gc3pie Documentation

Release 2.6.7

Sergio Maffioletti, Antonio Messina, Mark Monroe, Riccardo Murri

Aug 10, 2021

Contents

CHAPTER 1

Introduction

GC3Pie is a Python package for running large job campaigns on diverse batch-oriented execution environments (for instance: a Sun/Oracle/Open Grid Engine cluster, or the Swiss National Distributed Computing Infrastructure SMSCG). It also provides facilities for implementing command-line driver scripts, in the form of Python object classes whose behavior can be customized by overriding specified object methods.

GC3Pie documentation is divided in three sections:

- *User Documentation*: info on how to install, configure and run GC3Pie applications.
- *Programmer Documentation*: info for programmers who want to use the GC3Pie libraries to write their own scripts and applications.
- *Contributors documentation*: detailed information on how to contribute to GC3Pie and get your code included in the main library.

CHAPTER 2

Table of Contents

2.1 User Documentation

This section describes how to install and configure GC3Pie, and how to run *The GC3Apps software* and *The GC3Utils software*.

2.1.1 Table of Contents

Installation of GC3Pie

Quick start

We provide an installation script which automatically tries to install GC3pie in your home directory. The quick installation procedure has only been tested on variants of the GNU/Linux operating system; however, the script should work on MacOSX as well, provided you follow the preparation steps outlined in the "MacOSX installation" section below.

To install GC3Pie: (1) download the installation script into a file install.py, then (2) type this at your terminal prompt:

```
python install.py
```

The above command creates a directory `$HOME/gc3pie` and installs the latest release of GC3Pie and all its dependencies into it.

Alternatively, you can also perform both steps at the terminal prompt:

```
# use this if the `wget` command is installed
wget -O install.py https://raw.githubusercontent.com/uzh/gc3pie/master/install.py
python install.py

# use this if the `curl` command is installed instead
curl -O https://raw.githubusercontent.com/uzh/gc3pie/master/install.py
python install.py
```

Choose either one of the two methods above, depending on whether `wget` or `curl` is installed on your system (Linux systems normally have `wget`; MacOSX normally uses `curl`).

In case you have trouble running the installation script, please send an email to gc3pie@googlegroups.com or post a message on the web forum https://groups.google.com/forum/#!forum/gc3pie. Include the full output of the script in your email, in order to help us to identify the problem.

Now you can check your GC3Pie installation; follow the on-screen instructions to activate the virtual environment. Then, just type the command:

```
gc3utils --help
```

and you should see the following output appear on your screen:

```
Usage: gc3utils COMMAND [options]

Command `gc3utils` is a unified front-end to computing resources.
You can get more help on a specific sub-command by typing::
  gc3utils COMMAND --help
where command is one of these:
  clean
  cloud
  get
  info
  kill
  resub
  select
  servers
  stat
  tail
```

If you get some errors, do not despair! The GC3Pie users mailing-list is there to help you :-) (You can also post to the same forum using a web interface at https://groups.google.com/forum/#!forum/gc3pie.)

With the default configuration file, GC3Pie is set up to only run jobs on the computer where it is installed. To run jobs on remote resources, you need to edit the configuration file; the Configuration file documentation provides an explanation of the syntax.

Non-standard installation options

The installation script accept a few options that select alternatives to the standard behavior. In order to use these options, you have to:

1. download the installation script into a file named `install.py`:

   ```
   wget https://raw.githubusercontent.com/uzh/gc3pie/master/install.py
   ```

2. run the command:

   ```
   python install.py [options]
   ```

   replacing the string `[options]` with the actual options you want to pass to the script. Also, the `python` command should be the Python executable that you want to use to run GC3Pie applications.

The accepted options are as follows:

`--feature LIST`

Install optional features (comma-separated list). Currently defined features are:

- `openstack`: support running jobs in VMs on OpenStack clouds
- `ec2`: support running jobs in VMs on OpenStack clouds
- `optimizer`: install math libraries needed by the optimizer library

For instance, to install all features use `-a openstack,ec2,optimizer`. To install no optional feature, use `-a none`.

By default, all cloud-related features are installed.

`-d DIRECTORY`

Install GC3Pie in location `DIRECTORY` instead of `$HOME/gc3pie`

`--overwrite`

Overwrite the destination directory if it already exists. Default behavior is to abort installation.

`--develop`

Instead of installing the latest *release* of GC3Pie, it will install the *master branch* from the GitHub repository.

`--yes`

Run non-interactively, and assume a "yes" reply to every question.

`--no-gc3apps`

Do not install any of the GC3Apps, e.g., `gcodeml`, `grosetta` and `ggamess`.

Manual installation

In case you can't or don't want to use the automatic installation script, the following instructions will guide you through all the steps needed to manually install GC3Pie on your computer.

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 these system packages:

     ```
     apt-get install gcc g++ git python-dev libffi-dev libssl-dev make
     ```

   - On CentOS5, install these packages:

     ```
     yum install git python-devel gcc gcc-c++ libffi-devel make openssl-devel
     ```

   - On other Linux distributions, you will need to install:
     - the `git` command (from the Git VCS);
     - Python development headers and libraries; (for installing extension libraries written in C/C++)
     - a C/C++ compiler (this is usually installed by default);
     - include files for the FFI and OpenSSL libraries.

1. If virtualenv is not already installed on your system, get the Python package and install it:

```
wget http://pypi.python.org/packages/source/v/virtualenv/virtualenv-1.7.tar.gz
tar -xzf virtualenv-1.7.tar.gz && rm virtualenv-1.7.tar.gz
cd virtualenv-1.7/
```

   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 *have 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-1.7
```

2. Create a virtualenv to host the GC3Pie installation, and `cd` into it:

```
virtualenv --system-site-packages $HOME/gc3pie
cd $HOME/gc3pie/
source bin/activate
```

   In this step and in the following ones, the directory `$HOME/gc3pie` is going to be the installation folder of GC3Pie. You can change this to another directory path; 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` instead.

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

```
git clone https://github.com/uzh/gc3pie.git src
```

4. Install the `gc3pie` in "develop" mode, so any modification pulled from GitHub 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.

5. 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
```

6. Now you can check your GC3Pie installation; just type the command:

```
gc3utils --help
```

and you should see the following output appear on your screen:

```
Usage: gc3utils COMMAND [options]

Command `gc3utils` is a unified front-end to computing resources.
You can get more help on a specific sub-command by typing::
  gc3utils COMMAND --help
where command is one of these:
  clean
  cloud
  get
  info
  kill
  resub
  select
  servers
  stat
  tail
```

If you get some errors, do not despair! The *GC3Pie users mailing-list <gc3pie@googlegroups.com>* is there to help you :-) (You can also post to the same forum using the web interface at https://groups.google.com/forum/#!forum/gc3pie.)

7. With the default configuration file, GC3Pie is set up to only run jobs on the computer where it is installed. To run jobs on remote resources, you need to edit the configuration file; the configuration file documentation provides an explanation of the syntax.

Upgrade

If you used the installation script, the fastest way to upgrade is just to reinstall:

0. De-activate the current GC3Pie virtual environment:

```
deactivate
```

(If you get an error "command not found", do not worry and proceed on to the next step; in case of other errors please stop here and report to the *GC3Pie users mailing-list <mailto:gc3pie.googlegroups.com>*.)

1. Move the `$HOME/gc3pie` directory to another location, e.g.:

```
mv $HOME/gc3pie $HOME/gc3pie.OLD
```

2. Reinstall GC3Pie using the quick-install script (top of this page).

3. Once you have verified that your new installation is working, you can remove the `$HOME/gc3pie.OLD` directory.

If instead you installed GC3Pie using the "manual installation" instructions, then the following steps will update GC3Pie to the latest version in the code repository:

1. **cd** to the directory containing the GC3Pie virtualenv; assuming it is 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.

MacOSX Installation

Installation on MacOSX machines is possible, however there are still a few issues. If you need MacOSX support, please let us know on the *GC3Pie users mailing-list <mailto:gc3pie@googlegroups.com>* or by posting a message using the web interface at https://groups.google.com/forum/#!forum/gc3pie.

1) Standard usage of the installation script (i.e., with no options) works, but you have to use *curl* since *wget* is not installed by default.

2) In order to install GC3Pie you will need to install XCode and, in some of the MacOSX versions, also the *Command Line Tools for XCode*

3) Options can only be given in the abbreviated one-letter form (e.g., `-d`); the long form (e.g., `--directory`) will not work.

4) The *shellcmd* backend of GC3Pie depends on the GNU `time` command, which is not installed on MacOSX by default. This means that with a standard MacOSX installation the *shellcmd* resource will **not** work. However:

   - other resources, like *pbs* via *ssh* transport, will work.
   - you can install the GNU time command either via MacPorts, Fink, Homebrew or from this url. After installing it you don't need to update your `PATH` environment variable, it's enough to set the `time_cmd` option in your GC3Pie configuration file.

HTML Documentation

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

http://gc3pie.readthedocs.io/

If you installed GC3Pie manually, or if you installed it using the `install.py` script with the `--develop` option, you can also access a local copy of the documentation from the sources:

```
cd $HOME/gc3pie # or wherever the gc3pie virtualenv is installed
cd src/docs
make html
```

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

Configuration File

Location

All commands in *The GC3Apps software* and *The GC3Utils software* read a few configuration files at startup:

- system-wide one located at `/etc/gc3/gc3pie.conf`,
- a virtual-environment-wide configuration located at `$VIRTUAL_ENV/etc/gc3/gc3pie.conf`, and
- a user-private one at `$HOME/.gc3/gc3pie.conf`, or, alternately, the file in the location pointed to by the environmental variable `GC3PIE_CONF`.

All these files are optional, but at least one of them must exist.

All files use the same format. The system-wide one is read first, so that users can override the system-level configuration in their private file. Configuration data from corresponding sections in the configuration files is merged; the value in files read later overrides the one from the earler-read configuration.

If you try to start any GC3Utils command without having a configuration file, a sample one will be copied to the user-private location `~/.gc3/gc3pie.conf` and an error message will be displayed, directing you to edit the sample file before retrying.

Configuration file format

The GC3Pie configuration file follows the format understood by Python ConfigParser; see http://docs.python.org/library/configparser.html for reference.

Here is an example of what the configuration file looks like:

```
[auth/none]
type=none

[resource/localhost]
# change the following to `enabled=no` to quickly disable
enabled=yes
type=shellcmd
transport=local
auth=none
max_cores=2
max_cores_per_job=2
# ...
```

You can see that:

- The GC3Pie configuration file consists of several configuration sections. Each configuration section starts with a keyword in square brackets and continues until the start or the next section or the end of the file (whichever happens first).

- A section's body consists of a series of `word=value` assignments (we call these *configuration items*), each on a line by its own. The word before the = sign is called the *configuration key*, and the value given to it is the *configuration value*.
- Lines starting with the # character are comments: the line is meant for human readers and is completely ignored by GC3Pie.

The following sections are used by the GC3Apps/GC3Utils programs:

- `[DEFAULT]` – this is for global settings.
- `[auth/name]` – these are for settings related to identity/authentication (identifying yourself to clusters & grids).
- `[resource/name]` – these are for settings related to a specific computing resource (cluster, grid, etc.)

Sections with other names are allowed but will be ignored.

The `DEFAULT` section

The `[DEFAULT]` section is optional.

Values defined in the `[DEFAULT]` section can be used to insert values in other sections, using the `%(name)s` syntax. See documentation of the Python SafeConfigParser object at http://docs.python.org/library/configparser.html for an example.

`auth` sections

There can be more than one `[auth]` section.

Each authentication section must begin with a line of the form:

```
[auth/name]
```

where the *name* portion is any alphanumeric string.

You can have as many `[auth/name]` sections as you want; any name is allowed provided it's composed only of letters, numbers and the underscore character _. (Examples of valid names are: `[auth/cloud]`, `[auth/ssh1]`, and `[auth/user_name]`)

This allows you to define different auth methods for different resources. Each `[resource/name]` section can reference one (and one only) authentication section, but the same `[auth/name]` section can be used by more than one `[resource/name]` section.

Authentication types

Each `auth` section *must* specify a `type` setting.

`type` defines the authentication type that will be used to access a resource. There are three supported authentication types:

type=...	Use this for ...
ec2	EC2-compatible cloud resources
none	Resources that need no authentication
ssh	Resources that will be accessed by opening an SSH connection to the front-end node of a cluster

`none`-type authentication

This is for resources that actually need no authentication (`transport=local`) but still need to reference an `[auth/*]` section for syntactical reasons.

GC3Pie automatically inserts in every configuration file a section `[auth/none]`, which you can reference in resource sections with the line `auth=none`.

Because of the automatically-generated `[auth/none]`, there is hardly ever a reason to explicitly write such a section (doing so is not an error, though):

```
[auth/none]
type=none
```

`ssh`-type authentication

For the `ssh`-type auth, the following keys must be provided:

- `type`: must be `ssh`
- `username`: must be the username to log in as on the remote machine

The following configuration keys are instead optional:

- `port`: TCP port number where the SSH server is listening. The default value 22 is fine for almost all cases; change it only if you know what you are doing.
- `keyfile`: path to the (private) key file to use for SSH public key authentication.
- `ssh_config`: path to a SSH configuration file, where to read additional options from (default: `$HOME/ssh/config`:file:. The format of the SSH configuration file is documented in the ssh_config(5) man page.
- `timeout`: maximum amount of time (in seconds) that GC3Pie will wait for the SSH connection to be established.

Note: We advise you to use the SSH config file for setting port, key file, and connection timeout. Options `port`, `keyfile`, and `timeout` could be deprecated in future releases.

Example. The following configuration sections are used to set up two different accounts that GC3Pie programs can use. Which account should be used on which computational resource is defined in the *resource sections* (see below).

```
[auth/ssh1]
type = ssh
username = murri # your username here

[auth/ssh2] # I use a different account name on some resources
type = ssh
username = rmurri
# read additional options from this SSH config file
ssh_config = ~/.ssh/alt-config
```

`ec2`-type authentication

For the `ec2`-type auth, the following keys *can* be provided:

- `ec2_access_key`: Your personal access key to authenticate against the specific cloud endpoint. If not found, the value of the environment variable `EC2_ACCESS_KEY` will be used; if the environment variable is unset, GC3Pie will raise a `ConfigurationError`.
- `ec2_secret_key`: Your personal secret key associated with the above `ec2_access_key`. If not found, the value of the environment variable `EC2_SECRET_KEY` will be used; if the environment variable is unset, GC3Pie will raise a `ConfigurationError`.

Any other key/value pair will be silently ignored.

Example. The following configuration section is used to access an EC2-compatible resource (access and secret keys are of course invalid):

```
[auth/hobbes]
type=ec2
ec2_access_key=1234567890qwertyuiopasdfghjklzxc
ec2_secret_key=cxzlkjhgfdsapoiuytrewq0987654321
```

`resource` sections

Each resource section must begin with a line of the form:

```
[resource/name]
```

You can have as many `[resource/name]` sections as you want; this allows you to define many different resources. Each `[resource/name]` section must reference one (and one only) `[auth/name]` section (by its `auth` key).

Resources currently come in several flavours, distinguished by the value of the `type` key. Valid values for the `type=...` configuration line are listed in the table below.

`type=...`	The resource is ...
`ec2+shellcmd`	a cloud with EC2-compatible APIs: applications are run on Virtual Machines started on the cloud
`lsf`	an LSF batch-queuing system
`pbs`	a TORQUE or PBSPro batch-queuing system
`sge`	a Grid Engine batch-queuing system
`shellcmd`	a single Linux or MacOSX computer: applications are executed by spawning a local UNIX process
`slurm`	a SLURM batch-queuing system

Link to `[auth/...]` sections

All `[resource/name]` sections *must* reference a valid `[auth/aname]` section via the `auth=aname` line. If the `auth=...` line is omitted, GC3Pie's default is `auth=none`.

Type of the resource and the referenced `[auth/...]` section must match:

- Resources of type `ec2+shellcmd` can only reference `[auth/...]` sections of type `ec2`.
- Batch-queuing resources (type is one of `sge`, `pbs`, `lsf`, or `slurm`) and resources of type `shellcmd` can reference `[auth/...]` sections of type `ssh` (when `transport=ssh`) or `[auth/none]` (when `transport=local`).

Configuration keys common to *all* resource types

The following configuration keys are commmon to all resources, regardless of type.

Configuration key	Meaning
`type`	Resource type, see above.
`auth`	Name of a valid `[auth/name]` section; only the authentication section name (after the `/`) must be specified.
`max_cores_per_job`	Maximum number of CPU cores that a job can request; a resource will be dropped during the brokering process if a task requests more cores than this.
`max_memory_per_core`	Maximum amount of memory that a task can request; a resource will be dropped during the brokering process if a task requests more memory than this.
`max_walltime`	Maximum job running time.
`max_cores`	Total number of cores provided by the resource.
`architecture`	Processor architecture. Should be one of the strings `x86_64` (for 64-bit Intel/AMD/VIA processors), `i686` (for 32-bit Intel/AMD/VIA x86 processors), or `x86_64,i686` if both architectures are available on the resource.
`time_cmd`	Path to the GNU **`time`** program. Default is `/usr/bin/time`
`large_file_threshold`	Files less than this size will be copied in one go. Only relevant for SSH transfers; ignored otherwise.
`large_file_chunk_size`	Files larger than the threshold above will be copied in chunks of this size, one chunk at a time. Only relevant for SSH transfers; ignored otherwise.

Configuration keys common to batch-queuing resource types

The following configuration keys can be used in any resource of type `pbs`, `lsf`, `sge`, or `slurm`.

- `spooldir`: Root path to the batch jobs' working directories. GC3Pie will create dedicated temporary working directories, one for each job, within this root folder.

  By default, working directories are created as subdirectories of `$HOME/.gc3pie_jobs`.

  Note: The job working directories *must* be visible (with the same filesystem path) and writable on both the front-end node (with which GC3Pie interacts) and the compute nodes (where a job's payload actually runs).

- `prologue`: Path to a script file, whose contents are *inserted* into the submission script of each application that runs on the resource. Commands from the *prologue* script are executed before the real application; the *prologue* is intended to execute some shell commands needed to setup the execution environment before running the application (e.g. running a `module load ...` command).

  Note: The prologue script **must** be a valid plain **`/bin/sh`** script; she-bang indications will *not* be honored.

- *`application`*`_prologue`: Same as `prologue`, but it is used only when *`application`* matches the name of the application (as specified by the `application_name` attribute on the GC3Pie `Application` instance).

- `prologue_content`: A (possibly multi-line) string that will be *inserted* into the submission script and executed before the real application. Like the `prologue` script, commands must be given using **`/bin/sh`** syntax.

- *application*_prologue_content: Same as prologue_content, but it is used only when *application* matches the name of the application (as specified by the application_name attribute on the GC3Pie Application instance).

Warning: Errors in a prologue script will prevent any application from running on the resource! Keep prologue commands to a minimum and always check their correctness.

If several *prologue*-related options are specified, then commands are inserted into the submission script in the following order:

- first content of the prologue script,
- then content of the *application*_prologue script,
- then commands from the prologue_content configuration item,
- finally commands from the *application*_prologue_content configuration item.

A similar set of options allow defining commands to be executed *after* an application has finished running:

- epilogue: The content of the *epilogue* script will be *inserted* into the submission script and is executed after the real application has been submitted

  Note: The epilogue script **must** be a valid plain **/bin/sh** script; she-bang indications will *not* be honored.

- *application*_epilogue : Same as epilogue, but used only when {application}`:file: matches the name of the application (as specified by the ``application_name attribute on the GC3Pie Application instance).
- epilogue_content: A (possibly multi-line) string that will be *inserted* into the submission script and executed after the real application has completed. Like the epilogue script, commands must be given using **/bin/sh** syntax.
- *application*_epilogue_content : Same as epilogue_content, but used only when *application* matches the name of the application (as specified by the application_name attribute on the GC3Pie Application instance).

Warning: Errors in an epilogue script prevent GC3Pie from reaping the application's exit status. In particular, errors in the epilogue commands can make GC3Pie consider the whole application as failed, and use the epilogue's error exit as the overall exit code.

If several *epilogue*-related options are specified, then commands are inserted into the submission script in the following order:

- first contents of the epilogue script,
- then contents of the *application*_epilogue script,
- then commands from the epilogue_content configuration item,
- finally commands from the *application*_epilogue_content configuration item.

sge resources (all batch systems of the Grid Engine family)

The following configuration keys are required in a sge-type resource section:

- `frontend`: should contain the FQDN (Fully-qualified domain name) of the SGE front-end node. An SSH connection will be attempted to this node, in order to submit jobs and retrieve status info.
- `transport`: Possible values are: `ssh` or `local`. If `ssh`, GC3Pie tries to connect to the host specified in `frontend` via SSH in order to execute SGE commands. If `local`, the SGE commands are run directly on the machine where GC3Pie is installed.

To submit parallel jobs to SGE, a "parallel environment" name must be specified. You can specify the PE to be used with a specific application using a configuration parameter *application name* + `_pe` (e.g., `gamess_pe`, `zods_pe`); the `default_pe` parameter dictates the parallel environment to use if no application-specific one is defined. *If neither the application-specific, nor the ''default_pe'' parallel environments are defined, then submission of parallel jobs will fail.*

When a job has finished, the SGE batch system does not (by default) immediately write its information into the accounting database. This creates a time window during which no information is reported about the job by SGE, as if it never existed. In order not to mistake this for a "job lost" error, GC3Libs allow a "grace time": **`qacct`** job information lookups are allowed to fail for a certain time span after the first time **`qstat`** failed. The duration of this time span is set with the `sge_accounting_delay` parameter, whose default is 15 seconds (matches the default in SGE, as of release 6.2):

- `sge_accounting_delay`: Time (in seconds) a failure in **`qacct`** will *not* be considered critical.

