrates the whole workflow with a simple example consisting of 3 producers and 3 consumers.

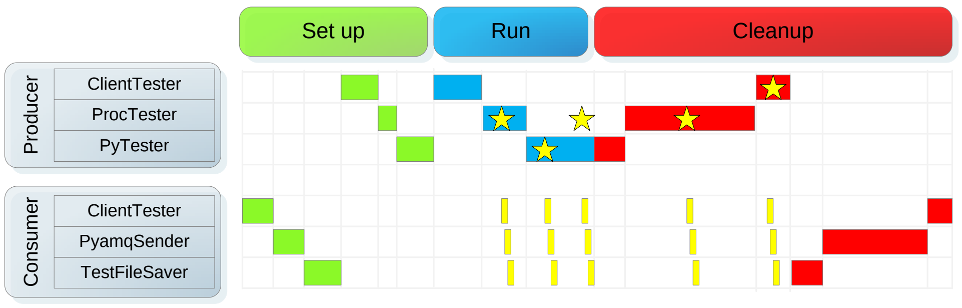

Figure 3: Visualizes the 3 phases and the order in which a *TestProducer* or *TestConsumer* becomes active. It also illustrates how a *Test* (a star) will be duplicated and handled by each *TestConsumer*.

You can even write classes which are both *TestConsumer* and *TestProducer*. They might also share information. See *ClientTester* in section 3.2.3 for an example.

That means the specific behaviour of the *Worker Node Client* is defined by the behaviour of each *TestConsumer* and *TestProducer*. Thus it is important to configure the client correctly.

2.2 Configuration

Langauge

The whole configuration is done with a single configuration file [`clients/wn/etc/grid-cm-client-wn.conf`]. The language is very similar to *Microsoft Windows INI* files. You can also use references to other key-value pairs. For more details see the *Python* documentation on *ConfigParser* [4]. Additionally you can use an environment variable reference if there is no key with the same name in the current or default section using almost the same syntax:

```
%(ENVIRONMENT_VARIABLE)s
%(ENVIRONMENT_VARIABLE:-DEFAULT_VALUE)s
```

Since the environment variable *GCM_HOME* is always set at the beginning of each execution, you can easily access the installation directory of the client like so:

```
[a_section]
  akey=%(GCM_HOME)s/subdirectory/file.txt
  another=%(USER:-root)s
```

Structure

The *Worker Node Client* configuration file has a quite dynamic structure. The only mandatory section is called *client* which contains several entries:

- **active** is used by the *JobWrapper* (see section 4) to determine whether the client should be executed or not.
- **timeout** is an overall timeout in seconds for the whole *Worker Node Client*. As soon as the timeout exceeds, the program will be shut down immediately.
- **consumer** is a list of references to python classes which are derived from the *TestConsumer* class. They will be executed as described in section 2.1 in the same order as listed here.
- **producer** is a list of references to python classes which are derived from the *TestProducer* class. They will be executed as described in section 2.1 in the same order as listed here.
- **interval** (optional) is the number of seconds which need to pass by until the client will execute again. Otherwise it will just do nothing and quit. See section 3.2.3 to read more about the clients memory.
- **memoryFile** (optional) is the path to the memory file of the client. See section 3.2.3 for details.

For each class in **consumer** and **producer** there has to be a section with the name as in the list above. These sections depend on the specfic requirements of the particular component. Each key-value pair will be sent to the `__init__` function of the class as keyword-arguments.

For debugging purposes there is also a section *logger* which uses the standard python *logging* module [5].

