gc3pie Documentation

Release 1.0rc4

Sergio Maffioletti, Mark Monroe, Riccardo Murri, Mike Packard

August 23, 2015

Contents

CHAPTER 1

Introduction

GC3Libs is a python package for controlling the life-cycle of a Grid or batch computational job.

GC3Libs provides services for submitting computational jobs to Grids and batch systems, controlling their execution, persisting job information, and retrieving the final output.

GC3Libs takes an application-oriented approach to batch computing. A generic *Application* class provides the basic operations for controlling remote computations, but different *Application* subclasses can expose adapted interfaces, focusing on the most relevant aspects of the application being represented.

This document is the technical reference for the GC3Libs programming model, aimed at programmers who want to use GC3Libs to implement computational workflows in Python.

1.1 Outline of the contents

The Programming overview section presents the main concepts behind GC3Libs programming.

The GC3Libs modules section is a comprehensive list of all the modules, classes and functions comprising GC3Libs; its content is automatically generated from docstrings in the source code.

CHAPTER 2

Installation of GC3Utils

Author Riccardo Murri <riccardo.murri@gmail.com>

Date 2010-10-06

Revision $Revision$

2.1 Installation

These instructions show how to install GC3Pie from the GC3 source repository into a separate python environment (called virtualenv). Installation into a virtualenv has two distinct advantages:

- All code is confined in a single directory, and can thus be easily replaced/removed.
- Better dependency handling: additional Python packages that GC3Pie depends upon can be installed even if they conflict with system-level packages.

0. Install software prerequisites:

   - On Debian/Ubuntu, install packages: `subversion`, `python-dev`, `python-profiler` and the C/C++ compiler:

```
apt-get install subversion python-dev python-profiler gcc g++
```

   - On CentOS5, install packages `subversion` and `python-devel` and the C/C++ compiler:

```
yum install subversion python-devel gcc gcc-c++
```

   - On other Linux distributions, you will need to install:
     - the `svn` command (from the SubVersion VCS)
     - Python development headers and libraries (for installing extension libraries written in C/C++)
     - the Python package `pstats` (it's part of the Python standard library, but sometimes it needs separate installation)
     - a C/C++ compiler (this is usually installed by default).

   In order to use the ARC backend (required for SMSCG), you need the NorduGrid/ARC binaries and a working `slcs-init` command installed on the same machine where GC3Pie are. You can find instructions for installing it at:

   http://www.smscg.ch/WP/middleware/release_2_0/smscg_ui_Binary_Installation.html

   Additional OS-specific installation details can be found at:

   http://code.google.com/p/gc3pie/wiki/OSSpecificInstallDetails

1. Choose a directory where the GC3Pie software will be installed; any directory that's writable by your Linux account will be ok.

   If you are installing system-wide as `root`, we suggest you install GC3Pie into `/opt/gc3pie`.

   If you are installing as a normal user, we suggest you install GC3Pie into `$HOME/gc3pie`.

2. If it's not already installed, get the virtualenv Python package and install it:

```
  wget http://pypi.python.org/packages/source/v/virtualenv/virtualenv-1.5.1.tar.gz
  tar -xzf virtualenv-1.5.1.tar.gz && rm virtualenv-1.5.1.tar.gz
  cd virtualenv-1.5.1/

If you are installing as `root`, the following command is all you
need:

  python setup.py install

If instead you are installing as a normal, unprivileged user,
things get more complicated::

  export PYTHONPATH=$HOME/lib64/python:$HOME/lib/python:$PYTHONPATH
  export PATH=$PATH:$HOME/bin
  mkdir -p $HOME/lib/python
  python setup.py install --home $HOME

(You will also need to add the two `export` lines above to the
`$HOME/.bashrc` file -if using the `bash` shell- or to the
`$HOME/.cshrc` file -if using the `tcsh` shell.)

In any case, once `virtualenv`_ has been installed, you can exit
its directory and remove it::

  cd ..
  rm -rf virtualenv-1.5.1
```

3. Create a virtualenv to host the `gc3pie` installation at the directory you chose in Step 1.:

```
virtualenv $HOME/gc3pie # use '/opt/gc3pie' if installing as root
cd $HOME/gc3pie/
source bin/activate
```

4. Check-out the `gc3pie` files in a `src/` directory:

```
svn co http://gc3pie.googlecode.com/svn/branches/1.0/gc3pie src
```

5. Install the `gc3pie` in "develop" mode, so any modification pulled from subversion is immediately reflected in the running environment:

```
cd src/
env CC=gcc ./setup.py develop
cd .. # back into the `gc3pie` directory
```

   This will place all the GC3Pie command into the `gc3pie/bin/` directory.