GC3Pie uses standard command line utilities to interact with the resource manager. By default these commands are searched using the `PATH` environment variable, but you can specify the full path of these commands and/or add some extra options. The following options are used by the SGE backend:

- `qsub`: submit a job.
- `qacct`: get info on resources used by a job.
- `qdel`: cancel a job.
- `qstat`: get the status of a job or the status of available resources.

If `transport` is `ssh`, then the following options are also read and take precedence above the corresponding options set in the "auth" section:

- `port`: TCP port number where the SSH server is listening.
- `keyfile`: path to the (private) key file to use for SSH public key authentication.
- `ssh_config`: path to a SSH configuration file, where to read additional options from. The format of the SSH configuration file is documented in the ssh_config(5) man page.
- `ssh_timeout`: maximum amount of time (in seconds) that GC3Pie will wait for the SSH connection to be established.

Note: We advise you to use the SSH config file for setting port, key file, and connection timeout. Options `port`, `keyfile`, and `timeout` could be deprecated in future releases.

`pbs` resources (TORQUE and PBSPro batch-queueing systems)

The following configuration keys are required in a `pbs`-type resource section:

- `transport`: Possible values are: `ssh` or `local`. If `ssh`, GC3Pie tries to connect to the host specified in `frontend` via SSH in order to execute Troque/PBS commands. If `local`, the TORQUE/PBSPro commands are run directly on the machine where GC3Pie is installed.

- `frontend`: should contain the FQDN of the TORQUE/PBSPro front-end node. This configuration item is only relevant if `transport` is `local`. An SSH connection will be attempted to this node, in order to submit jobs and retrieve status info.

GC3Pie uses standard command line utilities to interact with the resource manager. By default these commands are searched using the `PATH` environment variable, but you can specify the full path of these commands and/or add some extra options. The following options are used by the PBS backend:

- `queue`: the name of the queue to which jobs are submitted. If empty (the default), no queue will be specified during submission, using the resource manager's default.
- `qsub`: submit a job.
- `qdel`: cancel a job.
- `qstat`: get the status of a job or the status of available resources.
- `tracejob`: get info on resources used by a job.

If `transport` is `ssh`, then the following options are also read and take precedence above the corresponding options set in the "auth" section:

- `port`: TCP port number where the SSH server is listening.
- `keyfile`: path to the (private) key file to use for SSH public key authentication.
- `ssh_config`: path to a SSH configuration file, where to read additional options from. The format of the SSH configuration file is documented in the ssh_config(5) man page.
- `ssh_timeout`: maximum amount of time (in seconds) that GC3Pie will wait for the SSH connection to be established.

Note: We advise you to use the SSH config file for setting port, key file, and connection timeout. Options `port`, `keyfile`, and `timeout` could be deprecated in future releases.

`lsf` resources (IBM LSF)

The following configuration keys are required in a `lsf`-type resource section:

- `transport`: Possible values are: `ssh` or `local`. If `ssh`, GC3Pie tries to connect to the host specified in `frontend` via SSH in order to execute LSF commands. If `local`, the LSF commands are run directly on the machine where GC3Pie is installed.
- `frontend`: should contain the FQDN of the LSF front-end node. This configuration item is only relevant if `transport` is `local`. An SSH connection will be attempted to this node, in order to submit jobs and retrieve status info.

GC3Pie uses standard command line utilities to interact with the resource manager. By default these commands are searched using the `PATH` environment variable, but you can specify the full path of these commands and/or add some extra options. The following options are used by the LSF backend:

- `bsub`: submit a job.
- `bjobs`: get the status and resource usage of a job.
- `bkill`: cancel a job.
- `lshosts`: get info on available resources.

LSF commands use a weird formatting: lines longer than 79 characters are wrapped around, and the continuation line starts with a long run of spaces. The length of this run of whitespace seems to vary with LSF version; GC3Pie is normally able to auto-detect it, but there can be a few unlikely cases where it cannot. If this ever happens, the following configuration option is here to help:

- `lsf_continuation_line_prefix_length`: length (in characters) of the whitespace prefix of continuation lines in `bjobs` output. This setting is normally not needed.

If `transport` is `ssh`, then the following options are also read and take precedence above the corresponding options set in the "auth" section:

- `port`: TCP port number where the SSH server is listening.
- `keyfile`: path to the (private) key file to use for SSH public key authentication.
- `ssh_config`: path to a SSH configuration file, where to read additional options from. The format of the SSH configuration file is documented in the ssh_config(5) man page.
- `ssh_timeout`: maximum amount of time (in seconds) that GC3Pie will wait for the SSH connection to be established.

Note: We advise you to use the SSH config file for setting port, key file, and connection timeout. Options `port`, `keyfile`, and `timeout` could be deprecated in future releases.

`slurm` resources

The following configuration keys are required in a `slurm`-type resource section:

- `transport`: Possible values are: `ssh` or `local`. If `ssh`, GC3Pie tries to connect to the host specified in `frontend` via SSH in order to execute SLURM commands. If `local`, the SLURM commands are run directly on the machine where GC3Pie is installed.
- `frontend`: should contain the FQDN of the SLURM front-end node. This configuration item is only relevant if `transport` is `ssh`. An SSH connection will be attempted to this node, in order to submit jobs and retrieve status info.

GC3Pie uses standard command line utilities to interact with the resource manager. By default these commands are searched using the `PATH` environment variable, but you can specify the full path of these commands and/or add some extra options. The following options are used by the SLURM backend:

- `sbatch`: submit a job; can specify additional arguments (they will be inserted between the `sbatch` invocation and the GC3Pie-provided options)
- `srun`: run a job's payload; can specify additional arguments (they will be inserted between the `srun` invocation and the GC3Pie-provided options)
- `scancel`: cancel a job.
- `squeue`: get the status of a job or of the available resources.
- `sacct`: get info on resources used by a job.

If `transport` is `ssh`, then the following options are also read and take precedence above the corresponding options set in the "auth" section:

- `port`: TCP port number where the SSH server is listening.
- `keyfile`: path to the (private) key file to use for SSH public key authentication.

- `ssh_config`: path to a SSH configuration file, where to read additional options from. The format of the SSH configuration file is documented in the ssh_config(5) man page.
- `ssh_timeout`: maximum amount of time (in seconds) that GC3Pie will wait for the SSH connection to be established.

Note: We advise you to use the SSH config file for setting port, key file, and connection timeout. Options `port`, `keyfile`, and `timeout` could be deprecated in future releases.

`shellcmd` resources

The following optional configuration keys are available in a `shellcmd`-type resource section:

- `transport`: Like any other resources, possible values are `ssh` or `local`. Default value is `local`.
- `frontend`: If *transport* is *ssh*, then *frontend* is the FQDN of the remote machine where the jobs will be executed.
- `time_cmd`: `ShellcmdLrms` needs the GNU implementation of the command *time* in order to get resource usage of the submitted jobs. `time_cmd` must contains the path to the binary file if this is different from the standard (`/usr/bin/time`).
- `override`: `ShellcmdLrms` by default will try to gather information on the system the resource is running on, including the number of cores and the available memory. These values may be different from the values stored in the configuration file. If `override` is *True*, then the values automatically discovered will be used. If `override` is *False*, the values in the configuration file will be used regardless of the real values discovered by the resource.
- `spooldir`: Root path to a filesystem location where to create temporary working directories for processes executed through this backend. GC3Pie will create dedicated temporary working directories, one for each job, within this root folder.

  By default, working directories are created as subdirectories of `$HOME/.gc3pie_jobs`.

If `transport` is `ssh`, then the following options are also read and take precedence above the corresponding options set in the "auth" section:

- `port`: TCP port number where the SSH server is listening.
- `keyfile`: path to the (private) key file to use for SSH public key authentication.
- `ssh_config`: path to a SSH configuration file, where to read additional options from. The format of the SSH configuration file is documented in the ssh_config(5) man page.
- `ssh_timeout`: maximum amount of time (in seconds) that GC3Pie will wait for the SSH connection to be established.

Note: We advise you to use the SSH config file for setting port, key file, and connection timeout. Options `port`, `keyfile`, and `timeout` could be deprecated in future releases.

`ec2+shellcmd` resource

The following configuration options are available for a resource of type `ec2+shellcmd`. If these options are omitted, then the default of the boto python library will be used, which at the time of writing means use the default region on Amazon.

- `ec2_url`: The URL of the EC2 frontend. On Amazon's AWS this is something like `https://ec2.us-east-1.amazonaws.com` (this is valid for the zone `us-east-1` of course). If no value is specified, the environment variable `EC2_URL` will be used, and if not found an error is raised.
- `ec2_region`: the region you want to access to.
- `keypair_name`: the name of the keypair to use when creating a new instance on the cloud. If it's not found, a new keypair with this name and the key stored in `public_key` will be used. Please note that if the keypair exists already on the cloud but the associated public key is different from the one stored in `public_key`, then an error is raised and the resource will not be used.
- `public_key`: public key to use when creating the keypair.

Note: GC3Pie assumes that the corresponding private key is stored on a file with the same path but without the `.pub` extension. This private key is necessary in order to access the virtual machines created on the cloud.

Note: **For Amazon AWS users**: Please note that AWS EC2 does not accept *DSA* keys; use RSA keys only for AWS resources.

- `vm_auth`: the name of a valid `auth` stanza used to connect to the virtual machine.
- `instance_type`: the instance type (aka *flavor*, aka *size*) you want to use for your virtual machines by default.
- `<application>_instance_type`: you can override the default instance type for a specific application by defining an entry in the configuration file for that application. For example:

```
instance_type=m1.tiny
gc_gps_instance_type=m1.large
```

  will use instance type `m1.large` for the `gc_gps` GC3Pie application, and `m1.tiny` for all the other applications.

- `image_id`: the **ami-id** of the image you want to use.
- `<application>_image_id`: override the generic `image_id` for a specific application.

  For example:

```
image_id=ami-00000048
gc_gps_image_id=ami-0000002a
```

  will make GC3Pie use the image `ami-0000002a` when running `gc_gps`, and image `ami-00000048` for all other applications.

- `security_group_name`: name of the security group to associate with VMs started by GC3Pie.

  If the named security group cannot be found, it will be created using the rules found in `security_group_rules`. If the security group is found but some of the rules in `security_group_rules` are not present, they will be added to the security groups. Additional rules, which are listed in the EC2 console but not included in `security_group_rules`, will *not* be removed from the security group.

- `security_group_rules`: comma separated list of security rules the `security_group` must have.

  Each rule has the form:

```
PROTOCOL:PORT_RANGE_START:PORT_RANGE_END:IP_NETWORK
```

where:

- `PROTOCOL` is one of `tcp`, `udp`, `icmp`;
- `PORT_RANGE_START` and `PORT_RANGE_END` are integers and define the range of ports to allow. If `PROTOCOL` is `icmp` please use `-1` for both values since in `icmp` there is no concept of *port*.
- `IP_NETWORK` is a range of IP to allow in the form `A.B.C.D/N`.

For instance, to allow SSH access to the virtual machine from *any* machine in the internet you can use:

```
security_group_rules = tcp:22:22:0.0.0.0/0
```

Note: In order to be able to access the virtual machines it created, GC3Pie **needs** to be able to connect via SSH, so a rule like the above is probably necessary in any GC3Pie configuration. For better security, it is wise to only allow the IP address or the range of IP addresses in use at your institution.

- `vm_pool_max_size`: the maximum number of Virtual Machine GC3Pie will start on this cloud. If *0* then there is no predefined limit to the number of virtual machines that GC3Pie can spawn.
- `user_data`: the *content* of a script that will run after the startup of the machine. For instance, to automatically upgrade a ubuntu machine after startup you can use:

```
user_data=#!/bin/bash
  aptitude -y update
  aptitude -y safe-upgrade
```

Note: When entering multi-line scripts, lines after the first one (where `user_data=` is) *must* be indented, i.e., begin with one or more spaces.

- `<application>_user_data`: override the generic `user_data` for a specific application.

  For example:

```
# user_data=
warholize_user_data = #!/bin/bash
  aptitude -y update && aptitude -y install imagemagick
```

  will install the `imagemagick` package only for VMs meant to run the `warholize` application.

Example `resource` sections

Example 1. This configuration stanza defines a resource to submit jobs to the Grid Engine cluster whose front-end host is `ocikbpra.uzh.ch`:

```
[resource/ocikbpra]
# A single SGE cluster, accessed by SSH'ing to the front-end node
type = sge
auth = <auth_name> # pick an ``ssh`` type auth, e.g., "ssh1"
transport = ssh
frontend = ocikbpra.uzh.ch
gamess_location = /share/apps/gamess
max_cores_per_job = 80
max_memory_per_core = 2
```

(continues on next page)

(continued from previous page)

```
max_walltime = 2
ncores = 80
```

Example 2. This configuration stanza defines a resource to submit jobs on virtual machines that will be automatically started by GC3Pie on Hobbes, the private OpenStack cloud of the University of Zurich:

```
[resource/hobbes]
enabled=yes
type=ec2+shellcmd
ec2_url=http://hobbes.gc3.uzh.ch:8773/services/Cloud
ec2_region=nova

auth=ec2hobbes
# These values my be overwritten by the remote resource
max_cores_per_job = 8
max_memory_per_core = 2
max_walltime = 8
max_cores = 32
architecture = x86_64

keypair_name=my_name
# If keypair does not exists, a new one will be created starting from
# `public_key`. Note that if the desired keypair exists, a check is
# done on its fingerprint and a warning is issued if it does not match
# with the one in `public_key`
public_key=~/.ssh/id_dsa.pub
vm_auth=gc3user_ssh
instance_type=m1.tiny
warholize_instance_type = m1.small
image_id=ami-00000048
warholize_image_id=ami-00000035
security_group_name=gc3pie_ssh
security_group_rules=tcp:22:22:0.0.0.0/0, icmp:-1:-1:0.0.0.0/0
vm_pool_max_size = 8
user_data=
warholize_user_data = #!/bin/bash
    aptitude update && aptitude install -u imagemagick
```

Enabling/disabling selected resources

Any resource can be disabled by adding a line `enabled = false` to its configuration stanza. Conversely, a line `enabled = true` will undo the effect of an `enabled = false` line (possibly found in a different configuration file).

This way, resources can be temporarily disabled (e.g., the cluster is down for maintenance) without having to remove them from the configuration file.

You can selectively disable or enable resources that are defined in the system-wide configuration file. Two main use cases are supported: the system-wide configuration file :file:`/etc/gc3/gc3pie.conf` lists and enables all available resources, and users can turn them off in their private configuration file :file:`~/.gc3/gc3pie.conf`; or the system-wide configuration can list all available resources but keep them disabled, and users can enable those they prefer in the private configuration file.

Environment Variables

The following environmental variables affect GC3Pie operations.

GC3PIE_CONF

Path to an alternate configuration file, that is read upon initialization of GC3Pie. If defined, this file is read *instead* of the default `$HOME/.gc3/gc3pie.conf`; if undefined or empty, the usual configuration file is loaded.

If this variable is defined, the logging configuration file is looked for in the same directory as the `gc3pie.conf` file, falling back to `$HOME/.gc3/gc3pie.log.conf` if not found there.

GC3PIE_ID_FILE

Path to the a shared state file, used for recording the "next available" job ID number. By default, it is located at `~/.gc3/next_id.txt`:file:.

GC3PIE_NO_CATCH_ERRORS

Comma-separated list of unexpected/generic error patterns upon which GC3Pie will not act (by default, ignoring them). Each of these "unignored" errors will be propagated all the way up to top-level. This facilitates running GC3Pie scripts in a debugger and inspecting the code when some unexpected error condition happens.

You can specify which errors to "unignore" by:

- Error class name (e.g., `InputFileError`). Note that this must be the *exact* class name of the error: GC3Pie will not walk the error class hierarchy for matches.
- Function/class/module name: all errors handled in the specified function/class/module will be propagated to the caller.
- Additional keywords describing the error. Please have a look at the source code for these keywords.

GC3PIE_RESOURCE_INIT_ERRORS_ARE_FATAL

If this environmental variable is set to `yes` or `1`, GC3Pie will abort operations immediately if a configured resource cannot be initialized. The default behavior is instead to ignore initialization errors and only abort if *no* resources can be initialized.

GC3Pie usage tutorials

The following slides provide an overview of GC3Pie features and usage from a users' perspective. They are used in the GC3Pie for users training held regularly at the University of Zurich. (Thus, they may contain references to local infrastructure or systems but should be comprehensible and -hopefully- useful for a more general audience as well.)

Introduction to GC3Pie

Introduction to the software: what is GC3Pie, what is it for, and an overview of its features for writing high-throughput computing scripts.

Introduction to GC3Pie session-based scripts

An overview of the features of GC3Pie's *session-based scripts* and the associated command-line utilities.

The GC3Apps software

GC3Apps is a collection command line front-end to manage submission of a (potentially) large number of computational *job* to different batch processing systems. For example, the GC3Apps commands **`ggamess`** can run GAMESS jobs on the SMSCG infrastructure and on any computational cluster you can `ssh`:command: into.

This chapter is a tutorial for the GC3Apps command-line scripts: it explains the common concepts and features, then goes on to describe the specifics of each command in larger detail.

All GC3Apps scripts share a common set of functionalities, which are derive from a common blueprint, named a *session-based script*, described in Section *Introduction to session-based scripts* below. Script-specific sections detail the scope and options that are unique to a given script.

If you find a technical term whose meaning is not clear to you, please look it up in the *Glossary*. (But feel free to ask on the GC3Pie mailing list if it's still unclear!)

Introduction to *session-based* scripts

All GC3Apps scripts derive their core functionality from a common blueprint, named a *session-based script*. The purpose of this section is to describe this common functionality; script-specific sections detail the scope and options that are unique to a given script. Readers interested in Python programming can find the complete documentation about the *session-based script API* in the `SessionBasedScript` section.

The functioning of GC3Apps scripts revolves around a so-called *session*. A *session* is just a named collection of jobs. For instance, you could group into a single session jobs that analyze a set of related files.

Each time it is run, a GC3Apps script performs the following steps:

1. Reads the session directory and loads all stored jobs into memory. If the session directory does not exist, one will be created with empty contents.
2. Scans the command-line input arguments: if existing jobs do not suffice to analyze the input data, new jobs are added to the session.
3. The status of all existing jobs is updated, output from finished jobs is collected, and new jobs are submitted.

   Finally, a summary table of all known jobs is printed. (To control the amount of printed information, see the `-l` command-line option below.)
4. If the `-C` command-line option was given (see below), waits the specified amount of seconds, and then goes back to step 3.

   Execution can be interrupted at any time by pressing `Ctrl+C`.

Basic command-line usage and options

The exact command-line usage of *session-based scripts* varies from one script to the other, so please consult the documentation page for your application. There are quite a number of common options and behaviors, however, which are described here.

Continuous execution

While single-pass execution of a GC3Apps script is possible (and sometimes used), it is much more common to keep the script running and let it manage jobs until all are finished. This is accomplished with the following command-line option:

-C NUM, --continuous NUM Keep running, monitoring jobs and possibly submitting new ones or fetching results every NUM seconds.

When all jobs are finished, a GC3Apps script exits even if the `-C` option is given.

Verbose listing of jobs

Only a summary of job states is printed by default at the end of step 3., together with the count of jobs that are in the specified state. Use the `-l` option (see below) to get a detailed listing of all jobs.

-l STATE, --state STATE Print a table of jobs including their status.

The *STATE* argument restricts output to jobs in that particular state. It can be a single *state* word (e.g., `RUNNING`) or a comma-separated list thereof (e.g., `NEW,SUBMITTED,RUNNING`).

The pseudo-states `ok` and `failed` are also allowed for selecting jobs in `TERMINATED` state with exit code (respectively) 0 or nonzero.

If *STATE* is omitted, no restriction is placed on job states, and a table of *all* jobs is printed.

Maximum number of concurrent jobs

There is a maximum number of jobs that can be in `SUBMITTED` or `RUNNING` state at a given time. GC3Apps scripts will delay submission of newly-created jobs so that this limit is never exceeded. The default limit is 50, but it can be changed with the following command-line option:

-J NUM, --max-running NUM Set the maximum NUMber of jobs (default: 50) in `SUBMITTED` or `RUNNING` state.

Location of output files

By default, output files are placed in the same directory where the corresponding input file resides. This can be changed with the following option; it is also possible to specify output locations that vary depending on certain job features.

-o DIRECTORY, --output DIRECTORY Output files from all jobs will be collected in the specified *DIRECTORY* path. If the destination directory does not exist, it is created.

Job control options

These command-line options control the requirements and constraints of *new jobs*. Indeed, note that changing the arguments to these options *does not* change the corresponding requirements on jobs that already exist in the session.

-c NUM, --cpu-cores NUM Set the number of CPU cores required for each job (default: 1). *NUM* must be a whole number.

-m GIGABYTES, --memory-per-core GIGABYTES Set the amount of memory required per execution core; (Default: 2GB). Specify this as an integral number followed by a unit, e.g. '512MB' or '4GB'. Valid unit names are: 'B', 'GB', 'GiB', 'KiB', 'MB', 'MiB', 'PB', 'PiB', 'TB', 'TiB', 'kB'.

-r NAME, --resource NAME Submit jobs to a specific *resource*. `NAME` is a reource name or comma-separated list of resource names. Use the command *gservers* to list available resources.

-w DURATION, --wall-clock-time DURATION Set the time limit for each job; default is '8 hours'. Jobs exceeding this limit will be stopped and considered as 'failed'. The duration can be expressed as a whole number followed by a time unit, e.g., '3600 s', '60 minutes', '8 hours', or a combination thereof, e.g., '2hours 30minutes'. Valid unit names are: 'd', 'day', 'days', 'h', 'hour', 'hours', 'hr', 'hrs', 'm', 'microsec', 'microseconds', 'min', 'mins', 'minute', 'minutes', 'ms', 'nanosec', 'nanoseconds', 'ns', 's', 'sec', 'second', 'seconds', 'secs'.