Loading Configurations

A configuration will be loaded from 4 location. Each subsequent file overrides options and sections which has the same name in a preceding file:

1. $GCM_HOME/etc/grid-cm-client-wn.conf
2. $HOME/.grid-cm-client-wn.conf
3. $GCM_CLIENTS_WN_CONFIG
4. File passed to the **config** argument of the command line interface

Example

If you want to add a *TestConsumer* `myConsumer.TestPrinter`. You can specify your consumer in the list of consumers like so:

```
[client]
...
consumer=aConsumer.Consumer myConsumer.TestPrinter
...
```

Then you create a section with the same name, which contains some arguments:

```
[myConsumer.TestPrinter]
stream=stdout
color=blue
```

These arguments will be sent to the `__init__` function of the `TestPrinter` class. So your class definition can look like this:

```
class TestPrinter(TestConsumer):
    """This class prints out all tests it receives."""

    def __init__(self, stream, color="white"):
        ...
```

Since the `color`-argument has a default value it need not to appear in the configuration file.

3 Components

This chapter describes all components (*TestProducer* or *TestConsumer*) which were developed for the *Worker Node Client*.

3.1 TestProducer

A *TestProducer* [common/src/testproducer.py] is a class which produces one or more *Test* class instances. To support the 3 phases it has functions `setup(self)`, `run(self, event)` and `cleanup(self)` whereas `event` is the function which has to be called with a *Test* to send it to all *TestConsumer*.

3.1.1 ProcTester

The *ProcTester* [`clients/wn/src/producer/proctester.py`] is a *TestProducer* which behaves like a process manager. It executes processes, captures their output and converts them to a *Test*.

At first *ProcTester* looks for `*.testdef` files located in a specific directory. These files are formatted as specified in the *Grid Monitoring Probes Specification* [2] and contain several key-value pairs like `testName`, `testVersion` or `executionFile`. They might even contain pairs which will be used by another *TestConsumer* later on.

The important keys are `executionFile` and `executionArguments`. These two values are used to construct a system command which will be executed in a subprocess of the *Worker Node Client*. The output of this subprocess will be captured and converted to key-value pairs. Note that it is not possible to execute programs which are located outside of the specified directory.

The final *Test* which will be forwarded to the *TestConsumer* is the combination of all key-value pairs from the `*.testdef` file and the output of the command.

Configuration

The *ProcTester* accepts the following arguments:

- **directories** (mandatory) specifies the directories which will be scanned for `*.testdef` files separated by colon.
- **timeout** (`None`) is timeout in seconds. As soon as the timeout exceeds all pending subprocesses will be killed immediately.
- **threads** (`3`) determines how many programs can be executed simultaniously. Use 0 for indefinite.
- **memoryFile** (optional) is the path to the memory file of the client. See section 3.2.3 for more details.

Each `*.testdef` file can contain amongst others the following arguments:

- **testName** (mandatory) defines the name of the *Test*.

- **executionFile** (optional) identifies the file which will be executed by the *Worker Node Client*. If this argument is not defined, it uses the name of the `*.testdef` file and tries to execute `*`.
- **executionArguments** (optional) is a string of arguments which will be added to the command which executes `executionFile`.
- **executionProbability** (optional) determines the probability, whether an `executionFile` will be executed. Its value is a floating-point number between 0.0 and 1.0 (default 1.0).
- **executionInterval** (optional) is a number of seconds which need to pass by until a particular test will be executed again. Note that if the memory of the client is available on a worker node, the **execution-Probability** argument will be ignored. See section 3.2.3 for more information about the memory mechanism.

Additionally you can create the file `testdef.default` in a certain directory. Its tuples will be added to each `*.testdef` file in that directory as default values.

Example

Consider the following configuration. The `grid-cm-client-wn.conf` contains a section for the *ProcTester* like so:

```
[gcm.clients.wn.producer.proctester.ProcTester]
directories=%(GCM_HOME)s/lib/gcm/tests/wn
timeout=25
threads=3
```

The directory `%(GCM_HOME)s/lib/gcm/tests/wn` contains a file `system_os.testdef` which looks like this:

```
testName:wn_system_os
testVersion:1.1
executionProbability:0.7
```

When the *Worker Node Client* starts, it sets up the *ProcTester* which will scan the directory for `*.testdef` files, find `system_os.testdef` and load it.