6. GC3Pie comes with driver scripts to run and manage large families of jobs from a few selected applications. These scripts are not installed by default because not everyone needs them.

   Run the following commands to install the driver scripts for the applications you need:

```
# if you are insterested in GAMESS, do the following
ln -s '../src/gc3apps/gamess/ggamess.py' bin/ggamess

# if you are insterested in Rosetta, do the following
ln -s '../src/gc3apps/rosetta/gdocking.py' bin/gdocking
ln -s '../src/gc3apps/rosetta/grosetta.py' bin/grosetta
```

```
# if you are insterested in Codeml, do the following
ln -s '../src/gc3apps/codeml/gcodeml.py' bin/gcodeml
```

7. Before you can actually run the GC3Pie, you need to have a working configuration file; *the ConfigurationFile Wiki page <http://code.google.com/p/gc3pie/wiki/ConfigurationFile>* provides an explanation of the syntax to use in configuration files.

   An example configuration file, enabling access to the SMSCG infrastructure can be found at:

   http://gc3pie.googlecode.com/svn/branches/1.0/gc3pie/gc3libs/etc/gc3pie.conf.smscg

   Before you can actually use this file, you will need to insert into it three values, for which we can provide no default: `aai_username`, `idp`, and `vo`.

   aai_username: This is the "username" you are asked for when accessing any SWITCHaai/Shibboleth web page, e.g., https://gc3-aai01.uzh.ch/secure/

   idp: Find this out with the command "slcs-info": it prints a list of IdP (Identity Provider IDs) followed by the human-readable name of the associated institution. Pick the one that corresponds to you University. It is always the last two components of the University's Internet domain name (e.g., "uzh.ch" or "ethz.ch").

   vo: In order to use SMSCG, you must sign up to a VO (Virtual Organisation). One the words "life", "earth", "atlas" or "crypto" should be here. Find out more at: http://www.smscg.ch/www/user/

2.2 Upgrade

These instructions show how to upgrade the GC3Pie scripts to the latest version found in the GC3 svn repository.

1. *cd* to the directory containing the GC3Pie virtualenv; assuming it's named `gc3pie` as in the above installation instructions, you can issue the commands:

```
cd $HOME/gc3pie # use '/opt/gc3pie' if root
```

2. Activate the virtualenv

```
source bin/activate
```

3. Upgrade the *gc3pie* source and run the *setup.py* script again:

```
cd src
svn up
env CC=gcc ./setup.py develop
```

Note: A major restructuring of the SVN repository took place in r1124 to r1126 (Feb. 15, 2011); if your sources are older than SVN r1124, these upgrade instructions will not work, and you must *reinstall completely*. You can check what version the SVN sources are, by running the *svn info* command in the *src* directory: watch out for the *Revision:* line.

2.3 HTML Documentation

HTML documentation for the GC3Libs programming interface can be read online at:

http://gc3pie.googlecode.com/svn/branches/1.0/docs/html/index.html

You can also generate a local copy from the sources:

```
cd $HOME/gc3pie # use '/opt/gc3pie' if root
cd src/docs
make html
```

Note that you need the Python package Sphinx in order to build the documentation locally.

CHAPTER 3

Programming overview

GC3Libs takes an application-oriented approach to asynchronous computing. A generic `Application` class provides the basic operations for controlling remote computations and fetching a result; client code should derive specialized sub-classes to deal with a particular application, and to perform any application-specific pre- and post-processing.

The generic procedure for performing computations with GC3Libs is the following:

1. Client code creates an instance of an *Application* sub-class.
2. Asynchronous computation is started by submitting the application object; this associates the application with an actual (possibly remote) computational job.
3. Client code can monitor the state of the computational job; state handlers are called on the application object as the state changes.
4. When the job is done, the final output is retrieved and a post-processing method is invoked on the application object.

At this point, results of the computation are available and can be used by the calling program.