Session control options

This set of options control the placement and contents of the *session*.

-s PATH, --session PATH Store the session information in the directory at *PATH*. (By default, this is a subdirectory of the current directory, named after the script you are executing.)

If *PATH* is an existing directory, it will be used for storing job information, and an index file (with suffix `.csv`) will be created in it. Otherwise, the job information will be stored in a directory named after *PATH* with a suffix `.jobs` appended, and the index file will be named after *PATH* with a suffix `.csv` added.

-N, --new-session Discard any information saved in the session directory (see the `--session` option) and start a new session afresh. Any information about jobs previously recorded in the session is lost.

-u, --store-url URL Store GC3Pie job information at the persistent storage specified by *URL*. The *URL* can be any form that is understood by the `gc3libs.persistence.make_store()` function (which see for details). A few examples:

- `sqlite` – the jobs are stored in a SQLite3 database named `jobs.db` and contained in the session directory.
- */path/to/a/directory* – the jobs are stored in the given directory, one file per job (this is the default format used by GC3Pie)
- `sqlite:////path/to/a/file.db` – the jobs are stored in the given SQLite3 database file.
- `mysql://user,passwd@server/dbname` – jobs are stored in table `store` of the specified MySQL database. The DB server and connection credentials (username, password) are also part of the *URL*.

If this option is omitted, GC3Pie's *SessionBasedScript* defaults to storing jobs in the subdirectory `jobs` of the session directory; each job is saved in a separate file.

Exit code

A GC3Apps script exits when all jobs are finished, when some error occurred that prevented the script from completing, or when a user interrupts it with `Ctrl+C`

In any case, the exit code of GC3Apps scripts tracks job status (in the following sense). The exitcode is a bitfield; the 4 least-significant bits are assigned a meaning according to the following table:

Bit	Meaning
0	Set if a fatal error occurred: the script could not complete
1	Set if there are jobs in `FAILED` state
2	Set if there are jobs in `RUNNING` or `SUBMITTED` state
3	Set if there are jobs in `NEW` state

This boils down to the following rules:

- exitcode is 0: all jobs are *DONE*, no further action will be taken by the script (which exists immediately if called again on the same session).
- exitcode is 1: an error interrupted the script execution.
- exitcode is 2: all jobs finished, but some are in *FAILED* state.
- exitcode > 3: run the script again to make jobs progress.

The ggamess script

GC3Apps provide a script drive execution of multiple `gamess` jobs each of them with a different input file. It uses the generic *gc3libs.cmdline.SessionBasedScript* framework.

The purpose of GAMESS is to execute *several concurrent runs* of GAMESS each with separate input file. These runs are performed in parallel using every available GC3Pie parameters.

How to run GAMESS on the Grid

SSH to *ocikbgtw*, then run the command (it's one single command line, even if it appears broken in several ones in the mail):

```
ggamess.py -A ~/beckya-dmulti.changes.tar.gz -R 2011R3-beckya-dmulti -s "a_session_
↪name" "input_files_or_directories"
```

The parts in double quotes should be replaced with actual content:

`a_session_name`:

> Used for grouping. This is a word of your choosing (e.g., "*test1*", "*control_group*"), used as a label to tag a group of analyses. Multiple concurrent sessions can exist, and they won't interfere one with the other. Again, note that a single session can run many different *.inp* files.

`input_files_or_directories`:

> This part consists in the path name of *.inp* files or a directory containing *.inp* files. When a directory is specified, all the *.inp* files contained in it are submitted as GAMESS jobs.

After running, the program will print a short summary of the session (how many jobs running, how many queued, how many finished). Each finished job creates one directory (whose name is equal to the name of the input file, minus the trailing *.inp*), which contains the *.out* and *.dat* files.

For shorter typing, I have defined an alias *ggms* to expand to the above string `ggamess.py -A ... 2011R3-beckya-dmulti`, so you could shorten the command to just:

```
ggms -s "a_session_name" "input_files_or_directories"
```

For instance, to use `ggames.py` to analyse a single *.inp* file you must run:

```
ggms -s "single" dmulti/inp/neutral/dmulti_cc4l.inp
```

while to use `ggamess.py` to run several GAMESS jobs in parallel:

```
ggms -s "multiple" dmulti/inp/neutral
```

Tweaking execution

Command-line options (those that start with a dash character '-') can be used to alter the behavior of the `ggamess.py` command:

`-A` *filename.changes.tar.gz*

This selects the file containing your customized version of GAMESS in a format suitable for running in a virtual machine on the Grid. This file should be created following the procedure detailed below.

`-R` *version*

Select a specific version of GAMESS. This should have been installed in the virtual machine within a directory named `gamess-version`; for example, your modified GAMESS is saved in directory `gamess-2011R3-beckya-dmulti` so the "*version*" string is `2011R3-beckya-dmulti`.

If you omit the `-R` "*version*" part, you get the default GAMESS which is presently 2011R1.

`-s` *session*

Group jobs in a named session; see above.

`-w` *NUM*

Request a running time of at *NUM* hours. If you omit this part, the default is 8 hours.

`-m` *NUM*

Request *NUM* Gigabytes of memory for running each job. GAMESS' memory is measured in words, and each word is 8 bytes; add 1 GB to the total to be safe :-)

Updating the GAMESS code

For this you will need to launch the AppPot virtual machine, which is done by running the following command at the command prompt on *ocikbgtw*:

```
apppot-start.sh
```

After a few seconds, you should find yourself at the same `user@rootstrap` prompt that you get on your VirtualBox instance, so you can use the same commands etc.

The only difference of note is that you can exchange files between the AppPot virtual machine and *ocikbgtw* via the *job* directory (whereas it's `/scratch` in VirtualBox). So: files you copy into *job* in the AppPot VM will appear into your home directory on *ocikbgtw*, and conversely files from your home directory on *ocikbgtw* can be read/written as if they were into directory *job* in the AppPot VM.

Once you have compiled a new version of GAMESS that you wish to test, you need to run this command (at the `user@rootstrap` command prompt in the AppPot VM):

```
sudo apppot-snap changes ~/job/beckya-dmulti.changes.tar.gz
```

This will overwrite the file `beckya-dmulti.changes.tar.gz` with the new GAMESS version. If you don't want to overwrite it and instead create another one, just change the filename above (but it *has to* end with the string `.changes.tar.gz`), and the use the new name for the `-R` option to ggamess.py

Exit the AppPot VM by typing `exit` at the command prompt.

The `ggeotop` script

GC3Apps provide a script drive execution of multiple `GEOtop` jobs. It uses the generic *gc3libs.cmdline.SessionBasedScript* framework.

From GEOtop's "read me" file:

```
#
# RUNNING
# Run this simulation by calling the executable (GEOtop_1.223_static)
# and giving the simulation directory as an argument.
#
# EXAMPLE
# ls2:/group/geotop/sim/tmp/000001>./GEOtop_1.223_static ./
#
# TERMINATION OF SIMULATION BY GEOTOP
# When GEOtop terminates due to an internal error, it mostly reports this
# by writing a corresponding file (_FAILED_RUN or _FAILED_RUN.old) in the
# simulation directory. When is terminates sucessfully this file is
# named (_SUCCESSFUL_RUN or _SUCCESSFUL_RUN.old).
#
# RESTARTING SIMULATIONS THAT WERE TERMINATED BY THE SERVER
# When a simulation is started again with the same arguments as described
# above (RUNNING), then it continues from the last saving point. If
# GEOtop finds a file indicating a successful/failed run, it terminates.
```

Introduction

`ggeotop` driver script acan the specified INPUT directories recursively for simulation directories and submit a job for each one found; job progress is monitored and, when a job is done, its output files are retrieved back into the simulation directory itself.

A simulation directory is defined as a directory containing a `geotop.inpts` file, an `in` and an `out` folders.

The `ggeotop` command keeps a record of jobs (submitted, executed and pending) in a session file (set name with the `-s` option); at each invocation of the command, the status of all recorded jobs is updated, output from finished jobs is collected, and a summary table of all known jobs is printed. New jobs are added to the session if new input files are added to the command line.

Options can specify a maximum number of jobs that should be in 'SUBMITTED' or 'RUNNING' state; `ggeotop` will delay submission of newly-created jobs so that this limit is never exceeded.

Options can specify a maximum number of jobs that should be in 'SUBMITTED' or 'RUNNING' state; **`ggeotop`** will delay submission of newly-created jobs so that this limit is never exceeded.

In more detail, **`ggeotop`** does the following:

1. Reads the *session* (specified on the command line with the `--session` option) and loads all stored jobs into memory. If the session directory does not exist, one will be created with empty contents.

2. Recursively scans trough `input` folder searching for any valid folder.

   ggeotop will generate a collection of jobs one for each valid input folder. Each job will transfer the input folder to the remote execution node and run `GEOTop`. `GEOTop` reads geotop.inpts files for getting instructions on how to find the input data, what and how to process and where to place generated output results. Extracted from a generic geotop.inpts file:

```
DemFile = "in/dem"
MeteoFile = "in/meteo"
SkyViewFactorMapFile = "in/svf"
SlopeMapFile = "in/slp"
AspectMapFile = "in/asp"

!===========================================
!   DIST OUTPUT
!===========================================
SoilAveragedTempTensorFile = "out/maps/T"
NetShortwaveRadiationMapFile="out/maps/SWnet"
InShortwaveRadiationMapFile="out/maps/SWin"
InLongwaveRadiationMapFile="out/maps/LWin"
SWEMapFile= "out/maps/SWE"
AirTempMapFile = "out/maps/Ta"
```

3. Updates the state of all existing jobs, collects output from finished jobs, and submits new jobs generated in step 2.

4. For each of the terminated jobs, a post-process routine is executed to check and validate the consistency of the generated output. If no `_SUCCESSFUL_RUN` or `_FAILED_RUN` file is found, the related job will be resubmitted together with the current input and output folders. GEOTop is capable of restarting an interrupted claculation by inspecting the intermediate results generated in `out` folder.

   Finally, a summary table of all known jobs is printed. (To control the amount of printed information, see the `-l` command-line option in the *Introduction to session-based scripts* section.)

4. If the `-C` command-line option was given (see below), waits the specified amount of seconds, and then goes back to step 3.

   The program **ggeotop** exits when all jobs have run to completion, i.e., when all valid input folders have been computed.

Execution can be interrupted at any time by pressing `Ctrl+C`. If the execution has been interrupted, it can be resumed at a later stage by calling **ggeotop** with exactly the same command-line options.

Command-line invocation of `ggeotop`

The **ggeotop** script is based on GC3Pie's *session-based script* model; please read also the *Introduction to session-based scripts* section for an introduction to sessions and generic command-line options.

A **ggeotop** command-line is constructed as follows:

1. Each argument (at least one should be specified) is considered as a folder reference.

2. `-x` option is used to specify the path to the GEOtop executable file.

Example 1. The following command-line invocation uses **ggeotop** to run `GEOTop` on all valid input folder found in the recursive check of `input_folder`:

```
$ ggeotop -x /apps/geotop/bin/geotop_1_224_20120227_static ./input_folder
```

Example 2.

```
$ ggeotop --session SAMPLE_SESSION -w 24 -x /apps/geotop/bin/geotop_1_224_20120227_
↪static ./input_folder
```

In this example, job information is stored into session `SAMPLE_SESSION` (see the documentation of the `--session` option in *Introduction to session-based scripts*). The command above creates the jobs, submits them, and finally prints the following status report:

```
Status of jobs in the 'SAMPLE_SESSION' session: (at 10:53:46, 02/28/12)
NEW   0/50    (0.0%)
RUNNING   0/50    (0.0%)
STOPPED   0/50    (0.0%)
SUBMITTED   50/50   (100.0%)
TERMINATED   0/50    (0.0%)
TERMINATING   0/50    (0.0%)
total   50/50   (100.0%)
```

Calling **ggeotop** over and over again will result in the same jobs being monitored;

The `-C` option tells **ggeotop** to continue running until all jobs have finished running and the output files have been correctly retrieved. On successful completion, the command given in *example 2.* above, would print:

```
Status of jobs in the 'SAMPLE_SESSION' session: (at 11:05:50, 02/28/12)
NEW   0/50    (0.0%)
RUNNING   0/50    (0.0%)
STOPPED   0/540    (0.0%)
SUBMITTED   0/50    (0.0%)
TERMINATED   50/50   (100.0%)
TERMINATING   0/50    (0.0%)
ok   50/50   (100.0%)
total   50/50   (100.0%)
```

Each job will be named after the folder name (e.g. 000002) (you could see this by passing the `-l` option to **ggeotop**).; each of these jobs will fill the related input folder with the produced outputs.

For each job, the set of output files is automatically retrieved and placed in the locations described below.

Output files for `ggeotop`

Upon successful completion, the output directory of each **ggeotop** job contains:

- the `out` folder will contains what has been produced during the computation of the related job.

Example usage

This section contains commented example sessions with **ggeotop**.

Manage a set of jobs from start to end

In typical operation, one calls **ggeotop** with the `-C` option and lets it manage a set of jobs until completion.

So, to analyse all valid folders under `input_folder`, submitting 200 jobs simultaneously each of them requesting 2GB of memory and 8 hours of *wall-clock time*, one can use the following command-line invocation:

```
$ ggeotop -s example -C 120 -x
/apps/geotop/bin/geotop_1_224_20120227_static -w 8 input_folder
```

The `-s example` option tells **ggeotop** to store information about the computational jobs in the `example.jobs` directory.

The `-C 120` option tells **ggeotop** to update job state every 120 seconds; output from finished jobs is retrieved and new jobs are submitted at the same interval.

The above command will start by printing a status report like the following:

```
Status of jobs in the 'example.csv' session:
SUBMITTED   1/1 (100.0%)
```

It will continue printing an updated status report every 120 seconds until the requested parameter range has been computed.

In GC3Pie terminology when a job is finished and its output has been successfully retrieved, the job is marked as `TERMINATED`:

```
Status of jobs in the 'example.csv' session:
TERMINATED   1/1 (100.0%)
```

Using GC3Pie utilities

GC3Pie comes with a set of generic utilities that could be used as a complemet to the **ggeotop** command to better manage a entire session execution.

`gkill`: cancel a running job

To cancel a running job, you can use the command **gkill**. For instance, to cancel *job.16*, you would type the following command into the terminal:

```
gkill job.16
```

or:

```
gkill -s example job.16
```

gkill could also be used to cancel jobs in a given state

```
gkill -s example -l UNKNOWN
```

Warning: *There's no way to undo a cancel operation!* Once you have issued a **gkill** command, the job is deleted and it cannot be resumed. (You can still re-submit it with **gresub**, though.)

`ginfo`: accessing low-level details of a job

It is sometimes necessary, for debugging purposes, to print out all the details about a job; the **`ginfo`** command does just that: prints all the details that GC3Utils know about a single job.

For instance, to print out detailed information about *job.13* in session *example*, you would type

```
ginfo -s example job.13
```

For a job in `RUNNING` or `SUBMITTED` state, only little information is known: basically, where the job is running, and when it was started:

```
$ ginfo -s example job.13
job.13
    cores: 2
    execution_targets: hera.wsl.ch
    log:
        SUBMITTED at Tue May 15 09:52:05 2012
        Submitted to 'wsl' at Tue May 15 09:52:05 2012
        RUNNING at Tue May 15 10:07:39 2012
    lrms_jobid: gsiftp://hera.wsl.ch:2811/jobs/116613370683251353308673
    lrms_jobname: GC3Pie_00002
    original_exitcode: -1
    queue: smscg.q
    resource_name: wsl
    state_last_changed: 1337069259.18
    stderr_filename: ggeotop.log
    stdout_filename: ggeotop.log
    timestamp:
        RUNNING: 1337069259.18
        SUBMITTED: 1337068325.26
    unknown_iteration: 0
    used_cputime: 1380
    used_memory: 3382706
```

If you omit the job number, information about *all* jobs in the session will be printed.

Most of the output is only useful if you are familiar with GC3Utils inner working. Nonetheless, **`ginfo`** output is definitely something you should include in any report about a misbehaving job!

For a finished job, the information is more complete and can include error messages in case the job has failed:

```
$ ginfo -c -s example job.13
job.13
    _arc0_state_last_checked: 1337069259.18
    _exitcode: 0
    _signal: None
    _state: TERMINATED
    cores: 2
    download_dir: /data/geotop/results/00002
    execution_targets: hera.wsl.ch
    log:
        SUBMITTED at Tue May 15 09:52:04 2012
        Submitted to 'wsl' at Tue May 15 09:52:04 2012
        TERMINATING at Tue May 15 10:07:39 2012
        Final output downloaded to '/data/geotop/results/00002'
        TERMINATED at Tue May 15 10:07:43 2012
    lrms_jobid: gsiftp://hera.wsl.ch:2811/jobs/11441337068324584585032
```

(continues on next page)

(continued from previous page)

```
    lrms_jobname: GC3Pie_00002
    original_exitcode: 0
    queue: smscg.q
    resource_name: wsl
    state_last_changed: 1337069263.13
    stderr_filename: ggeotop.log
    stdout_filename: ggeotop.log
    timestamp:
        SUBMITTED: 1337068324.87
        TERMINATED: 1337069263.13
        TERMINATING: 1337069259.18
    unknown_iteration: 0
    used_cputime: 360
    used_memory: 3366977
    used_walltime: 300
```

With option `-v`, **`ginfo`** output is even more verbose and complete, and includes information about the application itself, the input and output files, plus some backend-specific information

```
$ ginfo -c -s example job.13
job.13
  arguments: 00002
  changed: False
  environment:
  executable: geotop_static
  executables: geotop_static
  execution:
      _arc0_state_last_checked: 1337069259.18
      _exitcode: 0
      _signal: None
      _state: TERMINATED
      cores: 2
      download_dir: /data/geotop/results/00002
      execution_targets: hera.wsl.ch
      log:
          SUBMITTED at Tue May 15 09:52:04 2012
          Submitted to 'wsl' at Tue May 15 09:52:04 2012
          TERMINATING at Tue May 15 10:07:39 2012
          Final output downloaded to '/data/geotop/results/00002'
          TERMINATED at Tue May 15 10:07:43 2012
      lrms_jobid: gsiftp://hera.wsl.ch:2811/jobs/1144133706832458458503 2
      lrms_jobname: GC3Pie_00002
      original_exitcode: 0
      queue: smscg.q
      resource_name: wsl
      state_last_changed: 1337069263.13
      stderr_filename: ggeotop.log
      stdout_filename: ggeotop.log
      timestamp:
          SUBMITTED: 1337068324.87
          TERMINATED: 1337069263.13
          TERMINATING: 1337069259.18
      unknown_iteration: 0
      used_cputime: 360
      used_memory: 3366977
      used_walltime: 300
```

(continues on next page)

(continued from previous page)

```
  jobname: GC3Pie_00002
  join: True
  output_base_url: None
  output_dir: /data/geotop/results/00002
  outputs:
      @output.list: file, , @output.list, None, None, None, None
      ggeotop.log: file, , ggeotop.log, None, None, None, None
  persistent_id: job.1698503
  requested_architecture: x86_64
  requested_cores: 2
  requested_memory: 4
  requested_walltime: 4
  stderr: None
  stdin: None
  stdout: ggeotop.log
  tags: APPS/EARTH/GEOTOP
```

The `grosetta` and `gdocking` scripts

GC3Apps provide two scripts to drive execution of applications (*protocols*, in Rosetta terminology) from the Rosetta bioinformatics suite.

The purpose of **`grosetta`** and **`gdocking`** is to execute *several concurrent runs* of minirosetta or docking_protocol on a set of input files, and collect the generated output. These runs are performed in parallel using every available GC3Pie *resource*; you can of course control how many runs should be executed and select what output files you want from each one.

The script **`grosetta`** is a relatively generic front-end that executes the minirosetta program by default (but a different application can be chosen with the `-x` *command-line option*). The **`gdocking`** script is specialized for running Rosetta's docking_protocol program.

Introduction

The **`grosetta`** and **`gdocking`** execute *several runs* of minirosetta or docking_protocol on a set of input files, and collect the generated output. These runs are performed in parallel, up to a limit that can be configured with the `-J` *command-line option*. You can of course control how many runs should be executed and select what output files you want from each one.

Note: The **`grosetta`** and **`gdocking`** scripts are very similar in usage. In the following, whatever is written about **`grosetta`** applies to **`gdocking`** as well; the differences will be pointed out on a case-by-case basis.

In more detail, **`grosetta`** does the following:

1. Reads the *session* (specified on the command line with the `--session` option) and loads all stored jobs into memory. If the session directory does not exist, one will be created with empty contents.

2. Scans the input file names given on the command-line, and generates a number of identical computational jobs, all running the same Rosetta program on the same set of input files. The objective is to compute a specified number *P* of decoys of any given PDB file.

   The number *P* of wanted decoys can be set with the `--total-decoys` option (see below). The option `--decoys-per-job` can set the number of decoys that each computational job can compute; this should be

a guessed based on the maximum allowed run time of each job and the time taken by the Rosetta protocol to compute a single decoy.

3. Updates the state of all existing jobs, collects output from finished jobs, and submits new jobs generated in step 2.

   Finally, a summary table of all known jobs is printed. (To control the amount of printed information, see the `-l` command-line option in the *Introduction to session-based scripts* section.)

4. If the `-C` command-line option was given (see below), waits the specified amount of seconds, and then goes back to step 3.

   The program **grosetta** exits when all jobs have run to completion, i.e., when the wanted number of decoys have been computed.

   Execution can be interrupted at any time by pressing `Ctrl+C`. If the execution has been interrupted, it can be resumed at a later stage by calling **grosetta** with exactly the same command-line options.