Since there is no `executionFile` key-value pair in the `system_os.testdef` it will guess the name and execute a file `system_os` with a probability of 70%. Assuming it will be executed, it then captures the programs output:

```
GlueHostArchitecturePlatformType:x86_64
GlueHostOperatingSystemName:ScientificCERNSLC
GlueHostOperatingSystemRelease:4.7
GlueHostOperatingSystemVersion:Beryllium
summary:Sci 4.7
```

As you can see the program `system_os` prints several key-value pairs which will be added to the ones which are in `system_os.testdef`. If there is a key which appears in both the file and output, the value of the output will be used. That means it is possible to overwrite key-value pairs from `system_os.testdef` with values from `system_os`.

After merging these pairs the final *Test* will be sent to all *TestConsumer* using the `event` function.

3.1.2 PyTester

The *PyTester* [`clients/wn/src/producer/pytester.py`] is loading python files from a directory and executes their method `getTest()`. This method has to return a *Test* class.

Be aware of the fact that the *PyTester* will not be executed in a separate thread. It is meant to be very lightweight and fast. This means that the loaded python scripts should not do a lot of things, otherwise they would slow down the whole *Worker Node Client*.

As in *ProcTester* the file `testdef.default` contains default tuples which will be merged with the test classes returned by all python modules of a certain directory.

Configuration

The *PyTester* accepts the following arguments:

- **directories** (mandatory) specifies the directories which will be scanned for python files separated by colon.

Example

Consider the following configuration. The `grid-cm-client-wn.conf` contains a section for the *PyTester* like so:

```
[gcm.clients.wn.producer.pytester.PyTester]
directories=%(GCM_HOME)s/lib/gcm/tests/wn
```

The directory `%(GCM_HOME)s/lib/gcm/tests/wn` contains a file `python_opensssl.py` which looks like this:

```
from gcm.common.test import Test

def getTest():
    t = Test("wn_software_python_openssl")
    t.testVersion = "1.0"
    try:
        import OpenSSL
        t.version = str(OpenSSL.__version__)
        t.summary = "y"
    except Exception, err:
        t.summary = "n"
        t.data = err
    return t
```

This python file checks whether the module *OpenSSL* is installed and returns either *yes* and the according version or *no* and an error message.

The returned *Test* class will then be forwarded to all *TestConsumer*.

3.1.3 SignedTester

The *SignedTester* [`clients/wn/src/producer/signed.py`] is a wrapper around both a *PyTester* and *ProcTester* to be able to verify the origin of the tests.

Therefore the *SignedTester* looks for signed tar balls (`*.tar.smime`) in a directory. It then verifies the validity of the certificate used to sign the tar ball. If it is valid it extracts the content into a temporary directory and uses it as the input parameter for *PyTester* and *ProcTester*.

Configuration

- **directories** (mandatory) specifies the directories which will be scanned for signed tar balls separated by colon.
- **certs** (mandatory) is a directory which contains the certificate of the certificate authority.
- **allowed** (mandatory) is a colon separated list of allowed distinguish names. (the subject of the certificate)
- **timeout** and **threads** as in *ProcTester*

Procedure

Write your tests and put all of them into a single tar ball. Then sign it using the following steps:

1. Encode the tar ball usign Base64.

```
openssl base64 -in yourfile.tar -out yourfile.tar.b64
```

2. Sign the tar ball using SMIME.

```
openssl smime -sign -signer yourCertificate.cert
    -inkey yourKey.key
    -in yourfile.tar.b64
    -out yourfile.tar.smime
```

You can also use the python function `signFile(in, out, cert, key)` located in [`common/src/openssl.py`] which does exactly the same.