The `Application` class (and its sub-classes) alow client code to control the above process by:

1. Specifying the characteristics (computer program to run, input/output files, memory/CPU/duration requirements, etc.) of the corresponding computational job. This is done by passing suitable values to the `Application` constructor. See the `Application` constructor documentation for more info.
2. Providing methods to control the "life-cycle" of the associated computational job: start, check execution state, stop, retrieve a snapshot of the output files. There are actually two different interfaces for this, detailed below:

   (a) A *passive* interface: a `Core` or a `Engine` object is used to start/stop/monitor jobs associated with the given application. For instance:

```
a = GamessApplication(...)

# create a `Core` object; only one instance is needed
g = Core(...)

# start the remote computation
g.submit(a)

# periodically monitor job execution
g.update_job_state(a)

# retrieve output when the job is done
g.fetch_output(a)
```

   The passive interface gives client code full control over the lifecycle of the job, but cannot support some use cases (e.g., automatic application re-start).

As you can see from the above example, the passive interface is implemented by methods in the `Core` and `Engine` classes (they implement the same interface). See those classes documentation for more details.

(b) An *active* interface: this requires that the `Application` object be attached to a `Core` or `Engine` instance:

```
a = GamessApplication(...)

# create a `Core` object; only one instance is needed
g = Core(...)

# tell application to use the active interface
a.attach(g)

# start the remote computation
a.submit()

# periodically monitor job execution
a.update_job_state()

# retrieve output when the job is done
a.fetch_output()
```

With the active interface, application objects can support automated restart and similar use-cases.

When an `Engine` object is used instead of a `Core` one, the job life-cycle is automatically managed, providing a fully asynchronous way of executing computations.

The active interface is implemented by the `Task` class and all its descendants (including `Application`).

3. Providing "state transition methods" that are called when a change in the job execution state is detected; those methods can implement application specific behavior, like restarting the computational job with changed input if the alloted duration has expired but the computation has not finished. In particular, a *postprocess* method is called when the final output of an application is available locally for processing.

   The set of "state transition methods" currently implemented by the `Application` class are: `new()`, `submitted()`, `running()`, `stopped()`, `terminated()` and `postprocess()`. Each method is called when the execution state of an application object changes to the corresponding state; see each method's documentation for exact information.

In addition, GC3Libs provides *collection* classes, that expose interfaces *2.* and *3.* above, allowing one to control a set of applications as a single whole. Collections can be nested (i.e., a collection can hold a mix of `Application` and `TaskCollection` objects), so that workflows can be implemented by composing collection objects.

Note that the term *computational job* (or just *job*, for short) is used here in a quite general sense, to mean any kind of computation that can happen independently of the main thread of the calling program. GC3Libs currently provide means to execute a job as a separate process on the same computer, or as a batch job on a remote computational cluster.

3.1 Execution model of GC3Libs applications

An *Application* can be regarded as an abstraction of an independent asynchronous computation, i.e., a GC3Libs' *Application* behaves much like an independent UNIX process (but it can actually run on a separate remote computer). Indeed, GC3Libs' *Application* objects mimic the POSIX process model: *Application* are started by a parent process, run independently of it, and need to have their final exit code and output reaped by the calling process.

The following table makes the correspondence between POSIX processes and GC3Libs' *Application* objects explicit.

os module function	*Core* function	purpose
exec	Core.submit	start new job
kill(..., SIGTERM)	Core.kill	terminate executing job
wait(..., WNOHANG)	Core.update_job_state	get job status
•	Core.fetch_output	retrieve output

Note:

1. With GC3Libs, it is not possible to send an arbitrary signal to a running job: jobs can only be started and stopped (killed).
2. Since POSIX processes are always executed on the local machine, there is no equivalent of the GC3Libs *fetch_output*.

3.1.1 Application exit codes

POSIX encodes process termination information in the "return code", which can be parsed through *os.WEXITSTATUS*, *os.WIFSIGNALED*, *os.WTERMSIG* and relative library calls.

Likewise, GC3Libs provides each `Application` object with an *execution.returncode* attribute, which is a valid POSIX "return code". Client code can therefore use *os.WEXITSTATUS* and relatives to inspect it; convenience attributes *execution.signal* and *execution.exitcode* are available for direct access to the parts of the return code. See `Run.returncode()` for more information.