The **gdocking** program works in exactly the same way, with the important exception that **gdocking** uses a separate Rosetta docking_protocol program invocation *per input file*.

Command-line invocation of `grosetta`

The **grosetta** script is based on GC3Pie's *session-based script* model; please read also the *Introduction to session-based scripts* section for an introduction to sessions and generic command-line options.

A **grosetta** command-line is constructed as follows:

1. The 1st argument is the *flags* file, containing options to pass to every executed Rosetta program;
2. then follows any number of input files (copied from your PC to the execution site);
3. then a literal colon character `:`;
4. finally, you can list any number of output file patterns (copied back from the execution site to your PC); wildcards (e.g., `*.pdb`) are allowed, but you must enclose them in quotes. Note that:
   - you can omit the output files: the default is `"*.pdb" "*.sc" "*.fasc"`
   - if you omit the output files patterns, omit the colon as well

Example 1. The following command-line invocation uses **grosetta** to run minirosetta on the molecule files `1bjpA.pdb`, `1ca7A.pdb`, and `1cgqA.pdb`. The `flags` file (1st command-line argument) is a text file containing options to pass to the actual minirosetta program. Additional input files are specified on the command line between the `flags` file and the PDB input files.

```
   $ grosetta flags alignment.filt query.fasta query.psipred_ss2 boinc_
↪aaquery03_05.200_v1_3.gz boinc_aaquery09_05.200_v1_3.gz 1bjpA.pdb 1ca7A.
↪pdb 1cgqA.pdb

You can see that the listing of output patterns has been omitted,
so `grosetta`:command: will use the default and retrieve all
`*.pdb`:file:, `*.sc`:file: and `*.fasc`:file: files.
```

There will be a number of *identical* jobs being executed as a result of a **grosetta** or **gdocking** invocation; this number depends on the ratio of the values given to options `-P` and `-p`:

-P NUM, --total-decoys NUM Compute *NUM* decoys per input file.

-p NUM, --decoys-per-job NUM Compute *NUM* decoys in a single job (default: 1). This parameter should be tuned so that the running time of a single job does not exceed the maximum wall-clock time (see the

`--wall-clock-time` command-line option in *Introduction to session-based scripts*).

If you omit `-P` and `-p`, they both default to 1, i.e., one job will be created (as in the *example 1.* above).

Example 2. The following command-line invocation will run 3 parallel instances of minirosetta, each of which generates 2 decoys (save the last one, which only generates 1 decoy) of the molecule described in file `1bjpA.pdb`:

```
$ grosetta --session SAMPLE_SESSION --total-decoys 5 --decoys-per-job 2␣
↪flags alignment.filt query.fasta query.psipred_ss2 boinc_aaquery03_05.200_
↪v1_3.gz boinc_aaquery09_05.200_v1_3.gz 1bjpA.pdb
```

In this example, job information is stored into session `SAMPLE_SESSION` (see the documentation of the `--session` option in *Introduction to session-based scripts*). The command above creates the jobs, submits them, and finally prints the following status report:

```
Status of jobs in the 'SAMPLE_SESSION' session: (at 10:53:46, 02/28/12)
        NEW    0/3     (0.0%)
    RUNNING    0/3     (0.0%)
    STOPPED    0/3     (0.0%)
  SUBMITTED    3/3   (100.0%)
 TERMINATED    0/3     (0.0%)
TERMINATING    0/3     (0.0%)
      total    3/3   (100.0%)
```

Note that the status report counts the number of *jobs in the session*, not the total number of decoys being generated. (Feel free to report this as a bug.)

Calling `grosetta` over and over again will result in the same jobs being monitored; to create new jobs, change the command line and raise the value for `-P` or `-p`. (To completely erase an existing session and start over, use the `--new-session` option, as per *session-based script* documentation.)

The `-C` option tells **`grosetta`** to continue running until all jobs have finished running and the output files have been correctly retrieved. On successful completion, the command given in *example 2.* above, would print:

```
Status of jobs in the 'SAMPLE_SESSION' session: (at 11:05:50, 02/28/12)
        NEW    0/3     (0.0%)
    RUNNING    0/3     (0.0%)
    STOPPED    0/3     (0.0%)
  SUBMITTED    0/3     (0.0%)
 TERMINATED    3/3   (100.0%)
TERMINATING    0/3     (0.0%)
         ok    3/3   (100.0%)
      total    3/3   (100.0%)
```

The three jobs are named `0--1`, `2--3` and `4--5` (you could see this by passing the `-l` option to **`grosetta`**); each of these jobs will create an output directory named after the job.

In general, **`grosetta`** jobs are named *N*`--`*M* with *N* and *M* being two integers from 0 up to the value specified with option `--total-decoys`. Jobs generated by **`gdocking`** are instead named after the input file, with a `.`*N*`--`*M* suffix added.

For each job, the set of output files is automatically retrieved and placed in the locations described below.

Note: The naming and contents of output files differ between **`grosetta`** and **`gdocking`**. Refer to the appropriate section below!

Output files for `grosetta`

Upon successful completion, the output directory of each **`grosetta`** job contains:

- A copy of the *input* PDB files;
- Additional `.pdb` files named `S_`*`random string`*`.pdb`, generated by minirosetta during its run;
- A file `score.sc`;
- Files `minirosetta.static.log`, `minirosetta.static.stdout.txt` and `minirosetta.static.stderr.txt`.

The `minirosetta.static.log` file contains the output log of the minirosetta execution. For each of the `S_*.pdb` files above, a line like the following should be present in the log file (the file name and number of elapsed seconds will of course vary!):

```
protocols.jd2.JobDistributor: S_1CA7A_1_0001 reported success in 124 seconds
```

The `minirosetta.static.stdout.txt` contains a copy of the minirosetta output log, plus the output of the wrapper script. In case of successful minirosetta run, the last line of this file will read:

```
minirosetta.static: All done, exitcode: 0
```

Output files for `gdocking`

Execution of `gdocking` yields the following output:

- For each `.pdb` input file, a `.decoys.tar` file (e.g., for `1bjpa.pdb` input, a `1bjpa.decoys.tar` output is produced), which contains the `.pdb` files of the decoys produced by **`gdocking`**.
- For each successful job, a *.N–M* directory: e.g., for the `1bjpa.1--2` job, a `1bjpa.1--2/` directory is created, with the following content:
  - `docking_protocol.log`: output of Rosetta's `docking_protocol` program;
  - `docking_protocol.stderr.txt`, `docking_protocol.stdout.txt`: obvoius meaning. The "stdout" file contains a copy of the `docking_protocol.log` contents, plus the output from the wrapper script.
  - `docking_protocol.tar.gz`: the `.pdb` decoy files produced by the job.

The following scheme summarizes the location of **`gdocking`** output files:

```
(directory where gdocking is run)/
  |
  +- file1.pdb      Original input file
  |
  +- file1.N--M/    Directory collecting job outputs from job file1.N--M
  |    |
  |    +- docking_protocol.tar.gz
  |    +- docking_protocol.log
  |    +- docking_protocol.stderr.txt
  |    ... etc
  |
  +- file1.N--M.fasc   FASC file for decoys N to M [1]
  |
  +- file1.decoys.tar  tar archive of PDB file of all decoys
  |                    generated corresponding to 'file1.pdb' [2]
```

(continues on next page)

(continued from previous page)

```
  |
  ...
```

Let *P* be the total number of decoys (the argument to the `-P` option), and *p* be the number of decoys per job (argument to the `-p` option). Then you would get in a single directory:

1. *(P/p)* different `.fasc` files, corresponding to the *(P/p)* jobs;
2. *P* different `.pdb` files, named `a_file.0.pdb` to `a_file.`*(P-1)*`.pdb`

Example usage

This section contains commented example sessions with **grosetta**. All the files used in this example are available in the GC3Pie Rosetta test directory (courtesy of Lars Malmstroem).

Manage a set of jobs from start to end

In typical operation, one calls **grosetta** with the `-C` option and lets it manage a set of jobs until completion.

So, to generate one decoy from a set of given input files, one can use the following command-line invocation:

```
$ grosetta -s example -C 120 -P 1 -p 1 \
    flags alignment.filt query.fasta \
    query.psipred_ss2 boinc_aaquery03_05.200_v1_3.gz \
    boinc_aaquery09_05.200_v1_3.gz 1bjpA.pdb 1ca7A.pdb \
    2fltA.pdb 2fm7A.pdb 2op8A.pdb 2ormA.pdb 2os5A.pdb \
    3c6vA.pdb
```

The `-s example` option tells **grosetta** to store information about the computational jobs in the `example.jobs` directory.

The `-C 120` option tells **grosetta** to update job state every 120 seconds; output from finished jobs is retrieved and new jobs are submitted at the same interval.

The `-P 1` and `-p 1` options set the total number of decoys to compute and the maximum number of decoys that a single computational job can handle. These values can be arbitrarily high (however the *p* value should be such that the computational job can actually compute that many decoys in the allotted *wall-clock time*).

The above command will start by printing a status report like the following:

```
Status of jobs in the 'example.csv' session:
 SUBMITTED    1/1 (100.0%)
```

It will continue printing an updated status report every 120 seconds until the requested number of decoys (set by the `-P` option) has been computed.

In GC3Pie terminology when a job is finished and its output has been successfully retrieved, the job is marked as `TERMINATED`:

```
Status of jobs in the 'example.csv' session:
 TERMINATED    1/1 (100.0%)
```

Managing a session by repeated `grosetta` invocation

We now show how one can obtain the same result by calling **grosetta** multiple times (there could be hours of interruption between one invocation and the next one).

Note: This is not the typical mode of operating with **grosetta**, but may still be useful in certain settings.

1. Create a session (1 job only, since no `-P` option is given); the session name is chosen with the `-s` (short for `--session`) option. You should take care of re-using the same session name with subsequent commands.

```
$ grosetta -s example flags alignment.filt query.fasta \
    query.psipred_ss2 boinc_aaquery03_05.200_v1_3.gz \
    boinc_aaquery09_05.200_v1_3.gz 1bjpA.pdb 1ca7A.pdb \
    2fltA.pdb 2fm7A.pdb 2op8A.pdb 2ormA.pdb 2os5A.pdb
Status of jobs in the 'example.csv' session:
 SUBMITTED   1/1 (100.0%)
```

2. Now we call **grosetta** again, and request that 3 decoys be computed starting from a single PDB file (`--total-decoys 3` on the command line). Since we are submitting a single PDB file, the 3 decoys will be computed all in a single run, so the `--decoys-per-job` option will have value `3`.

```
$ grosetta -s example --total-decoys 3 --decoys-per-job 3 \
    flags alignment.filt query.fasta \
    query.psipred_ss2 boinc_aaquery03_05.200_v1_3.gz \
    boinc_aaquery09_05.200_v1_3.gz 3c6vA.pdb
Status of jobs in the 'example.csv' session:
 SUBMITTED   3/3 (100.0%)
```

   Note that 3 jobs were submitted: **grosetta** interprets the `--total-decoys` option globally, and adds one job to compute the 2 missing decoys from the file set from step 1. (This is currently a limitation of **grosetta**)

   From here on, one could simply run `grosetta -C 120` and let it manage the session until completion of all jobs, as in the example *Manage a set of jobs from start to end* above. For the sake of showing how the use of several command-line options of **grosetta**, we shall further show how manage the session by repeated separate invocations.

3. Next step is to monitor the session, so we add the command-line option `-l` which tells **grosetta** to list all the jobs with their status. Also note that we keep the `-s example` option to tell **grosetta** that we would like to operate on the session named *example*.

   All non-option arguments can be omitted: as long as the total number of decoys is unchanged, they're not needed.

```
$ grosetta -s example -l
Decoys Nr.       State (JobID)       Info
================================================================================
0--1             RUNNING (job.766)   Running at Mon Dec 20 19:32:08 2010
2--3             RUNNING (job.767)   Running at Mon Dec 20 19:33:23 2010
0--2             RUNNING (job.768)   Running at Mon Dec 20 19:33:43 2010
```

   Without the `-l` option only a summary of job statuses is presented:

```
$ grosetta -s example
Status of jobs in the 'grosetta.csv' session:
 RUNNING     3/3 (100.0%)
```

Alternatively, we can keep the command line arguments used in the previous invocation: they will be ignored since they do not add any new job (the number of decoys to compute is always 1):

```
$ grosetta -s example -l flags alignment.filt query.fasta \
    query.psipred_ss2 boinc_aaquery03_05.200_v1_3.gz \
    boinc_aaquery09_05.200_v1_3.gz 1bjpA.pdb 1ca7A.pdb \
    2fltA.pdb 2fm7A.pdb 2op8A.pdb 2ormA.pdb 2os5A.pdb \
    3c6vA.pdb
Decoys Nr.       State (JobID)       Info
================================================================================
0--1             RUNNING (job.766)
2--3             RUNNING (job.767)   Running at Mon Dec 20 19:33:23 2010
0--2             RUNNING (job.768)   Running at Mon Dec 20 19:33:43 2010
```

Note that the `-l` option is available also in combination with the `-C` option (see *Manage a set of jobs from start to end*).

4. Calling `grosetta` again when jobs are done triggers automated download of the results:

```
$ ../grosetta.py
File downloaded:
gsiftp://idgc3grid01.uzh.ch:2811/jobs/21466129286975746820276 5/minirosetta.static.
↪stdout.txt
File downloaded:
gsiftp://idgc3grid01.uzh.ch:2811/jobs/21466129286975746820276 5/minirosetta.static.
↪log
 ...
File downloaded:
gsiftp://idgc3grid01.uzh.ch:2811/jobs/21466129286975746820276 5/.arc/input
Status of jobs in the 'grosetta.csv' session:
 TERMINATED  1/1 (100.0%)
 ok          1/1 (100.0%)
```

The `-l` option comes handy to see what directory contains the job output:

```
$ grosetta -l
Decoys Nr.       State (JobID)         Info
================================================================================
0--1             TERMINATED (job.766)  Output retrieved into directory '/tmp/0--1'
```

The `gcrypto` script

GC3Apps provide a script drive execution of multiple `gnfs-cmd` jobs each of them with a different parameter set. Allotogehter they form a single crypto simulation of a large parameter space. It uses the generic *gc3libs.cmdline.SessionBasedScript* framework.

The purpose of **`gcrypto`** is to execute *several concurrent runs* of `gnfs-cmd` on a parameter set. These runs are performed in parallel using every available GC3Pie *resource*; you can of course control how many runs should be executed and select what output files you want from each one.

Introduction

Like in a *for*-loop, the **`gcrypto`** driver script takes as input three mandatory arguments:

1. RANGE_START: initial value of the range (e.g., 800000000)

2. RANGE_END: final value of the range (e.g., 1200000000)
3. SLICE: extent of the range that will be examined by a single job (e.g., 1000)

For example:

```
# gcrypto 800000000 1200000000 1000
```

will produce 400000 jobs; the first job will perform calculations on the range 800000000 to 800000000+1000, the 2nd one will do the range 800001000 to 800002000, and so on.

Inputfile archive location (e.g. *lfc://lfc.smscg.ch/crypto/lacal/input.tgz*) can be specified with the '-i' option. Otherwise a default filename 'input.tgz' will be searched in current directory.

Job progress is monitored and, when a job is done, output is retrieved back to submitting host in folders named: `RANGE_START + (SLICE * ACTUAL_STEP)` Where `ACTUAL_STEP` correspond to the position of the job in the overall execution.

The **gcrypto** command keeps a record of jobs (submitted, executed and pending) in a session file (set name with the '-s' option); at each invocation of the command, the status of all recorded jobs is updated, output from finished jobs is collected, and a summary table of all known jobs is printed. New jobs are added to the session if new input files are added to the command line.

Options can specify a maximum number of jobs that should be in 'SUBMITTED' or 'RUNNING' state; **gcrypto** will delay submission of newly-created jobs so that this limit is never exceeded.

The **gcrypto** execute *several runs* of `gnfs-cmd` on a parameter set, and collect the generated output. These runs are performed in parallel, up to a limit that can be configured with the `-J` *command-line option*. You can of course control how many runs should be executed and select what output files you want from each one.

In more detail, **gcrypto** does the following:

1. Reads the *session* (specified on the command line with the `--session` option) and loads all stored jobs into memory. If the session directory does not exist, one will be created with empty contents.
2. Divide the initial parameter range, given in the command-line, into chunks taking the `-J` value as a reference. So from a coomand line argument like the following:

   ```
   $ gcrypto  800000000 1200000000 1000 -J 200
   ```

   gcrypto will generate an initial chunks of 200 jobs starting from the initial range 800000000 incrementing of 1000. All jobs will run `gnfs-cmd` on a specific parameter set (e.g. 800000000, 800001000, 800002000, ...). **gcrypto** will keep constant the number of simulatenous jobs running retrieving those terminated and submitting new ones untill the whole parameter range has been computed.
3. Updates the state of all existing jobs, collects output from finished jobs, and submits new jobs generated in step 2.

   Finally, a summary table of all known jobs is printed. (To control the amount of printed information, see the `-l` command-line option in the *Introduction to session-based scripts* section.)
4. If the `-C` command-line option was given (see below), waits the specified amount of seconds, and then goes back to step 3.

   The program **gcrypto** exits when all jobs have run to completion, i.e., when the whole paramenter range has been computed.

Execution can be interrupted at any time by pressing `Ctrl+C`. If the execution has been interrupted, it can be resumed at a later stage by calling **gcrypto** with exactly the same command-line options.

gcrypto requires a number of default input files common to every submited job. This list of input files is automatically fetched by **gcrypto** from a default storage repository. Those files are:

```
gnfs-lasieve6
M1019
M1019.st
M1037
M1037.st
M1051
M1051.st
M1067
M1067.st
M1069
M1069.st
M1093
M1093.st
M1109
M1109.st
M1117
M1117.st
M1123
M1123.st
M1147
M1147.st
M1171
M1171.st
M8e_1200
M8e_1500
M8e_200
M8e_2000
M8e_2500
M8e_300
M8e_3000
M8e_400
M8e_4200
M8e_600
M8e_800
tdsievemt
```

When **gcrypto** has to be executed with a different set of input files, an additional command line argument `--input-files` could be used to specify the locatin of a `tar.gz` archive containing the input files that `gnfs-cmd` will expect. Similarly, when a different version of gnfs-cmd command needs to be used, the command line argument `--gnfs-cmd` could be used to specify the location of the `gnfs-cmd` to be used.

Command-line invocation of `gcrypto`

The **gcrypto** script is based on GC3Pie's *session-based script* model; please read also the *Introduction to session-based scripts* section for an introduction to sessions and generic command-line options.

A **gcrypto** command-line is constructed as follows: Like a *for*-loop, the **gcrypto** driver script takes as input three mandatory arguments:

1. *RANGE_START*: initial value of the range (e.g., 800000000)
2. *RANGE_END*: final value of the range (e.g., 1200000000)
3. *SLICE*: extent of the range that will be examined by a single job (e.g., 1000)

Example 1. The following command-line invocation uses **gcrypto** to run `gnfs-cmd` on the parameter set ranging from 800000000 to 1200000000 with an increment of 1000.

```
$ gcrypto 800000000 1200000000 1000
```

In this case **gcrypto** will use the default values for determine the chunks size from the default value of the `-J` option (default value is 50 simulatenous jobs).

Example 2.

```
$ gcrypto --session SAMPLE_SESSION -c 4 -w 4 -m 8 800000000 1200000000 1000
```

In this example, job information is stored into session `SAMPLE_SESSION` (see the documentation of the `--session` option in *Introduction to session-based scripts*). The command above creates the jobs, submits them, and finally prints the following status report:

```
Status of jobs in the 'SAMPLE_SESSION' session: (at 10:53:46, 02/28/12)
NEW   0/50    (0.0%)
RUNNING   0/50    (0.0%)
STOPPED   0/50    (0.0%)
SUBMITTED   50/50   (100.0%)
TERMINATED   0/50    (0.0%)
TERMINATING   0/50    (0.0%)
total   50/50   (100.0%)
```

Note that the status report counts the number of *jobs in the session*, not the total number of jobs that would correspond to the whole parameter range. (Feel free to report this as a bug.)

Calling **gcrypto** over and over again will result in the same jobs being monitored;

The `-C` option tells **gcrypto** to continue running until all jobs have finished running and the output files have been correctly retrieved. On successful completion, the command given in *example 2.* above, would print:

```
Status of jobs in the 'SAMPLE_SESSION' session: (at 11:05:50, 02/28/12)
NEW   0/400k    (0.0%)
RUNNING   0/400k    (0.0%)
STOPPED   0/400k    (0.0%)
SUBMITTED   0/400k    (0.0%)
TERMINATED   50/400k   (100.0%)
TERMINATING   0/400k    (0.0%)
ok   400k/400k   (100.0%)
total   400k/400k   (100.0%)
```

Each job will be named after the parameter range it has computed (e.g. 800001000, 800002000, ...) (you could see this by passing the `-l` option to **gcrypto**); each of these jobs will create an output directory named after the job.

For each job, the set of output files is automatically retrieved and placed in the locations described below.

Output files for `gcrypto`

Upon successful completion, the output directory of each **gcrypto** job contains:

- a number of `.tgz` files each of them correspondin to a step within the execution of the `gnfs-cmd` command.
- A log file named `gcrypto.log` containing both the *stdout* and the *stderr* of the `gnfs-cmd` execution.

Note: The number of `.tgz` files may depend on whether the execution of the `gnfs-cmd` command has completed or not (e.g. jobs may be killed by the batch system when exausting requested resources)

Example usage

This section contains commented example sessions with **gcrypto**.

Manage a set of jobs from start to end

In typical operation, one calls **gcrypto** with the `-C` option and lets it manage a set of jobs until completion.