In order to verifiy that your file was signed properly use these commands:

1. Verify the tar ball using SMIME.

```
openssl smime -verify -in yourfile.tar.smime
    -CApath /path/to/CA
    -signer temp.cert
    -out signedtar.tar
```

2. Get the certificates subject to check it against the configuration file.

```
openssl x509 -subject -noout -in temp.cert
```

You can also use the python equivalent functions `verifyFile(in, out, certs)` and `x509subject(cert)`. The latter one will return the subject which has to appear in the configuration file.

3.2 TestConsumer

A *TestConsumer* [`common/src/testconsumer.py`] retrieves *Tests* from a *TestProducer* and processes them somehow. To support the 3 phases it has functions `setup(self)`, `consume(self, test)` and `cleanup(self)` whereas `consume` will be executed with a *Test* as soon as a *TestProducer* was calling its `event` function.

3.2.1 PyamqSender

??

The *PyamqSender* is a *TestConsumer* which sends the *Tests* it is receiving with *ActiveMQ* to a certain topic. It uses the module *Pyamq* located in [`common/src/pyamq`].

Additionally it is possible to encrypt messages so that only allowed people can read them.

Configuration

The *PyamqSender* 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 client 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 all messages will be sent. For example `/topic/grid.config.workernode`
- **connections** (optional) is the number of simultaneous connections. (default 1)
- **random** (optional) states whether the next connection is established to a broker choosen randomly from the list of brokers.
- **timeout** (optional) is the overall timeout for the *PyamqSender* in seconds. As soon as the timeout exceeds it will stop immediately sending messages. If not defined it will try to send the messages until all are sent or no broker are left.
- **pubkey** (optional) is a path to a public key file (suitable for OpenSSL) which will be used to encrypt *Tests* which have a `pyamqEncrypt:1` argument.
- **testfilter** (optional) is a regular expression as described in pythons *re* module [7]. It will sort out all *Tests* whose `testName` attribute does not match the regular expression.

The configuration of *Pyamq* can be changed by a test using the following tuples:

- **pyamqEncrypt** will force the message to be encrypted using **pubkey** before being sent.
- **pyamqSubtopic** will redirect a message to a subtopic of the global **destination**.

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-client-wn.conf` contains a section for the *PyamqSender* like so:

```
[gcm.clients.wn.consumer.pyamqsender.PyamqSender]
timeout=29
broker=stomp://gridmsg101:6163?timeout=3
destination=/topic/grid.config.workernode
connections=1
random=0
pubkey=%(GCM_HOME)s/lib/gcm/certs/grid-cm-client-wn.pubkey
testfilter=(gcm.*)
```

Now assume that a *TestProducer* is producing the following:

```
testName:gcmExampleTest
testVersion:0.1
pyamqEncrypt:1
pyamqSubtopic:myTopic
summary:secret-value
```

When the *Test* is being received by the *PyamqSender* it tries to match the `testName` value `gcmExampleTest` with `(gcm.*)`. Since these strings match, it then encrypts the *Test* with the public key located at `%(GCM_HOME)s/lib/gcm/certs/grid-cm-client-wn.pubkey`. After that it forwards the message to the module *Pyamq* which has almost 29 seconds left to connect to the broker `gridmsg101` and to send it to the topic `grid.config.workernode.myTopic`.

3.2.2 TestFileSaver

The *TestFileSaver* [`clients/wn/src/consumer/testfilesaver.py`] is a *TestConsumer* which writes all tests it receives to a file.

The file it creates has the following format:

```
[testName1]
key1:val1
key2:val2
[testName2]
key3:val3
```

The `detailsdata` attribute will not be saved.

Configuration

The *TestFileSaver* accepts the following arguments:

- **path** (mandatory) is the path to the file which will be created.
- **testfilter** (optional) is a regular expression as described in pythons *re* module [7]. It will sort out all *Tests* whose `testName` attribute does not match the regular expression.