However, GC3Libs has to deal with error conditions that are not catered for by the POSIX process model: for instance, execution of an application may fail because of an error connecting to the remote execution cluster.

To this purpose, GC3Libs encodes information about abnormal job termination using a set of pseudo-signal codes in a job's *execution.returncode* attribute: i.e., if termination of a job is due to some grid/batch system/middleware error, the job's *os.WIFSIGNALED(app.execution.returncode)* will be *True* and the signal code (as gotten from *os.WTERMSIG(app.execution.returncode)*) will be one of those listed in the `Run.Signals` documentation.

3.1.2 Application execution states

At any given moment, a GC3Libs job is in any one of a set of pre-defined states, listed in the table below. The job state is always available in the *.execution.state* instance property of any *Application* or *Task* object; see `Run.state()` for detailed information.

GC3Libs' Job state	purpose	can change to
NEW	Job has not yet been submitted/started (i.e., gsub not called)	SUBMITTED (by gsub)
SUBMITTED	Job has been sent to execution resource	RUNNING, STOPPED
STOPPED	Trap state: job needs manual intervention (either user- or sysadmin-level) to resume normal execution	TERMINATED (by gkill), SUBMITTED (by miracle)
RUNNING	Job is executing on remote resource	TERMINATED
TERMINATED	Job execution is finished (correctly or not) and will not be resumed	None: final state

When an `Application` object is first created, its *.execution.state* attribute is assigned the state NEW. After a successful start (via *Core.submit()* or similar), it is transitioned to state SUBMITTED. Further transitions to RUNNING or STOPPED or TERMINATED state, happen completely independently of the creator program: the *Core.update_job_state()* call provides updates on the status of a job. (Somewhat like the POSIX *wait(..., WNOHANG)* system call, except that GC3Libs provide explicit RUNNING and STOPPED states, instead of encoding them into the return value.)

The STOPPED state is a kind of generic "run time error" state: a job can get into the STOPPED state if its execution is stopped (e.g., a SIGSTOP is sent to the remote process) or delayed indefinitely (e.g., the remote

batch system puts the job "on hold"). There is no way a job can get out of the STOPPED state automatically: all transitions from the STOPPED state require manual intervention, either by the submitting user (e.g., cancel the job), or by the remote systems administrator (e.g., by releasing the hold).

The TERMINATED state is the final state of a job: once a job reaches it, it cannot get back to any other state. Jobs reach TERMINATED state regardless of their exit code, or even if a system failure occurred during remote execution; actually, jobs can reach the TERMINATED status even if they didn't run at all!

A job that is not in the NEW or TERMINATED state is said to be a "live" job.

3.1.3 Computational job specification

One of the purposes of GC3Libs is to provide an abstraction layer that frees client code from dealing with the details of job execution on a possibly remote cluster. For this to work, it necessary to specify job characteristics and requirements, so that the GC3Libs scheduler can select an appropriate computational resource for executing the job.

GC3Libs *Application* provide a way to describe computational job characteristics (program to run, input and output files, memory/duration requirements, etc.) loosely patterned after ARC's xRSL language.

The description of the computational job is done through keyword parameters to the `Application` constructor, which see for details. Changes in the job characteristics *after* an `Application` object has been constructed are not currently supported.

CHAPTER 4

GC3Libs modules

4.1 *gc3libs*

4.2 *gc3libs.core*

4.3 *gc3libs.Default*

Warning: This module is deprecated and will be removed in a future release. Do not depend on it.

4.4 *gc3libs.Exceptions*

4.5 *gc3libs.persistence*

4.6 *gc3libs.application*

4.7 *gc3libs.application.gamess*

4.8 *gc3libs.application.rosetta*

4.9 *gc3libs.authentication*

4.10 *gc3libs.authentication.ssh*

4.11 *gc3libs.authentication.grid*

4.12 *gc3libs.backends*

4.13 *gc3libs.backends.arc*

4.14 *gc3libs.backends.sge*

4.15 *gc3libs.backends.transport*

4.16 *gc3libs.utils*

4.17 *gc3libs.Resource*

Warning: This module is deprecated and will be removed in a future release. Do not depend on it.

4.18 *gc3libs.scheduler*

4.19 *gc3libs.InformationContainer*

Warning: This module is deprecated and will be removed in a future release. Do not depend on it.

CHAPTER 5

GC3Utils modules

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

g

G