So, to compute a whole parameter range from 800000000 to 1200000000 with an increment of 1000, submitting 200 jobs simultaneously each of them requesting 4 computing cores, 8GB of memory and 4 hours of *wall-clock time*, one can use the following command-line invocation:

```
$ gcrypto -s example -C 120 -J 200 -c 4 -w 4 -m 8 800000000 1200000000 1000
```

The `-s example` option tells **gcrypto** to store information about the computational jobs in the `example.jobs` directory.

The `-C 120` option tells **gcrypto** to update job state every 120 seconds; output from finished jobs is retrieved and new jobs are submitted at the same interval.

The above command will start by printing a status report like the following:

```
Status of jobs in the 'example.csv' session:
SUBMITTED   1/1 (100.0%)
```

It will continue printing an updated status report every 120 seconds until the requested parameter range has been computed.

In GC3Pie terminology when a job is finished and its output has been successfully retrieved, the job is marked as `TERMINATED`:

```
Status of jobs in the 'example.csv' session:
TERMINATED   1/1 (100.0%)
```

Using GC3Pie utilities

GC3Pie comes with a set of generic utilities that could be used as a complemet to the **gcrypto** command to better manage a entire session execution.

`gkill`: cancel a running job

To cancel a running job, you can use the command **gkill**. For instance, to cancel *job.16*, you would type the following command into the terminal:

```
gkill job.16
```

or:

```
gkill -s example job.16
```

gkill could also be used to cancel jobs in a given state

```
gkill -s example -l UNKNOWN
```

Warning: *There's no way to undo a cancel operation!* Once you have issued a **gkill** command, the job is deleted and it cannot be resumed. (You can still re-submit it with **gresub**, though.)

ginfo: accessing low-level details of a job

It is sometimes necessary, for debugging purposes, to print out all the details about a job; the **ginfo** command does just that: prints all the details that GC3Utils know about a single job.

For instance, to print out detailed information about *job.13* in session *example*, you would type

```
ginfo -s example job.13
```

For a job in RUNNING or SUBMITTED state, only little information is known: basically, where the job is running, and when it was started:

```
$ ginfo -s example job.13
job.13
    cores: 2
    execution_targets: hera.wsl.ch
    log:
        SUBMITTED at Tue May 15 09:52:05 2012
        Submitted to 'wsl' at Tue May 15 09:52:05 2012
        RUNNING at Tue May 15 10:07:39 2012
    lrms_jobid: gsiftp://hera.wsl.ch:2811/jobs/116613370683251353308673
    lrms_jobname: LACAL_800001000
    original_exitcode: -1
    queue: smscg.q
    resource_name: wsl
    state_last_changed: 1337069259.18
    stderr_filename: gcrypto.log
    stdout_filename: gcrypto.log
    timestamp:
        RUNNING: 1337069259.18
        SUBMITTED: 1337068325.26
    unknown_iteration: 0
    used_cputime: 1380
    used_memory: 3382706
```

If you omit the job number, information about *all* jobs in the session will be printed.

Most of the output is only useful if you are familiar with GC3Utils inner working. Nonetheless, **ginfo** output is definitely something you should include in any report about a misbehaving job!

For a finished job, the information is more complete and can include error messages in case the job has failed:

```
$ ginfo -c -s example job.13
job.13
    _arc0_state_last_checked: 1337069259.18
    _exitcode: 0
    _signal: None
    _state: TERMINATED
    cores: 2
    download_dir: /data/crypto/results/example.out/8000001000
    execution_targets: hera.wsl.ch
    log:
        SUBMITTED at Tue May 15 09:52:04 2012
```

(continues on next page)

(continued from previous page)

```
        Submitted to 'wsl' at Tue May 15 09:52:04 2012
        TERMINATING at Tue May 15 10:07:39 2012
        Final output downloaded to '/data/crypto/results/example.out/8000001000'
        TERMINATED at Tue May 15 10:07:43 2012
    lrms_jobid: gsiftp://hera.wsl.ch:2811/jobs/1144133706832458458503 2
    lrms_jobname: LACAL_800001000
    original_exitcode: 0
    queue: smscg.q
    resource_name: wsl
    state_last_changed: 1337069263.13
    stderr_filename: gcrypto.log
    stdout_filename: gcrypto.log
    timestamp:
        SUBMITTED: 1337068324.87
        TERMINATED: 1337069263.13
        TERMINATING: 1337069259.18
    unknown_iteration: 0
    used_cputime: 360
    used_memory: 3366977
    used_walltime: 300
```

With option `-v`, **`ginfo`** output is even more verbose and complete, and includes information about the application itself, the input and output files, plus some backend-specific information:

```
$ ginfo -c -s example job.13
job.13
  arguments: 800000800, 100, 2, input.tgz
  changed: False
  environment:
  executable: gnfs-cmd
  executables: gnfs-cmd
  execution:
      _arc0_state_last_checked: 1337069259.18
      _exitcode: 0
      _signal: None
      _state: TERMINATED
      cores: 2
      download_dir: /data/crypto/results/example.out/8000001000
      execution_targets: hera.wsl.ch
      log:
          SUBMITTED at Tue May 15 09:52:04 2012
          Submitted to 'wsl' at Tue May 15 09:52:04 2012
          TERMINATING at Tue May 15 10:07:39 2012
          Final output downloaded to '/data/crypto/results/example.out/8000001000'
          TERMINATED at Tue May 15 10:07:43 2012
      lrms_jobid: gsiftp://hera.wsl.ch:2811/jobs/1144133706832458458503 2
      lrms_jobname: LACAL_800001000
      original_exitcode: 0
      queue: smscg.q
      resource_name: wsl
      state_last_changed: 1337069263.13
      stderr_filename: gcrypto.log
      stdout_filename: gcrypto.log
      timestamp:
          SUBMITTED: 1337068324.87
          TERMINATED: 1337069263.13
```

(continues on next page)

(continued from previous page)

```
          TERMINATING: 1337069259.18
      unknown_iteration: 0
      used_cputime: 360
      used_memory: 3366977
      used_walltime: 300
  inputs:
      srm://dpm.lhep.unibe.ch/dpm/lhep.unibe.ch/home/crypto/gnfs-cmd_20120406: gnfs-
↪cmd
      srm://dpm.lhep.unibe.ch/dpm/lhep.unibe.ch/home/crypto/lacal_input_files.tgz:␣
↪input.tgz
  jobname: LACAL_800000900
  join: True
  output_base_url: None
  output_dir: /data/crypto/results/example.out/8000001000
  outputs:
      @output.list: file, , @output.list, None, None, None, None
      gcrypto.log: file, , gcrypto.log, None, None, None, None
  persistent_id: job.1698503
  requested_architecture: x86_64
  requested_cores: 2
  requested_memory: 4
  requested_walltime: 4
  stderr: None
  stdin: None
  stdout: gcrypto.log
  tags: APPS/CRYPTO/LACAL-1.0
```

The GC3Utils software

The GC3Utils are lower-level commands, provided to perform common operations on jobs, regardless of their type or the application they run.

For instance, GC3Utils provide commands to obtain the list and status of computational resources (**`gservers`**); to clear the list of jobs from old and failed ones (**`gclean`**); to get detailed information on a submitted job (**`ginfo`**, mainly for debugging purposes).

This chapter is a tutorial for the GC3Utils command-line utilities.

If you find a technical term whose meaning is not clear to you, please look it up in the *Glossary*. (But feel free to ask on the GC3Pie mailing list if it's still unclear!)

Contents

- **gresub**: *re-submit a failed job*
- **gservers**: *list available resources*
- **ginfo**: *accessing low-level details of a job*
- **gselect**: *select job ids from from a session*
- **gcloud**: *manage VMs created by the EC2 backend*

gsession: manage sessions

All jobs managed by one of the GC3Pie scripts are grouped into sessions; information related of a session is stored into a directory. The **gsession** command allows you to show the jobs related to a specific session, to abort the session or to completely delete it.

The **gsession** accept two mandatory arguments: *command* and *session*. *command* must be one of:

list list jobs related to the session.

log show the session history.

abort kill all jobs related to the session.

delete abort the session and delete the session directory from disk.

For instance, if you want to check the status of the main tasks of a session, just run:

```
$ gsession list SESSION_DIRECTORY
+--------------------------------+--------------------------+-------+----------------
↪-----------------+
| JobID                          | Job name                 | State | Info          ␣
↪                 |
+--------------------------------+--------------------------+-------+----------------
↪-----------------+
| ParallelTaskCollection.1140527 | ParallelTaskCollection-N1 | NEW   | NEW at Fri Feb␣
↪22 16:39:34 2013 |
+--------------------------------+--------------------------+-------+----------------
↪-----------------+
```

This command will only show the *top-level tasks*, e.g. the main tasks created by the GC3 script. If you want to see **all** the tasks related to the session run the command with the option `-r`:

```
$ gsession list SESSION_DIRECTORY -r
+---------------------------------------+--------------------------+------------+--
↪-------------------------------------+
| JobID                                 | Job name                 | State      |␣
↪Info                                 |
+---------------------------------------+--------------------------+------------+--
↪-------------------------------------+
| ParallelTaskCollection.1140527        | ParallelTaskCollection-N1 | NEW        |␣
↪NEW at Fri Feb 22 16:39:34 2013      |
|   WarholizeWorkflow.1140528           | WarholizedWorkflow       | RUNNING    |␣
↪RUNNING at Fri Feb 22 16:39:34 2013  |
|     GrayScaleConvertApplication.1140529 |                        | TERMINATED |␣
↪TERMINATED at Fri Feb 22 16:39:25 2013 |
|     TricolorizeMultipleImages.1140530 | Warholizer_Parallel      | NEW        | ␣
↪                                     |
|       TricolorizeImage.1140531        | TricolorizeImage         | NEW        | ␣
↪                                     |
```

(continues on next page)

(continued from previous page)

```
|          CreateLutApplication.1140532    |                            | NEW         | ␣
↪                                        |
|       TricolorizeImage.1140533           | TricolorizeImage           | NEW         | ␣
↪                                        |
|          CreateLutApplication.1140534    |                            | NEW         | ␣
↪                                        |
|       TricolorizeImage.1140535           | TricolorizeImage           | NEW         | ␣
↪                                        |
|          CreateLutApplication.1140536    |                            | NEW         | ␣
↪                                        |
|       TricolorizeImage.1140537           | TricolorizeImage           | NEW         | ␣
↪                                        |
|          CreateLutApplication.1140538    |                            | NEW         | ␣
↪                                        |
+-----------------------------------------+----------------------------+-------------+--
↪---------------------------------------+
```

To have the full history of the session run *gsession log*:

```
$ gsession log SESSION_DIRECTORY
Feb 22 16:39:01 GrayScaleConvertApplication.1140529: Submitting to 'hobbes' at Fri␣
↪Feb 22 16:39:01 2013
Feb 22 16:39:08 GrayScaleConvertApplication.1140529: RUNNING
Feb 22 16:39:08 GrayScaleConvertApplication.1140529: SUBMITTED
Feb 22 16:39:08 GrayScaleConvertApplication.1140529: Submitted to 'hobbes' at Fri Feb␣
↪22 16:39:08 2013
Feb 22 16:39:08 WarholizeWorkflow.1140528: SUBMITTED
Feb 22 16:39:24 GrayScaleConvertApplication.1140529: TERMINATING
Feb 22 16:39:25 WarholizeWorkflow.1140528: RUNNING
Feb 22 16:39:25 ParallelTaskCollection.1140527: RUNNING
Feb 22 16:39:25 GrayScaleConvertApplication.1140529: Final output downloaded to
↪'Warholized.lena.jpg'
Feb 22 16:39:25 GrayScaleConvertApplication.1140529: TERMINATED
Feb 22 16:39:34 WarholizeWorkflow.1140528: NEW
Feb 22 16:39:34 ParallelTaskCollection.1140527: NEW
Feb 22 16:39:34 WarholizeWorkflow.1140528: RUNNING
```

To abort a session, run the *gsession abort* command:

```
$ gsession abort SESSION_DIRECTORY
```

This will kill all the running jobs and retrieve the results of the terminated jobs, but will leave the session directory untouched. To also delete the session directory, run *gsession delete*:

```
$ gsession delete SESSION_DIRECTORY
```

`gstat`: monitor the status of submitted jobs

To see the status of all the jobs you have submitted, use the **`gstat`** command. Typing:

```
gstat -s SESSION
```

will print to the screen a table like the following:

```
Job ID    Status
====================
job.12    TERMINATED
job.15    SUBMITTED
job.16    RUNNING
job.17    RUNNING
job.23    NEW
```

Note: If you have never submitted any job, or if you have cleared your job list with the **gclean** command, then **gstat** will print *nothing* to the screen!

A job can be in one and only one of the following states:

`NEW`

The job has been created but not yet submitted: it only exists on the local disk.

`RUNNING`

The job is currently running – there's nothing to do but wait.

`SUBMITTED`

The job has been sent to a compute resource for execution – it should change to `RUNNING` status eventually.

`STOPPED`

The job was sent to a remote cluster for execution, but it is stuck there for some unknown reason. There is no automated procedure in this case: the best thing you can do is to contact the systems administrator to determine what has happened.

`UNKNOWN`

Job info is not found, possibly because the remote resource is currently not accessible due to a network error, a misconfiguration or because the remote resource is not available anymore. When the root cause is fixed, and the resource is available again, the status of the job should automatically move to another state.

`TERMINATED`

The job has finished running; now there are three things you can do:

1. Use the **gget** command to get the command output files back from the remote execution cluster.
2. Use the **gclean** command to remove this job from the list. After issuing **gclean** on a job, any information on it is lost, so be sure you have retrieved any interesting output with **gget** before!
3. If something went wrong during the execution of the job (it did not complete its execution or -possibly- it did not even start), you can use the **ginfo** command to try to debug the problem.

The list of submitted jobs persists from one session to the other: you can log off, shut your computer down, then turn it on again next day and you will see the same list of jobs.

Note: Completed jobs persist in the **gstat** list until they are cleared off with the **gclean** command.

`gtail`: peeking at the job output and error report

Once a job has reached `RUNNING` status (check with **`gstat`**), you can also monitor its progress by looking at the last lines in the job output and error stream.

An example might clarify this: assume you have submitted a long-running computation as *job.16* and you know from **`gstat`** that it got into `RUNNING` state; then to take a peek at what this job is doing, you issue the following command:

```
gtail job.16
```

This would produce the following output, from which you can deduce how far GAMESS has progressed into the computation:

```
RECOMMEND NRAD ABOVE  50 FOR ZETA'S ABOVE 1E+4

RECOMMEND NRAD ABOVE  75 FOR ZETA'S ABOVE 1E+5

RECOMMEND NRAD ABOVE 125 FOR ZETA'S ABOVE 1E+6

DFT IS SWITCHED OFF, PERFORMING PURE SCF UNTIL SWOFF THRESHOLD IS REACHED.


ITER EX DEM     TOTAL ENERGY        E CHANGE  DENSITY CHANGE    DIIS ERROR

  1  0  0    -1079.0196780290 -1079.0196780290   0.343816910   1.529879639

         * * *   INITIATING DIIS PROCEDURE   * * *

  2  1  0    -1081.1910665431    -2.1713885141   0.056618918   0.105322104

  3  2  0    -1081.2658345285    -0.0747679855   0.019565324   0.044813607
```

By default, **`gtail`** only outputs the last 10 lines of a job output/error stream. To see more, use the command line option `-n`; for example, to see the last 25 lines of the output, issue the command:

```
gtail -n 25 job.16
```

The command **`gtail`** is especially useful for long computations: you can see how far a job has gotten and, e.g., cancel it if it's gotten stuck into an endless/unproductive loop.

To "keep an eye" over what a job is doing, you can add the `-f` option to **`gtail`**: this will run **`gtail`** in "follow" mode, i.e., **`gtail`** will continue to display the contents of the job output and update it as time passes, until you hit Ctrl+C to interrupt it.

`gkill`: cancel a running job

To cancel a running job, you can use the command **`gkill`**. For instance, to cancel *job.16*, you would type the following command into the terminal:

```
gkill job.16
```

Warning: *There's no way to undo a cancel operation!* Once you have issued a **`gkill`** command, the job is deleted and it cannot be resumed. (You can still re-submit it with **`gresub`**, though.)

`gget`: retrieve the output of finished jobs

Once a job has reached `RUNNING` status (check with **`gstat`**), you can retrieve its output files with the **`gget`** command. For instance, to download the output files of *job.15* you would use:

```
gget job.15
```

This command will print out a message like:

```
Job results successfully retrieved in '/path/to/some/directory'
```

If you are not running the **`gget`** command on your computer, but rather on a shared front-end like *ocikbgtw*, you can copy+paste the path within quotes to the sftp command to get the files to your usual workstation. For example, you can run the following command in a terminal on your computer to get the output files back to your workstation:

```
sftp ocikbgtw:'/path/to/some/directory'
```

This will take you to the directory where the output files have been stored.

`gclean`: remove a completed job from the status list

Jobs persist in the **`gstat`** list until they are cleared off; you need to use the **`gclean`** command for that.

Just call the **`gclean`** command followed by the job identifier *job.NNN*. For example:

```
gclean job.23
```

In normal operation, you can only remove jobs that are in the `TERMINATED` status; if you want to force **`gclean`** to remove a job that is not in any one of those states, just add `-f` to the command line.

`gresub`: re-submit a failed job

In case a job failed for accidental causes (e.g., the site where it was running went unexpectedly down), you can re-submit it with the **`gresub`** command.

Just call **`gresub`** followed by the job identifier *job.NNN*. For example:

```
gresub job.42
```

Resubmitting a job that is not in a terminal state (i.e., `TERMINATED`) results in the job being killed (as with **`gkill`**) before being submitted again. If you are unsure what state a job is in, check it with **`gstat`**.

`gservers`: list available resources

The **`gservers`** command prints out information about the configured resources. For each resource, a summary of the information recorded in the configuration file and the current resource status is printed. For example:

```
$ gservers
+---------------------------------------------------------------+
|                      smscg                                    |
+===============================================================+
|                   Frontend host name / frontend   giis.smscg.ch |
|                              Access mode / type   arc0         |
```

(continues on next page)

(continued from previous page)

```
|                      Authorization name / auth   smscg         |
|                          Accessible? / updated   1             |
|                  Total number of cores / ncores   4000         |
|                      Total queued jobs / queued   3475         |
|                    Own queued jobs / user_queued   0           |
|                      Own running jobs / user_run   0           |
|           Max cores per job / max_cores_per_job   256          |
| Max memory per core (MB) / max_memory_per_core   2000          |
|  Max walltime per job (minutes) / max_walltime   1440          |
+----------------------------------------------------------------+
```

The meaning of the printed fields is as follows:

- The title of each box is the "resource name", as you would write it after the *-r* option to **gsub**.
- *Access mode / type*: it is the kind of software that is used for accessing the resource; consult Section *Configuration File* for more information about resource types.
- *Authorization name / auth*: this is paired with the *Access mode / type*, and identifies a section in the *configuration file* where authentication information for this resource is stored; see Section *Configuration File* for more information.
- *Accessible? / updated*: whether you are *currently* authorized to access this resource; note that if this turns *False* or *0* for resources that you should have access to, then something is wrong either with the state of your system, or with the resource itself. (The procedure on how to diagnose this is too complex to list here; consult your friendly systems administrator :-))
- *Total number of cores*: the total number of cores present on the resource. Note this can vary over time as cluster nodes go in and out of service: computers break, then are repaired, then break again, etc.
- *Total queued jobs*: number of jobs (from all users) waiting to be executed on the remote compute cluster.
- *Own queued jobs*: number of jobs (submitted by you) waiting to be executed on the remote compute cluster.
- *Own running jobs*: number of jobs (submitted by you) currently executing on the remote compute cluster.
- *Max cores per job*: the maximum number of cores that you can request for a single computational job on this resource.
- *Max memory per core*: maximum amount of memory (per core) that you can request on this resource. The amount shows the maximum requestable memory in MB.
- *Max walltime per job*: maximum duration of a computational job on this resource. The amount shows the maximum time in seconds.

The whole point of GC3Utils is to abstract job submission and management from detailed knowledge of the resources and their hardware and software configuration, but it is sometimes convenient and sometimes necessary to get into this level of detail...

`ginfo`: accessing low-level details of a job

It is sometimes necessary, for debugging purposes, to print out all the details about a job; the **ginfo** command does just that: prints all the details that GC3Utils know about a single job.

For instance, to print out detailed information about *job.13* in session *TEST1*, you would type:

```
ginfo -s TEST1 job.13
```

For a job in RUNNING or SUBMITTED state, only little information is known: basically, where the job is running, and when it was started:

```
$ ginfo -s XXX job.13
job.13
    execution_targets: hera.wsl.ch
    log:
        SUBMITTED at Wed Mar  7 17:40:07 2012
        Submitted to 'smscg' at Wed Mar  7 17:40:07 2012
    lrms_jobid: gsiftp://hera.wsl.ch:2811/jobs/5935133113840717715461 95
    resource_name: smscg
    state_last_changed: 1331138407.33
    timestamp:
        SUBMITTED: 1331138407.33
```

If you omit the job number, information about *all* jobs in the session will be printed.

Most of the output is only useful if you are familiar with GC3Utils inner working. Nonetheless, **ginfo** output is definitely something you should include in any report about a misbehaving job!