Example

Consider the following configuration. The `grid-cm-client-wn.conf` contains a section for the *TestFileSaver* like so:

```
[gcm.clients.wn.consumer.testfilesaver.TestFileSaver]
path=%(HOME)s/grid-cm-client-wn-tests.data
testfilter=(atest|anothertest)
```

Now assume that a *TestProducer* is producing the following:

```
testName:atest
testVersion:0.1
summary:aresult
detailsdata:this is a multi
line value
```

When the *Test* is being received by the *TestFileSaver* it tries to match the `testName` value `atest` with `(atest|anothertest)`. Since these strings match, it will save it to the file `%(HOME)s/grid-cm-client-wn-tests.data` like so:

```
[atest]
testName:atest
testVersion:0.1
summary:aresult
```

3.2.3 ClientTester

The *ClientTester* is both a *TestConsumer* and a *TestProducer*. It is used to identify what *Tests* were being produced during the execution of the *Worker Node Client*. It collects all `testName` attributes in a list and creates a new *Test*. Additionally it records log messages and writes down how long the *Worker Node Client* was being executed. Therefore it should be the first component in both the `consumer` and `producer` list in the client configuration. See section 2.2 for a description on how to do that.

It also maintains the memory of the client.

Configuration

The *ClientTester* accepts the following arguments:

- **level** (optional) describes which log messages are being tracked. See pythons logging module [5] for details.
- **memoryFile** (optional) is the path to the memory file.

Memory

The memory of the client is just a file which contains a list of test names and their last execution timestamp. This allows the *ProcTester* to schedule their tests in a more precise way rather than only define a probability. It also contains the clients last execution time so that the client can be scheduled e.g. to be executed just once a day.

The *ClientTester* updates this list of timestamps with the tests it receives during the execution of the client.

Example

The *Worker Node Client* client consists only of one *TestProducer* and the *ClientTester*. The *TestProducer* creates 3 *Tests*: `test1`, `test2` and `test3`.

After all *Tests* are being created the third phase (cleanup) will be started. Now the *ClientTest* will create a test called `wn_client` which will look like this:

```
testName:wn_client
testVersion:1.0
executionTime:1.234
testlist:[test1, test2, test3]
memoryAvailable:0
detailsdata: This are log messages
from the Worker Node Client.
```

4 JobWrapper

The label *GCM WN Client JobWrapper* stands for a simple bash script which integrates the *Worker Node Client* into the *gLite* middleware. It will be called by another script which is also called *JobWrapper* [3]. This script will be executed each time a job is being started on a worker node and will launch all programs in a certain directory. That means the *GCM WN Client JobWrapper* is located in this directory.

However, the *GCM WN Client JobWrapper* just looks for the *Worker Node Client* configuration file, checks whether `active` has the value 1 and if so it then executes the *Worker Node Client*.

5 Deployment

These are some random notes which should be considered when releasing a new version of the *Worker Node Client*:

- Update version information:
  - Updating the changelog in `grid-cm.spec`.
  - Increasing the versions in `grid-cm.spec`, the `__VERSION__` variable in `clients/wn/src/clients.py`.
- Check *Test* file permissions:
  - Check that new executable *Test* files were added as executables to SVN. (They have to have the executable flag when adding the file) Otherwise those files will not get the executable flag when checking out from the repository. And that means they will not appear in the rpm-packages.

6 References

[1] Apache. Activemq. http://activemq.apache.org/.

[2] J. Casey. Grid monitoring probes specification. https://twiki.cern.ch/twiki/bin/view/LCG/GridMonitoringProbeSpecification, 2008.

[3] P. Nyczyk. Goc wiki - sam jobwrapper tests. http://goc.grid.sinica.edu.tw/gocwiki/SAM_jobwrapper_tests.

[4] Python. Configparser - configuration file parser. http://docs.python.org/library/configparser.html.

[5] Python. Logging - logging facility for python. http://docs.python.org/library/logging.html.

[6] Python. Python programming language. http://www.python.org/.

[7] Python. re - regular expression operations. http://docs.python.org/library/re.html.