For a finished job, the information is more complete and can include error messages in case the job has failed:

```
$ ginfo -s TEST1 job.13
job.13
    cores: 1
    download_dir: /home/rmurri/gc3/gc3pie.googlecode.com/gc3pie/gc3apps/gamess/exam01
    execution_targets: idgc3grid01.uzh.ch
    log:
        SUBMITTED at Wed Mar  7 15:52:37 2012
        Submitted to 'idgc3grid01' at Wed Mar  7 15:52:37 2012
        TERMINATING at Wed Mar  7 15:54:52 2012
        Final output downloaded to '/home/rmurri/gc3/gc3pie.googlecode.com/gc3pie/
↪gc3apps/gamess/exam01'
        TERMINATED at Wed Mar  7 15:54:53 2012
        Execution of gamess terminated normally wed mar  7 15:52:42 2012
    lrms_jobid: gsiftp://idgc3grid01.uzh.ch:2811/jobs/29387133113195716781566 70
    lrms_jobname: exam01
    original_exitcode: 0
    queue: all.q
    resource_name: idgc3grid01
    state_last_changed: 1331132093.18
    stderr_filename: exam01.out
    stdout_filename: exam01.out
    timestamp:
        SUBMITTED: 1331131957.49
        TERMINATED: 1331132093.18
        TERMINATING: 1331132092.74
    used_cputime: 0
    used_memory: 492019
    used_walltime: 60
```

With option -v, **ginfo** output is even more verbose and complete, and includes information about the application itself, the input and output files, plus some backend-specific information:

```
$ ginfo -c -s TEST1 job.13
job.13
    application_tag: gamess
    arguments: exam01.inp
```

(continues on next page)

(continued from previous page)

```
    changed: False
    environment:
    executable: /$GAMESS_LOCATION/nggms
    execution:
        _arc0_state_last_checked: 1331138407.33
        _exitcode: None
        _signal: None
        _state: SUBMITTED
        execution_targets: hera.wsl.ch
        log:
            SUBMITTED at Wed Mar  7 17:40:07 2012
            Submitted to 'smscg' at Wed Mar  7 17:40:07 2012
        lrms_jobid: gsiftp://hera.wsl.ch:2811/jobs/59351331138407177154619 5
        resource_name: smscg
        state_last_changed: 1331138407.33
        timestamp:
            SUBMITTED: 1331138407.33
    inp_file_path: test/data/exam01.inp
    inputs:
        file:///home/rmurri/gc3/gc3pie.googlecode.com/gc3pie/gc3apps/gamess/test/data/
↪exam01.inp: exam01.inp
    job_name: exam01
    jobname: exam01
    join: True
    output_base_url: None
    output_dir: /home/rmurri/gc3/gc3pie.googlecode.com/gc3pie/gc3apps/gamess/exam01
    outputs:
        exam01.dat: file, , exam01.dat, None, None, None, None
        exam01.out: file, , exam01.out, None, None, None, None
    persistent_id: job.33998
    requested_architecture: None
    requested_cores: 1
    requested_memory: 2
    requested_walltime: 8
    stderr: None
    stdin: None
    stdout: exam01.out
    tags: APPS/CHEM/GAMESS-2010
    verno: None
```

`gselect`: select job ids from from a session

The **`gselect`** command allows you to select Job IDs from a GC3Pie session that satisfy the selected criteria. This command is usually used in combination with **`gresub`**, **`gkill`**, **`ginfo`**, **`gget`** or **`gclean`**, for instance:

```
$ gselect -l STOPPED | xargs gresub
```

The output of this command is a list of Job IDs, one per line. The criteria specified by command-line options will be AND'ed together, i.e., a job must satisfy all of them in order to be selected.

You can select a job based on the following criteria:

`JobID regexp`

Use option *–jobid REGEXP* to select jobs whose ID matches the supplied regular expression (case insensitive)

`Job state`

Use option *–state STATE[,STATE...]* to select jobs in one of the specified states, for instance to select jobs in either STOPPED or SUBMITTED state, run *gselect –state STOPPED,SUBMITTED*.

`exit status`

You can select jobs that terminated with exit status equal to 0 with *–ok* option. To select failed jobs instead (exit status different from 0), use option *–failed*

`Submission time`

Use option *–submitted-before DATE* and *–submitted-after DATE* to select jobs submitted before or after a specific date. *DATE* must be in a human readable format recognized by the *parsedatetime <https://pypi.python.org/pypi/parsedatetime/>* module, for instance *in 2 hours*, *yesterday* or *10 November 2014, 1pm*.

`gcloud`: manage VMs created by the EC2 backend

The **`gcloud`** command allows you to show and manage VMs created by the EC2 backend.

To show a list of VMs currently running on the EC2 resources correctly configured run:

```
$ gcloud list
====================================
VMs running on EC2 resource `hobbes`
====================================

+------------+---------+---------------+-------------+--------------+---------+
|     id     |  state  |   public ip   | Nr. of jobs |   image id   | keypair |
+------------+---------+---------------+-------------+--------------+---------+
| i-0000053e | running | 130.60.193.45 |      1      | ami-00000035 | antonio |
+------------+---------+---------------+-------------+--------------+---------+
```

This command will show various information, if available, including the number of jobs currently running (or in *TERMINATED* state) on those VM, so that you can easily identify if there is a VM which is not used by any of yours script and you can safely terminate it.

If you want to terminate a VM run the *gcloud terminate* command. In this case, however, you also have to specify the name of the resource with the option `-r`, and the ID of the VM you want to terminate:

```
$ gcloud terminate -r hobbes i-0000053e
```

An empty output is a signal that the VM has been terminated.

The *EC2* backend keeps track of all the VM it created, so that if a VM is not needed anymore it is able to terminate it automatically. However, sometimes you may need to keep a VM up&running and thus you need to tell the EC2 backend to ignore that VM.

This is possible with the *gcloud forget* command. You must supply the correct resource name with `-r RESOURCE_NAME` and a valid VM ID, and if the command succeeds then the VM will never be used by the EC2 backend. Please note also that after running *gcloud forget*, the VM will not be shown in the output of *gcloud list*.

The following example will explain the behavior:

```
$ gcloud list -r hobbes

====================================
VMs running on EC2 resource `hobbes`
```

(continues on next page)

(continued from previous page)

```
====================================

+------------+---------+---------------+-------------+--------------+---------+
|     id     |  state  |   public ip   | Nr. of jobs |   image id   | keypair |
+------------+---------+---------------+-------------+--------------+---------+
| i-00000540 | pending | 130.60.193.45 |     N/A     | ami-00000035 | antonio |
+------------+---------+---------------+-------------+--------------+---------+
```

then we run *gcloud forget*:

```
$ gcloud forget  -r hobbes i-00000540
```

and we run again *gcloud list*:

```
$ gcloud list -r hobbes

====================================
VMs running on EC2 resource `hobbes`
====================================

  no known VMs are currently running on this resource.
```

You can also create a new VM using the default settings using the *gcloud run* command. In this case too you have to specify the `-r` command line option. The output of this command contains some basic information about the created VM:

```
$ gcloud run -r hobbes
+------------+---------+-----------------------------------------------+--------------+--
↪------------+---------+
|     id     |  state  |                  public ip                    | Nr. of jobs | ␣
↪ image id   | keypair |
+------------+---------+-----------------------------------------------+--------------+--
↪------------+---------+
| i-00000541 | pending | server-4e68ebc4-ea52-45ff-82d0-79699300b323 |     N/A     |␣
↪ami-00000035 | antonio |
+------------+---------+-----------------------------------------------+--------------+--
↪------------+---------+
```

Please note that while the VM is still in *pending* state, the value of the *public ip* field may be meaningless. A successive run of *gcloud list* should show you the correct *public ip*.

Troubleshooting GC3Pie

This page lists a number of errors and issues that you might run into, together with their solution. Please use the GC3Pie mailing list for further help and for any problem not reported here!

Each section covers a different Python error; the section is named after the error name appearing in the *last line* of the Python traceback. (See section *What is a Python traceback?* below)

Contents

What is a Python traceback?

A *traceback* is a long Python error message, detailing the call stack in the code that lead to a specific error condition.

Tracebacks always look like this one (the number of lines printed, the files involved and the actual error message will, of course, vary):

```
Traceback (most recent call last):
 File "/home/mpackard/gc3pie/bin/gsub", line 9, in <module>
   load_entry_point('gc3pie==1.0rc7', 'console_scripts', 'gsub')()
 File "/home/mpackard/gc3pie/lib/python2.5/site-packages/gc3pie-1.0rc7-py2.5.egg/
↪gc3utils/frontend.py", line 137, in main
   import gc3utils.commands
 File "/home/mpackard/gc3pie/lib/python2.5/site-packages/gc3pie-1.0rc7-py2.5.egg/
↪gc3utils/commands.py", line 31, in <module>
   import cli.app
 File "/home/mpackard/gc3pie/lib/python2.5/site-packages/pyCLI-2.0.2-py2.5.egg/cli/
↪app.py", line 37, in <module>
   from cli.util import ifelse, ismethodof
 File "/home/mpackard/gc3pie/lib/python2.5/site-packages/pyCLI-2.0.2-py2.5.egg/cli/
↪util.py", line 28, in <module>
   BaseStringIO = StringIO.StringIO
AttributeError: 'module' object has no attribute 'StringIO'
```

Let's analyize how a traceback is formed, top to bottom.

A traceback is *always* started by the line:

```
Traceback (most recent call last):
```

Then follow a number of line pairs like this one:

```
File "/home/mpackard/gc3pie/lib/python2.5/site-packages/gc3pie-1.0rc7-py2.5.egg/
↪gc3utils/frontend.py", line 137, in main
  import gc3utils.commands
```

The first line shows the file name and the line number where the program stopped; the second line displays the instruction that Python was executing when the error occurred. *We shall always omit this part of the traceback in the listings below.*

Finally, the traceback ends with the error message on the *last* line:

```
AttributeError: 'module' object has no attribute 'StringIO'
```

Just look up this error message in the section headers below; if you cannot find any relevant section, please write to the GC3Pie mailing list for help.

Common errors using GC3Pie

This section section lists Python errors that may happen when using GC3Pie; each section is named after the error name appearing in the *last line* of the Python traceback. (See section *What is a Python traceback?* above.)

If you get an error that is not listed here, please get in touch via the GC3Pie mailing list.

AttributeError: *module* object has no attribute *StringIO*

This error:

```
Traceback (most recent call last):
 ...
 File "/home/mpackard/gc3pie/lib/python2.5/site-packages/pyCLI-2.0.2-py2.5.egg/cli/
↪util.py",
line 28, in <module>
   BaseStringIO = StringIO.StringIO
AttributeError: 'module' object has no attribute 'StringIO'
```

is due to a conflicts of the pyCLI library (prior to version 2.0.3) and the Debian/Ubuntu package *python-stats*

There are three ways to get rid of the error:

1. Uninstall the **python-stats* package* <*python-stats*> (run the command `apt-get remove python-stats` as user `root`)
2. Upgrade pyCLI to version 2.0.3 at least.
3. *Upgrade* GC3Pie, which will force an upgrade of pyCLI.

DistributionNotFound

If you get this error:

```
Traceback (most recent call last):
    ...
pkg_resources.DistributionNotFound: gc3pie==1.0rc2
```

It usually means that you didn't run `source ../bin/activate; ./setup.py develop` when upgrading GC3Pie.

Please re-do the steps in the *GC3Pie Upgrade instructions* to fix the error.

ImportError: No module named `pstats`

This error only occurs on Debian and Ubuntu GNU/Linux:

```
Traceback (most recent call last):
File ".../pyCLI-2.0.2-py2.6.egg/cli/util.py", line 19, in <module>
   import pstats
ImportError: No module named pstats
```

To solve the issue: install the **python-profiler* package <python-profiler>*:

```
apt-get install python-profiler # as `root` user
```

NoResources: Could not initialize any computational resource - please check log and configuration file.

This error:

```
Traceback (most recent call last):
  ...
  File ".../src/gc3libs/core.py", line 150, in submit
    raise gc3libs.exceptions.NoResources("Could not initialize any computational␣
↪resource"
gc3libs.exceptions.NoResources: Could not initialize any computational resource -␣
↪please check log and configuration file.
```

can have two different causes:

1. You didn't create a configuration file, or you did not list any resource in it.
2. Some other error prevented the resources from being initialized, or the configuration file from being properly read.

ValueError: I/O operation on closed file

Sample error traceback (may be repeated multiple times over):

```
Traceback (most recent call last):
  File "/usr/lib/python2.5/logging/__init__.py", line 750, in emit
    self.stream.write(fs % msg)
ValueError: I/O operation on closed file
```

This is discussed in Issue 182; a fix have been committed to release 1.0, so if you are seeing this error, you are running a pre-release version of GC3Pie and should *Upgrade*.

ValueError: Expected version spec in ...

When trying to install GC3Pie with `pip install`, you get a long error report that ends with this Python traceback:

```
Traceback (most recent call last):
  File "/opt/python/2.7.9/lib/python2.7/site-packages/pip/basecommand.py", line 232,␣
↪in main
    status = self.run(options, args)
  File "/opt/python/2.7.9/lib/python2.7/site-packages/pip/commands/install.py", line␣
↪339, in run
    requirement_set.prepare_files(finder)
  File "/opt/python/2.7.9/lib/python2.7/site-packages/pip/req/req_set.py", line 436,␣
↪in prepare_files
    req_to_install.extras):
  File "/opt/python/2.7.9/lib/python2.7/site-packages/pip/_vendor/pkg_resources/__
↪init__.py", line 2496, in requires
    dm = self._dep_map
  File "/opt/python/2.7.9/lib/python2.7/site-packages/pip/_vendor/pkg_resources/__
↪init__.py", line 2491, in _dep_map
```

(continues on next page)

(continued from previous page)

```
    dm.setdefault(extra,[]).extend(parse_requirements(reqs))
  File "/opt/python/2.7.9/lib/python2.7/site-packages/pip/_vendor/pkg_resources/__
↪init__.py", line 2820, in parse_requirements
    "version spec")
  File "/opt/python/2.7.9/lib/python2.7/site-packages/pip/_vendor/pkg_resources/__
↪init__.py", line 2785, in scan_list
    raise ValueError(msg, line, "at", line[p:])
ValueError: ('Expected version spec in', 'python-novaclient;python_version>="2.7"',
↪'at', ';python_version>="2.7"')
```

This means that the `pip` command is too old to properly parse `Python environment markers <https://www.python.org/dev/peps/pep-0508/>`_; ``pip`` version 8.1.2 is the first one known to work well.

To fix the issue, please upgrade `pip` to (at least) version 8.1.2:

```
pip install --upgrade 'pip>=8.1.2'
```

User-visible changes across releases

This is a list of user-visible changes worth mentioning. In each new release, items are added to the top of the file and identify the version they pertain to.

Contents

GC3Pie 2.6

GC3Pie 2.6.0 introduces compatibility with Python 3.5+. The changes for this are rather extensive, but lucklily mostly confined to GC3Pie internals, so users of the library should not notice.

Warning: This release can introduce a few backwards-incompatible changes in the format for persisting tasks in files and databases (see *Important changes* below). Be sure to have all your currently-running sessions done before you upgrade!

This release depends on a few new external packages; if you're upgrading from earlier sources, be sure to re-run *pip install .* in the GC3Pie source directory; no such additional step is needed if you're installing from PyPI with *pip install gc3pie* or using GC3Pie's own *install.py* script.

Important changes

- Python 3.5+ is now fully supported and tested!
- The on-disk format for saving jobs might have changed incompatibly in some cases: a few internal classes have completely changed their inheritance hierarchy so Python's *pickle* might not be able to read them back.
- GC3Pie now defaults to using "unicode" strings everywhere, but will make a best attempt at converting parameters passed as byte strings:
    - command-line arguments and paths for I/O need to be converted using the locale's own encoding/charset and revert to mapping byte strings to Unicode code points by keeping the *numeric* value of bytes (instead of the textual / glyph value) if the former attempt has failed
    - output from commands (e.g., when interacting with a batch-queuing system): we *assume* that programs are complying with the locale-defined encoding and use the locale's own encoding to convert the output into a unicode text string.
- Minor (internal) API changes:
    - class *gc3libs.Default* is now a separate module *gc3libs.default*.
    - a few unused utility methods have been removed from module *gc3libs.utils*.

Bug fixes

- SLURM: ignore extraneous lines in *squeue* output.
- LSF: If accounting command name contains the string *bjobs*, parse its output like bjobs'.

GC3Pie 2.5

New features

- New `SessionBasedDaemon` and accompanying "Inbox" classes to implement scripts that automatically detach themselves into background and react to events in configurable sources (filesystem, database, S3/SWIFT storage).
- Dropping cloud infrastructure support in Python 2.6; if you run GC3Pie on Python 2.6, you will only be able to run tasks on the "localhost" resource, or on any of the supported batch-queuing systems.
- Terminal log output is now colorized according to message level! (Thanks to Adrian Etter for suggesting this feature.)

Incompatible changes

- Old-style sessions are not supported any more. (This should not be a problem, as they have been automatically converted to "new-style" since years now. In the unlikely case you still have an old-style session directory on disk, just run any session command from version 2.4 and it will convert the format automatically.)

GC3Pie 2.4

New features

- The environment variable `GC3PIE_RESOURCE_INIT_ERRORS_ARE_FATAL` can be set to `yes` or `1` to cause GC3Pie to abort if any errors occur while initializing the configured resources. The default behavior of GC3Pie is instead to keep running until there is at least one resource that can be used.
- A resource is now automatically disabled if an unrecoverable error occurs during its use.

GC3Pie 2.3

Incompatible changes

- The ARC backends and supporting code have been removed: it is no longer possible to use GC3Pie to submit tasks to an ARC job manager.
- The environment variable `GC3PIE_NO_CATCH_ERRORS` now can specify a list of patterns to selectively unignore unexpected/generic errors in the code. As this feature should only be used in debugging code, we allow ourselves to break backwards compatibility.
- The cloud and mathematics libraries are no longer installed by default with `pip install gc3pie` – please use:

```
pip install gc3pie[openstack,ec2,optimizer]
```

to install support for all optional backends and libraries.

- The *gc3libs.utils.ifelse* function was removed in favor of Python's ternary operator.

New features

- New task collection `DependentTaskCollection` to run a collection of tasks with given pre/post dependencies across them.
- GC3Pie will now parse and obey the `Port`, `Identity`, `User`, `ConnectionTimeout`, and `ProxyCommand` options from the SSH config file. Location of an alternate configuration file to use with GC3Pie can be set in any `[auth/*]` section of type SSH; see the *Configuration File* section for details. Thanks to Niko Eherenfeuchter and Karandash8 for feature requests and preliminary implementations.
- Application prologue and epilogue scripts can now be embedded in the GC3Pie configuration file, or referenced by file name.
- New selection options have been added to the *gselect: select job ids from from a session* command.
- *gc3libs.Configuration* will now raise different exceptions depending on whether no files could be read (`NoAccessibleConfigurationFile`) or could not be parsed (`NoValidConfigurationFile`).

Important bug fixes

- Shell metacharacters are now allowed in *Application* arguments. Each argument string is now properly quoted before passing it to the execution layer.
- LSF backend updated to work with *both* bjobs *and* bacct for accounting, or to parse information provided in the final output file as a last resort.
- All backends should now set a Task's *returncode* and *exitcode* values according to the documented meaning. Thanks to Y. Yakimovitch for reporting the issue.

GC3Pie 2.2

New features

- New `openstack` backend for running jobs on ephemeral VMs on OpenStack-compatible IaaS cloud systems. This is preferred over the OpenStack EC2 compatibility layer.
- New configurable scheduler for GC3Pie's `Engine`
- Session-based scripts can now snapshot the output of RUNNING jobs at every cycle.
- ARC backends are now deprecated: they will be removed in the next major version of GC3Pie.
- The `pbs` backend can now handle also Altair's PBSPro.

Changes to command-line utilities

- `gget`: New option *-A* to download output files of *all* tasks in a session.
- `gget`: New option `-c`/`--changed-only` to only download files that have apparently changed remotely.
- The GC3Apps collection has been enriched with several new applications.

Important bug fixes

- Working directory for remote jobs using the `shellcmd` backend is now stored in `/var/tmp` instead of `/tmp`, which should allow results to be retrieved even after a reboot of the remote machine.

GC3Pie 2.1

New features and incompatible changes

- GC3Pie now requires Python 2.6 or above to run.
- New `ec2` backend for running jobs on ephemeral VMs on EC2-compatible IaaS cloud systems.
- New package *gc3libs.optimizer* to find local optima of functions that can be computed through a job. Currently only implements the "Differential Evolution" algorithm, but the framework is generic enough to plug any genetic algorithm.
- New configuration options `prolog_content` and `epilog_content`, to allow execute oneliners before or after the command without having to create an auxiliary file.
- New `resourcedir` option for `shellcmd` resources. This is used to modify the default value for the directory containing job informations.

Changes to command-line utilities

- New command `gcloud` to interface with cloud-based VMs that were spawned by GC3Pie to run jobs.
- Table output now uses a different formatting (we use Python's *prettytable* package instead of the *texttable* package that we were using before, due to Py3 compatibility).

GC3Pie 2.0

New features and incompatible changes

- GC3Pie can now run on MacOSX.
- A session now has a configurable storage location, which can be a directory on the filesystem (FilesystemStore, the default so far) or can be a table in an SQL database (of any kind supported by SQLAlchemy).
- New ARC1 backend to use ARC resources through the new NorduGrid 1.x library API.
- New backend "subprocess": execute applications as local processes.
- New backends for running on various batch-queueing systems: SLURM, LSF, PBS.
- Implement recursive upload and download of directories if they are specified in an *Application*'s *input* or *output* attribute.
- New execution state *TERMINATING*: task objects are in this state when execution is finished remotely, but the task output has not yet been retrieved.
- Reorganize documentation and move it to http://gc3pie.readthedocs.org/
- Script logging is now controlled by a single configuration file *.gc3/gc3utils.log.conf*
- Session-based scripts now print WARNING messages to STDERR by default (previously, only ERROR messages were logged).

- Add caching to ARC backends, to reduce the number of network queries.
- Use GNU ".~NUMBER~" format for backup directories.

Configuration file changes

- Rename ARC0 resource type to *arc0*

Changes to command-line utilities

- New *gsession* command to manage sessions.
- The `glist` command was renamed to `gservers`
- The `gsub` and `gnotify` commands were removed.
- The `PATH` tag no longer gets any special treatment in session-based scripts `--output` processing.
- `ginfo`: New option `--tabular` to print information in table format.
- `gkill`: New option *-A*/*–all* to remove all jobs in a session.
- Use the *rungms* script to execute GAMESS.

API changes

- Module `gc3libs.dag` has been renamed to `gc3libs.workflow`.
- API changes in `gc3libs.cmdline.SessionBasedScript` allow *new_tasks()* in *SessionBasedScript* instances to return *Task* instances instead of quadruples.
- Interpret *Application.requested_memory* as the *total* memory for the job.
- the `Resource` and `LRMS` objects were merged
- the `gc3libs.scheduler` module has been removed; its functionality is now incorporated in the `Application` class.
- configuration-related code moved into *gc3libs.config* module
- removed the application registry.
- New package *gc3libs.compat* to provide 3rd-party functionality that is not present in all supported versions of Python.
- Implement *gc3libs.ANY_OUTPUT* to retrieve the full contents of the output directory, whatever it is.
- New *RetryableTask* class to wrap a task and re-submit it on failure until some specified condition is met.

GC3Pie 1.0

Configuration file changes

- Renamed configuration file to `gc3pie.conf`: the file `gc3utils.conf` will no longer be read!
- SGE clusters must now have `type = sge` in the configuration file (instead of `type = ssh-sge`)
- All computational resource must have an `architecture = ...` line; see the ConfigurationFile wiki page for details

- Probably more changes than it's worth to list here: check your configuration against the *Configuration File* page!

Command-line utilities changes

- GC3Utils and GC3Apps (`grosetta`/`ggamess`/etc.) now all accept a `-s`/`--session` option for locating the job storage directory: this allows grouping jobs into folders instead of shoveling them all into `~/.gc3/jobs`.
- GC3Apps: replaced option `-t`/`--table` with `-l`/`--states`. The new option prints a table of submitted jobs in addition to the summary stats; if a comma-separated list of job states follows the option, only job in those states are printed.
- Command `gstat` will now print a summary of the job states if the list is too long to fit on screen; use the -v option to get the full job listing regardless of its length.
- Command `gstat` can now print information on jobs in a certain state only; see help text for option `--state`
- Removed `-l` option from ginfo; use `-v` instead.
- GC3Utils: all commands accepting multiple job IDs on the command line, now exit with the number of errors/failures occurred. Since exit codes are practically limited to 7 bits, exit code 126 means that more than 125 failures happened.

GC3Pie 0.10

- First release for public use outside of GC3

2.2 Programmer Documentation

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.

The *Programming overview* section is the starting point for whoever wants to start developing applications with GC3Pie. It gives an overview of the main components of the library and how they interact with each other.

The Tutorials section contains documentation that describes in more detail the various components discussed in the programming overview, as well as many working examples (took from exercises done during the training events) and the *The "Warholize" Workflow Tutorial*: a step-by-step tutorial that will show you how to write a complex GC3Pie workflow.

The *GC3Libs programming API* section instead contains the API reference of GC3Pie library.

2.2.1 Programming overview

Computational job lifecycle

A computational job (for short: *job*) is a single run of a non-interactive application. The prototypical example is a run of GAMESS on a single input file.

The GC3Utils commands support the following workflow:

1. Submit a GAMESS job (with a single input file): **`ggamess`**
2. Monitor the status of the submitted job: **`gstat`**
3. Retrieve the output of a job once it's finished: **`gget`**

Usage and some examples on how to use the mentioned commands are provided in the next sections

Managing jobs with GC3Libs

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 a detailed description of the parameters.
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:
    1. 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.

    2. 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.

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*.

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.

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
UNKNOWN	Job info not found or lost track of job (e.g., network error or invalid job ID)	any other state
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 UNKNOWN state is a temporary error state: whenever GC3Pie is unable to get any information on the job, its state move to UNKNOWN. It is usually related to a (hopefully temporary) failure while accessing the remote resource, because of a network error or because the resource is not correctly configured. After the underlying cause of the error is fixed and GC3Pie is able again to get information on the job, its state will change to the proper state.

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.

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.

2.2.2 GC3Pie programming tutorials

Contents

Implementing scientific workflows with GC3Pie

This is the course material prepared for the "GC3Pie for Programmers" training, held at the University of Zurich for the first time on July 11-14, 2016. (The slides presented here are revised at each course re-run.)

The course aims at showing how to implement patterns commonly seen in scientific computational workflows using Python and GC3Pie, and provide users with enough knowledge of the tools available in GC3Pie to extend and adapt the examples provided.

Introduction to the training

A presentation of the training material and outline of the course. Probably not much useful unless you're actually sitting in class.

Overview of GC3Pie use cases

A quick overview of the kind of computational use cases that GC3Pie can easily solve.

GC3Pie basics

The basics needed to write simple GC3Pie scripts: the minimal session-based script scaffolding, and the properties and features of the `Application` object.

Useful debugging commands

Recall a few GC3Pie utilities that are especially useful when debugging code.

Customizing command-line processing

How to set up command-line argument and option processing in GC3Pie's `SessionBasedScript`

Application requirements

How to specify running requirements for `Application` tasks, e.g., how much memory is needed to run.

Application control and post-processing

How to check and react on the termination status of a GC3Pie Task/Application.

Introduction to workflows

A worked-out example of a many-step workflow.

Running tasks in a sequence

How to run tasks in sequence: basic usage of `SequentialTaskCollection` and `StagedTaskCollection`

Running tasks in parallel

How to run independent tasks in parallel: the `ParallelTaskCollection`

Automated construction of task dependency graphs

How to use the `DependentTaskCollection` for automated arrangement of tasks given their dependencies.

Dynamic and Unbounded Sequences of Tasks

How to construct `SequentialTaskCollection` classes that change the sequence of tasks while being run.

A bottom-up introduction to programming with GC3Pie

This is the course material made for the GC3Pie 2012 Training event held at the University of Zurich on October 1-2, 2012.

The presentation starts with low-level concepts (e.g., the `Application` and how to do manual task submission) and then gradually introduces more sophisticated tools (e.g., the `SessionBasedScript` and workflows).

This order of introducing concepts will likely appeal most to those already familiar with batch-computing and grid computing, as it provides an immediate map of the job submission and monitoring commands to GC3Pie equivalents.

Introduction to GC3Pie

Introduction to the software: what is GC3Pie, what is it for, and an overview of its features for writing high-throughput computing scripts.

Basic GC3Pie programming

The *Application* class, the smallest building block of GC3Pie. Introduction to the concept of Job, states of an application and to the *Core* class.

Application requirements

How to define extra requirements for an application, such as the minimum amount of memory it will use, the number of cores needed or the architecture of the CPUs.

Managing applications: the SessionBasedScript class

Introduction to the highest-level interface to build applications with GC3Pie, the *SessionBasedScript*. Information on how to create simple scripts that take care of the execution of your applications, from submission to getting back the final results.

The GC3Utils commands

Low-level tools to aid debugging the scripts.

Introduction to Workflows with GC3Pie

Using a practical example (the *The "Warholize" Workflow Tutorial*) we show how workflows are implemented with GC3Pie. The following slides will cover in more details the single steps needed to produce a complex workflow.

ParallelTaskCollection

Description of the *ParallelTaskCollection* class, used to run tasks in parallel.

StagedTaskCollection

Description of the *StagedTaskCollection* class, used to run a sequence of a fixed number of jobs.

SequentialTaskCollection

Description of the *SequentialTaskCollection* class, used to run a sequence of jobs that can be altered during runtime.

The "Warholize" Workflow Tutorial

In this tutorial we show how to use the GC3Pie libraries in order to build a command line script which runs a complex workflow with both parallelly- and sequentially-executing tasks.

The tutorial itself contains the complete source code of the application (see Literate Programming on Wikipedia), so that you will be able to test/modify it and produce a working `warholize.py` script by downloading the `pylit.py`:file: script from the PyLit Homepage and running the following command on the `docs/programmers/tutorials/warholize/warholize.rst` file, from within the source tree of GC3Pie:

```
$ ./pylit warholize.rst warholize.py
```

Introduction

Warholize is a GC3Pie demo application to produce, from a generic image picture, a new picture like the famous Warhol's work: Marylin. The script uses the powerful ImageMagick set of tools (at least version 6.3.5-7). This tutorial will assume that both *ImageMagick* and *GC3Pie* are already installed and configured.

In order to produce a similar image we have to do a series of transformations on the picture:

1) convert the original image to grayscale.

2) *colorize* the grayscale image using three different colors each time, based on the gray levels. We may, for instance, make all pixels with luminosity between 0-33% in red, pixels between 34-66% in yellow and pixels between 67% and 100% in green.

   To do that, we first have to:

   a) create a *Color Lookup Table* (LUT) using a combination of three randomly chosen colors

   b) apply the LUT to the grayscale image

3) Finally, we can merge together all the colorized images and produce our *warholized* image.

Clearly, step 2) depends on the step 1), and 3) depends on 2), so we basically have a sequence of tasks, but since step 2) need to create *N* different independent images, we can parallelize this step.

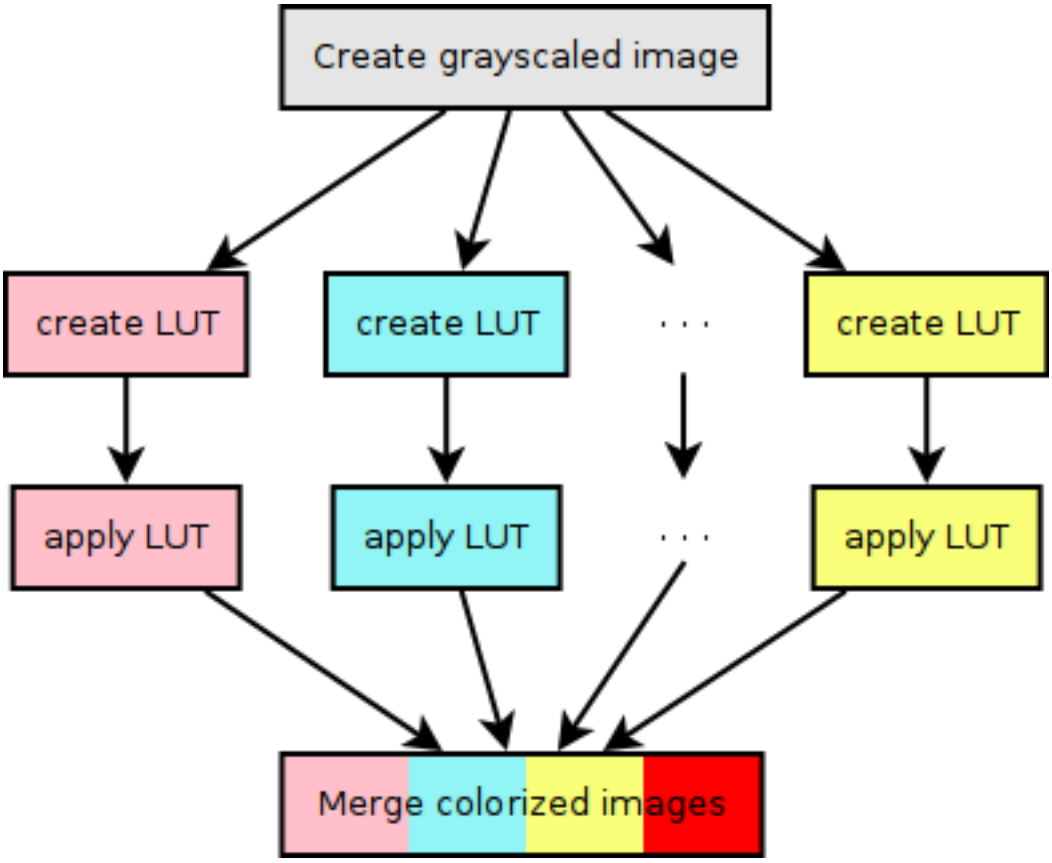

Fig. 1: Workflow of the *warholize* script

From top to bottom

We will write our script starting from the top and will descend to the bottom, from the command line script, to the workflow and finally to the single execution units which compose the application.

The script

The *SessionBasedScript* class in the *gc3libs.cmdline* module is used to create a generic script. It already have all what is needed to read gc3pie configuration files, manage resources, schedule jobs etc. The only missing thing is, well, your application!

Let's start by creating a new empty file and importing some basic modules:

```
import os
import gc3libs
from gc3libs.cmdline import SessionBasedScript
```

we then create a class which inherits from *SessionBasedScript* (in GC3Pie, most of the customizations are done by inheriting from a more generic class and overriding the *__init__* method and possibly others):

```
class WarholizeScript(SessionBasedScript):
    """
    Demo script to create a `Warholized` version of an image.
    """
    version='1.0'
```

Please note that you must either write a small docstring, or add a *description* attribute. These values are used when the script is called with options `--help` or `--version`, which are automatically added by GC3Pie.

The way we want to use our script is straightforward:

```
$  warholize.py inputfile [inputfiles ...]
```

and this will create a directory `Warholized.<inputfile>` in which there will be a file called `warhol_<inputfile>` containing the desired warholized image (and a lot of temporary files, at least for now).

But we may want to add some additional options to the script, in order to decide how many colorized pictures the warholized image will be made of, or if we want to resize the image. *SessionBasedScript* uses the PyCLI module which is, in turn, a wrapper around standard argparse (or optparse for older pythons) module. To customize the script you may define a *setup_options* method and put in there some calls to *SessionBasedScript.add_param()*, which is inherited from *cli.app.CommandLineApp*:

```
def setup_options(self):
    self.add_param('--copies', default=4, type=int,
                   help="Number of copyes (Default:4). It has to be a perfect square!
↪")
```

In this example we will accept a `--copies` option to define how many colorized copies the final picture will be made of. Please refer to the documentation of the PyCLI module for details on the syntax of the *add_param* method.

The *heart* of the script is, however, the *new_tasks* method, which will be called to create the initial tasks of the scripts. In our case it will be something like:

```
def new_tasks(self, extra):
    gc3libs.log.info("Creating main sequential task")
    for (i, input_file) in enumerate(self.params.args):
        extra_args = extra.copy()
        extra_args['output_dir'] = 'Warholized.%s' % os.path.basename(input_file)
        yield WarholizeWorkflow(input_file,
                                self.params.copies,
                                **extra_args)
```

new_tasks is used as a *generator* (but it could return a list as well). Each *yielded* object is a *task*. In GC3Pie, a *task* is either a single application or a complex workflow, and rapresents an *execution unit*. In our case we create a *WarholizeWorkflow* task which is the workflow described before.

In our case we yield a different *WarholizeWorkflow* task for each input file. These tasks will run in parallel.

Please note that we are using the *gc3libs.log* module to log informations about the execution. This module works like the logging module and has methods like *error*, *warning*, *info* or *debug*, but its logging level is automatically

configured by *SessionBasedScript*'s constructor. This way you can increase the verbosity of your script by simply adding `-v` options from the command line.

The workflows

Main sequential workflow

The module *gc3libs.workflow* contains two main objects: *SequentialTaskCollection* and *ParallelTaskCollection*. They execute tasks in serial and in parallel, respectively. We will use both of them to create our workflow; the first one, *WarholizeWorkflow*, is a sequential task, therefore we have to inherit from *SequentialTaskCollection* and customize its *__init__* method:

```
from gc3libs.workflow import SequentialTaskCollection, ParallelTaskCollection
import math
from gc3libs import Run

class WarholizeWorkflow(SequentialTaskCollection):
    """
    Main workflow.
    """

    def __init__(self, input_image,  copies, **extra_args):
        self.input_image = input_image
        self.output_image = "warhol_%s" % os.path.basename(input_image)

        gc3libs.log.info(
            "Producing a warholized version of input file %s "
            "and store it in %s" % (input_image, self.output_image))

        self.output_dir = os.path.relpath(extra_args.get('output_dir'))

        self.copies = copies

        # Check that copies is a perfect square
        if math.sqrt(self.copies) != int(math.sqrt(self.copies)):
            raise gc3libs.exceptions.InvalidArgument(
                "`copies` argument must be a perfect square.")

        self.jobname = extra_args.get('jobname', 'WarholizedWorkflow')
        self.grayscaled_image = "grayscaled_%s" % os.path.basename(self.input_image)
```

Up to now we just parsed the arguments. The following lines, instead, create the first task that we want to execute. By now, we can create only the first one, *GrayScaleConvertApplication*, which will produce a grayscale image from the input file:

```
self.tasks = [
    GrayScaleConvertApplication(
        self.input_image, self.grayscaled_image, self.output_dir,
        self.output_dir),
    ]
```

Finally, we call the parent's constructor.:

```
SequentialTaskCollection.__init__(
    self, self.tasks)
```

This will create the initial task list, but we have to run also step 2 and 3, and this is done by creating a *next* method. This method will be called after all the tasks in *self.tasks* are finished. We cannot create all the jobs at once because we don't have all the needed input files yet. Please note that by creating the tasks in the *next* method you could decide *at runtime* which tasks to run next and what arguments we may want to give to them.

In our case, however, the *next* method is quite simple:

```
def next(self, iteration):
    last = self.tasks[-1]

    if iteration == 0:
        # first time we got called.  We have the grayscaled image,
        # we have to run the Tricolorize task.
        self.add(TricolorizeMultipleImages(
            os.path.join(self.output_dir, self.grayscaled_image),
            self.copies, self.output_dir))
        return Run.State.RUNNING
    elif iteration == 1:
        # second time, we already have the colorized images, we
        # have to merge them together.
        self.add(MergeImagesApplication(
            os.path.join(self.output_dir, self.grayscaled_image),
            last.warhol_dir,
            self.output_image))
        return Run.State.RUNNING
    else:
        self.execution.returncode = last.execution.returncode
        return Run.State.TERMINATED
```

At each iteration, we call *self.add()* to add an instance of a task-like class (*gc3libs.Application*, *gc3libs.workflow.ParallelTaskCollection* or *gc3libs.workflow.SequentialTaskCollection*, in our case) to complete the next step, and we return the current state, which will be *gc3libs.Run.State.RUNNING* unless we have finished the computation.

Step one: convert to grayscale

GrayScaleConvertApplication is the application responsible to convert to grayscale the input image. The command we want to execute is:

```
$    convert -colorspace gray <input_image> grayscaled_<input_image>
```

To create a generic application we create a class which inherit from *gc3libs.Application* and we usually only need to customize the *__init__* method:

```
# An useful function to copy files
from shutil import copyfile

class GrayScaleConvertApplication(gc3libs.Application):
    def __init__(self, input_image, grayscaled_image, output_dir, warhol_dir):
        self.warhol_dir = warhol_dir
        self.grayscaled_image = grayscaled_image

        arguments = [
            'convert',
            os.path.basename(input_image),
            '-colorspace',
```

(continues on next page)

(continued from previous page)

```
            'gray',
            ]

        gc3libs.log.info(
            "Craeting  GrayScale convert application from file %s"
            "to file %s" % (input_image, grayscaled_image))

        gc3libs.Application.__init__(
            self,
            arguments = arguments + [grayscaled_image],
            inputs = [input_image],
            outputs = [grayscaled_image, 'stderr.txt', 'stdout.txt'],
            output_dir = output_dir,
            stdout = 'stdout.txt',
            stderr = 'stderr.txt',
            )
```

Creating a *gc3libs.Application* is straigthforward: you just call the constructor with the executable, the arguments, and the input/output files you will need.

If you don't specify the `output_dir` directory, gc3pie libraries will create one starting from the class name. If the output directory exists already, the old one will be renamed.

To do any kind of post processing you can define a *terminate* method for your application. It will be called after your application will terminate. In our case we want to copy the gray scale version of the image to the *warhol_dir*, so that it will be easily reachable by all other applications:

```
def terminated(self):
    """Move grayscale image to the main output dir"""
    copyfile(
        os.path.join(self.output_dir, self.grayscaled_image),
        self.warhol_dir)
```

Step two: parallel workflow to create colorized images

The *TricolorizeMultipleImages* is responsible to create multiple versions of the grayscale image with different coloration chosen randomly from a list of available colors. It does it by running multiple instance of *TricolorizeImage* with different arguments. Since we want to run the various colorization in parallel, it inherits from *gc3libs.workflow.ParallelTaskCollection* class. Like we did for *GrayScaleConvertApplication*, we only need to customize the constructor *__init__*, creating the various subtasks we want to run:

```
import itertools
import random

class TricolorizeMultipleImages(ParallelTaskCollection):
    colors = ['yellow', 'blue', 'red', 'pink', 'orchid',
              'indigo', 'navy', 'turquoise1', 'SeaGreen', 'gold',
              'orange', 'magenta']

    def __init__(self, grayscaled_image, copies, output_dir):
        gc3libs.log.info(
            "TricolorizeMultipleImages for %d copies run" % copies)
        self.jobname = "Warholizer_Parallel"
        ncolors = 3
        ### XXX Why I have to use basename???
```

(continues on next page)

(continued from previous page)

```
        self.output_dir = os.path.join(
            os.path.basename(output_dir), 'tricolorize')
        self.warhol_dir = output_dir

        # Compute a unique sequence of random combination of
        # colors. Please note that we can have a maximum of N!/3! if N
        # is len(colors)
        assert copies <= math.factorial(len(self.colors)) / math.factorial(ncolors)

        combinations = [i for i in itertools.combinations(self.colors, ncolors)]
        combinations = random.sample(combinations, copies)

        # Create all the single tasks
        self.tasks = []
        for i, colors in enumerate(combinations):
            self.tasks.append(TricolorizeImage(
                os.path.relpath(grayscaled_image),
                "%s.%d" % (self.output_dir, i),
                "%s.%d" % (grayscaled_image, i),
                colors,
                self.warhol_dir))

        ParallelTaskCollection.__init__(self, self.tasks)
```

The main loop will fill the *self.tasks* list with various *TricolorizedImage* tasks, each one with an unique combination of three colors to use to generate the colorized image. The GC3Pie framework will then run these tasks in parallel, on any available resource.

The *TricolorizedImage* class is indeed a *SequentialTaskCollection*, since it has to generate the LUT first, and then apply it to the grayscale image. We already saw how to create a *SequentialTaskCollection*: we modify the constructor in order to add the first job (*CreateLutApplication*), and the *next* method will take care of running the *ApplyLutApplication* application on the output of the first job:

```
class TricolorizeImage(SequentialTaskCollection):
    """
    Sequential workflow to produce a `tricolorized` version of a
    grayscale image
    """
    def __init__(self, grayscaled_image, output_dir, output_file,
                 colors, warhol_dir):
        self.grayscaled_image = grayscaled_image
        self.output_dir = output_dir
        self.warhol_dir = warhol_dir
        self.jobname = 'TricolorizeImage'
        self.output_file = output_file

        if not os.path.isdir(output_dir):
            os.mkdir(output_dir)

        gc3libs.log.info(
            "Tricolorize image %s to %s" % (
                self.grayscaled_image, self.output_file))

        self.tasks = [
            CreateLutApplication(
                self.grayscaled_image,
```

(continues on next page)

(continued from previous page)

```
                "%s.miff" % self.grayscaled_image,
                self.output_dir,
                colors, self.warhol_dir),
            ]

        SequentialTaskCollection.__init__(self, self.tasks)

    def next(self, iteration):
        last = self.tasks[-1]
        if iteration == 0:
            # First time we got called. The LUT has been created, we
            # have to apply it to the grayscale image
            self.add(ApplyLutApplication(
                self.grayscaled_image,
                os.path.join(last.output_dir, last.lutfile),
                os.path.basename(self.output_file),
                self.output_dir, self.warhol_dir))
            return Run.State.RUNNING
        else:
            self.execution.returncode = last.execution.returncode
            return Run.State.TERMINATED
```

The *CreateLutApplication* is again an application which inherits from *gc3libs.Application*. The command we want to execute is something like:

```
$    convert -size 1x1 xc:<color1> xc:<color2> xc:<color3> +append -resize 256x1!
↪<output_file.miff>
```

This will basically create an image 256x1 pixels big, made of a gradient using all the listed colors. The code will look like:

```
class CreateLutApplication(gc3libs.Application):
    """Create the LUT for the image using 3 colors picked randomly
    from CreateLutApplication.colors"""

    def __init__(self, input_image, output_file, output_dir, colors, working_dir):
        self.lutfile = os.path.basename(output_file)
        self.working_dir = working_dir
        gc3libs.log.info("Creating lut file %s from %s using "
                         "colors: %s" % (
            self.lutfile, input_image, str.join(", ", colors)))
        gc3libs.Application.__init__(
            self,
            arguments = [
                'convert',
                '-size',
                '1x1'] + [
                "xc:%s" % color for color in colors] + [
                '+append',
                '-resize',
                '256x1!',
                self.lutfile,
                ],
            inputs = [input_image],
            outputs = [self.lutfile, 'stdout.txt', 'stderr.txt'],
            output_dir = output_dir + '.createlut',
```

(continues on next page)

(continued from previous page)

```
            stdout = 'stdout.txt',
            stderr = 'stderr.txt',
            )
```

Similarly, the *ApplyLutApplication* application will run the following command:

```
$    convert grayscaled_<input_image> <lutfile.N.miff> -clut grayscaled_<input_image>.
↪<N>
```

This command will apply the LUT to the grayscaled image: it will modify the grayscaled image by *coloring* a generic pixel with a luminosity value of *n* (which will be an integer value from 0 to 255, of course) with the color at position *n* in the LUT image (actually, *n+1*). Each *ApplyLutApplication* will save the resulting image to a file named as `grayscaled_<input_image>.<N>`.

The class will look like:

```
class ApplyLutApplication(gc3libs.Application):
    """Apply the LUT computed by `CreateLutApplication` to
    `image_file`"""

    def __init__(self, input_image, lutfile, output_file, output_dir, working_dir):

        gc3libs.log.info("Applying lut file %s to %s" % (lutfile, input_image))
        self.working_dir = working_dir
        self.output_file = output_file

        gc3libs.Application.__init__(
            self,
            arguments = [
                'convert',
                os.path.basename(input_image),
                os.path.basename(lutfile),
                '-clut',
                output_file,
                ],
            inputs = [input_image, lutfile],
            outputs = [output_file, 'stdout.txt', 'stderr.txt'],
            output_dir = output_dir + '.applylut',
            stdout = 'stdout.txt',
            stderr = 'stderr.txt',
            )
```

The *terminated* method:

```
def terminated(self):
    """Copy colorized image to the output dir"""
    copyfile(
        os.path.join(self.output_dir, self.output_file),
        self.working_dir)
```

will copy the colorized image file in the top level directory, so that it will be easier for the last application to find all the needed files.

Step three: merge all them together

At this point we will have in the main output directory a bunch of files named after `grayscaled_<input_image>.N` with N a sequential integer and `<input_image>` the name of the original image. The last application, *MergeImagesApplication*, will produce a `warhol_<input_image>` image by merging all of them using the command:

```
$    montage grayscaled_<input_image>.* -tile 3x3 -geometry +5+5 -background white␣
↪warholized_<input_image>
```

Now it should be easy to write such application:

```
import re

class MergeImagesApplication(gc3libs.Application):
    def __init__(self, grayscaled_image, input_dir, output_file):
        ifile_regexp = re.compile(
            "%s.[0-9]+" % os.path.basename(grayscaled_image))
        input_files = [
            os.path.join(input_dir, fname) for fname in os.listdir(input_dir)
            if ifile_regexp.match(fname)]
        input_filenames = [os.path.basename(i) for i in input_files]
        gc3libs.log.info("MergeImages initialized")
        self.input_dir = input_dir
        self.output_file = output_file

        tile = math.sqrt(len(input_files))
        if tile != int(tile):
            gc3libs.log.error(
                "We would expect to have a perfect square"
                "of images to merge, but we have %d instead" % len(input_files))
            raise gc3libs.exceptions.InvalidArgument(
                "We would expect to have a perfect square of images to merge, but we␣
↪have %d instead" % len(input_files))

        gc3libs.Application.__init__(
            self,
            arguments = ['montage'] + input_filenames + [
                '-tile',
                '%dx%d' % (tile, tile),
                '-geometry',
                '+5+5',
                '-background',
                'white',
                output_file,
                ],
            inputs = input_files,
            outputs = [output_file, 'stderr.txt', 'stdout.txt'],
            output_dir = os.path.join(input_dir, 'output'),
            stdout = 'stdout.txt',
            stderr = 'stderr.txt',
            )
```

Making the script executable

Finally, in order to make the script *executable*, we add the following lines to the end of the file. The *WarholizeScritp().run()* call will be executed only when the file is run as a script, and will do all the magic related to argument parsing, creating the session etc. . . :

```
if __name__ == '__main__':
    import warholize
    warholize.WarholizeScript().run()
```

Please note that the `import warholize` statement is important to address issue 95 and make the gc3pie scripts work with your current session (*gstat*, *ginfo*. . .)

Testing

To test this script I would suggest to use the famous *Lena* picture, which can be found in the *miscelaneous* section of the Signal and Image Processing Institute page. Download the image, rename it as `lena.tiff` and run the following command:

```
$    ./warholize.py -C 1 lena.tiff --copies 9
```

(add `-r localhost` if your gc3pie.conf script support it and you want to test it locally).

After completion a file `Warholized.lena.tiff/output/warhol_lena.tiff` will be created.

Example scripts

A collection of small example scripts highlighting different features of GC3Pie is available in the source distribution, in folder `examples/`:file:

gdemo_simple.py

> Simplest script you can create. It only uses *Application* and *Engine* classes to create an application, submit it, check its status and retrieve its output.

grun.py

> a *SessionBasedScript* that executes its argument as command. It can also run it multiple times by wrapping it in a ParallelTaskCollection or a SequentialTaskCollection, depending on a command line option. Useful for testing a configured resource.

gdemo_session.py

> a simple *SessionBasedScript* that sums two values by customizing a *SequentialTaskCollection*.

warholize.py

> an enhanced version of the *warholize* script proposed in the *The "Warholize" Workflow Tutorial*

2.2.3 GC3Libs programming API

gc3libs

gc3libs.application

Fig. 2: Warholized version of *Lena*

gc3libs.defaults

gc3libs.events

gc3libs.exceptions

gc3libs.optimizer

gc3libs.optimizer.dif_evolution

gc3libs.optimizer.drivers

gc3libs.optimizer.extra

gc3libs.persistence

gc3libs.persistence.accessors

gc3libs.persistence.filesystem

gc3libs.persistence.idfactory

gc3libs.persistence.serialization

gc3libs.persistence.sql

gc3libs.persistence.store

gc3libs.poller

gc3libs.quantity

gc3libs.session

gc3libs.template

gc3libs.testing

gc3libs.testing.helpers

gc3libs.url

gc3libs.utils

gc3libs.workflow

gc3utils

gc3utils.commands

gc3utils.frontend

This is the main entry point for command **`gc3utils`** – a simple command-line frontend to distributed resources

This is a generic front-end code; actual implementation of commands can be found in `gc3utils.commands`

`gc3utils.frontend.`**`main`**`()`
Generic front-end function to invoke the commands in *gc3utils/commands.py*

2.3 Contributors documentation

This section contains information needed by people who want to contribute code to GC3Pie.

2.3.1 Contributing to GC3Pie

First of all, thanks for wanting to contribute to GC3Pie! GC3Pie is an open-ended endeavour, and we're always looking for new ideas, suggestions, and new code. (And also, for fixes to bugs old and new ;-))

The paragraphs below should brief you about the organization of the GC3Pie code repositories, and the suggested guidelines for code and documentation style. Feel free to request more info or discuss the existing recommendations on the GC3Pie mailing list

Code repository organization

GC3Pie code is hosted in a GitHub repository, which you can access online or using any Git client.

We encourage anyone to fork the repository and contribute back modifications in the form of pull requests.

The *master* branch should always be deployable: code in *master* should normally run without major known issues (but it may contain code that has yet not been released to PyPI). A tag is created on the *master* branch each time code is released to PyPI. Development happens on separate branches (or forks) which are then merged into *master* via pull requests.

Repository structure

The GC3Pie code repository has the following top-level structure; there is one subdirectory for each of the main parts of GC3Pie:

- The `gc3libs` directory contains the GC3Libs code, which is the core of GC3Pie. GC3Libs are extensively described in the *API* section of this document; read the module descriptions to find out where your new suggested functionality would suit best. If unsure, ask on the GC3Pie mailing list.
- The `gc3utils` directory contains the sources for the low-level GC3Utils command-line utilities.
- The `gc3apps` directory contains the sources for higher level scripts that implement some computational use case of independent interest.

  The `gc3apps` directory contains one subdirectory per *application script*. Actually, each subdirectory can contain one or more Python scripts, as long as they form a coherent bundle; for instance, Rosetta is a suite of applications in computational biology: there are different GC3Apps script corresponding to different uses of the Rosetta suite, all of them grouped into the `rosetta` subdirectory.

  Subdirectories of the `gc3apps` directory follow this naming convention:

  – the directory name is the main application name, if the application that the scripts wrap is a known, publicly-released computational application (e.g., Rosetta, GAMESS)

  - the directory name is the requestor's name, if the application that the scripts wrap is some research code that is being internally developed. For instance, the `bf.uzh.ch` directory contains scripts that wrap code for economic simulations that is being developed at the Banking and Finance Institute of the University of Zurich

Package generation

Due to issue 329, we don't use the automatic discovery *feature* of `setuptools`, so the files included in the distributed packages are those in the `MANIFEST.in` file, please check The MANIFEST.in template section of the python documentation for a syntax reference. We usually include only code, documentation, and related files. We also include the regression tests, but we **do not include** the application tests in `gc3apps/*/test` directories.

Testing the code

In developing GC3Pie we try to use a Test Driven Development approach, in the light of the quote: *It's tested or it's broken*. We use tox and pytest as test runners, which make creating tests very easy.

Running the tests

You can both run tests on your current environment using `pytest` or use *tox_* to create and run tests on separate environments. We suggest you to use `pytest` while you are still fixing the problem, in order to be able to run only the failing test, but we strongly suggest you to run `tox` *before* committing your code.

Running tests with `pytest`

In order to have the `pytest` program, you need to install *pytest_* in your current environment **and** gc3pie must be installed in develop mode:

```
pip install pytest
python setup.py develop
```

Then, from the top level directory, run the tests with:

```
pytest -v
```

PyTest will then crawl the directory tree looking for available tests. You can also specify a subset of the available sets, by:

- specifying the directory from which nose should start looking for tests:

```
# Run only backend-related tests
pytest -v gc3libs/backends/
```

- specifying the file containing the tests you want to run:

```
# Run only tests contained in a specific file
pytest -v gc3libs/tests/test_session.py
```

- specifying the id of the test (a test ID is the file name, a double colon, and the test function name):

```
# Run only test `test_engine_limits` in file `test_engine.py`
pytest test_engine.py::test_engine_limits
```

Running multiple tests

In order to test GC3Pie against multiple version of python we use tox, which creates virtual environments for all configured python version, runs pytest inside each one of them, and prints a summary of the test results.

You don't need to have `tox` installed in the virtual environment you use to develop gc3pie, you can create a new virtual environment and install `tox` on it with.

Running tox is straightforward; just type `tox` on the command-line in GC3Pie's top level source directory.

The default `tox.ini` file shipped with GC3Pie attempts to test all Python versions from 2.4 to 2.7 (inclusive). If you want to run tests only for a specific version of python, for instance Python 2.6, use the `-e` option:

```
tox -e py26
[...]
Ran 118 tests in 14.168s

OK (SKIP=9)
_____________________________________________________________ [tox summary] _____________
↪_____________________________________________
[TOX] py26: commands succeeded
[TOX] congratulations :)
```

Option `-r` instructs **`tox`** to re-build the testing virtual environment. This is usually needed when you update the dependencies of GC3Pie or when you add or remove command line programs or configuration files. However, if you feel that the environments can be *unclean*, you can clean up everything by:

1) deleting all the `*.pyc` file in your source tree:

```
find . -name '*.pyc' -delete
```

2) deleting and recreating tox virtual environments:

```
tox -r
```

Writing tests

Please remember that it may be hard to understand, whenever a test fails, if it's a bug in the code or in the tests! Therefore please remember:

- Try to keep tests as simple as possible, and *always* simpler than the tested code. (*Debugging is twice as hard as writing the code in the first place.*, Brian W. Kernighan and P. J. Plauger)
- Write multiple indipendent tests to test different possible behavior and/or different methods of a class.
- Tests should cover methods and functions, but also specific use cases.
- If you are fixing a bug, it's good practice to write a test to check if the bug is still there, in order to avoid to re-include the bug in the future.
- Tests should clean up every temporary file they create.

Writing tests is very easy: just create a file whose name begins with `test_`, then put in it some functions which name begins with `test_`; the pytest framework will automatically call each one of them. Moreover, pytest will run also any pytest which will be found in the code.

The module `gc3libs.testing` contains a few helpers that make writing GC3Pie tests easier.

Full documentation of the pytest framework is available at the pytest website.

Organizing tests

Each single python file should have a test file inside a `tests` subpackage with filename created by prefixing `test_` to the filename to test. For example, if you created a file `foo.py`, there should be a file `tests/test_foo.py` which will contains tests for `foo.py`.

Even though following the naming convention above is not always possible, each test regarding a specific component should be in a file inside a `tests` directory inside that component. For instance, tests for the subpackage *gc3libs.persistence* are located inside the directory `gc3libs/persistence/tests` but are not named after the specific file.

Coding style

Python code should be written according to 'PEP 8'_ recommendations. (And by this we mean not just the code style.)

Please take the time to read PEP 8 through, as it is widely-used across the Python programming community – it will benefit your contribution to any free/open-source Python project!

Anyway, here's a short summary for the impatient:

- use English nouns to name variables and classes; use verbs to name object methods.
- use 4 spaces to indent code; never use TAB characters.
- use lowercase letters for method and variable names; use underscores _ to separate words in multi-word identifiers (e.g., `lower_case_with_underscores`)
- use "CamelCase" for class and exception names.
- but, above all, do not blindly follow the rules and try to do the thing that *enhances code clarity and readability!*

Here's other code conventions that apply to GC3Pie code; since they are not always widely followed or known, a short rationale is given for each of them.

- Every class and function should have a docstring. Use reStructuredText markup for docstrings and documentation text files.

  Rationale: A concise English description of the purpose of a function can be faster to read than the code. Also, undocumented functions and classes do not appear in this documentation, which makes them invisible to new users.

- Use fully-qualified names for all imported symbols; i.e., write `import foo` and then use `foo.bar()` instead of `from foo import bar`. If there are few imports from a module, and the imported names do *clearly* belong to another module, this rule can be relaxed if this enhances readability, but *never* do use unqualified names for exceptions.

  Rationale: There are so many functions and classes in GC3Pie, so it may be hard to know to which module the function *count* belongs. (Think especially of people who have to bugfix a module they didn't write in the first place.)

- When calling methods or functions that accept both positional and optional arguments like:

```
def foo(a, b, key1=defvalue1, key2=defvalue2):
```

  always specify the argument name for optional arguments, which means **do not call**:

```
foo(1, 2, value1, value2)
```

  but **call instead**:

```
foo(1, 2, key1=value1, key2=value2)
```

Rationale: calling the function with explicit argument names will reduce the risk of hit some compatibility issues. It is perfectly fine, from the point of view of the developer, to change the signature of a function by swapping two different *optional* arguments, so this change can happen any time, although changing *positional* arguments will break backward compatibility, and thus it's usually well advertised and tested.

- Use double quotes `"` to enclose strings representing messages meant for human consumption (e.g., log messages, or strings that will be printed on the users' terminal screen).

  Rationale: The apostrophe character `'` is a normal occurrence in English text; use of the double quotes minimizes the chances that you introduce a syntax error by terminating a string in its middle.

- Follow normal typographic conventions when writing user messages and output; prefer clarity and avoid ambiguity, even if this makes the messages longer.

  Rationale: Messages meant to be read by users *will* be read by users; and if they are not read by users, they will be fired back verbatim on the mailing list on the next request for support. So they'd better be clear, or you'll find yourself wondering what that message was intended to mean 6 months ago.

  Common typographical conventions enhance readability, and help users identify lines of readable text.

- Use single quotes `'` for strings that are meant for internal program usage (e.g., attribute names).

  Rationale: To distinguish them visually from messages to the user.

- Use triple quotes `"""` for docstrings, even if they fit on a single line.

  Rationale: Visual distinction.

- Each file should have this structure:
  - the first line is the hash-bang line,
  - the module docstring (explain briefly the module purpose and features),
  - the copyright and licence notice,
  - module imports (in the order suggested by **PEP 8**)
  - and then the code...

  Rationale: The docstring should be on top so it's the first thing one reads when inspecting a file. The copyright notice is just a waste of space, but we're required by law to have it.

Documentation

The documentation can be found in gc3pie/docs. It is generated using Sphinx (http://sphinx-doc.org/contents.html).

GC3Pie documentation is divided in three sections:

- *User Documentation*: info on how to install, configure and run GC3Pie applications.
- *Programmer Documentation*: info for programmers who want to use the GC3Pie libraries to write their own scripts and applications.
- *Contributors documentation*: detailed information on how to contribute to GC3Pie and get your code included in the main library.

The *GC3Libs programming API <gc3libs_>* is the most relevant part of the docs for developers contributing code and is generated automatically from the docstrings inside the modules. Automatic documentation in Sphinx is described under http://sphinx-doc.org/tutorial.html#autodoc. While updating the docs of existing modules is simply done by running `make html`, adding documentation for a new module requires one of the following two procedures:

- Add a reference to the new module in `docs/programmers/api/index.rst`. Additionally, create a file that enables automatic documentation for the module. For the module `core.py`, for example, automatic documentation is enabled by a file `docs/programmers/api/gc3libs/core.rst` with the following content:

```
`gc3libs.core`
==============

.. automodule:: gc3libs.core
     :members:
```

- Execute the script `docs/programmers/api/makehier.sh`, which automates the above. Note that the `makehier.sh` script will re-create all `.rst` files for all GC3Pie modules, so check if there were some unexpected changes (e.g., with `git status`) before you commit!

Docstrings are written in reStructuredText format. To be able to cross-reference between differen objects in the documentation, you should be familiar with Sphinx domains in general and the Python domain in particular.

Questions?

Please write to the GC3Pie mailing list; we try to do our best to answer promptly.

2.4 Publications

This is an index of papers, slide decks, and other assorted material on GC3Pie. Most recent contributions are listed first.

If you would like to add your contributions here, please send a message to the GC3Pie mailing-list (or use the web interface).

2.4.1 GC3Pie overviews

The following slides and papers provide an overview of GC3Pie and its features. (Most recent entries first.)

- GC3Pie: orchestrating large-scale execution of scientific applications. Presentation of GC3Pie and its features, especially focusing on the GC3Pie backend for EasyBuild. Held at the HPC-CH forum. June 11, 2015.
- GC3Pie: orchestrating large-scale execution of scientific applications. Presentation of GC3Pie and its features at the HPC-CH forum. June 18, 2014.
- Presentation of GC3Pie and GC3Libs (PDF) at the a private meeting with the developers of iBRAIN2/3 (later renamed to screeningBee), May 2012.
- GC3Pie: A Python framework for high-throughput computing (MAFFIOLETTI, Sergio, and Riccardo Murri). Proceedings of the EGI Community Forum 2012/EMI Second Technical Conference (EGICF12-EMITC2). 26-30 March, 2012. Munich, Germany. Published online at, id. 143. Vol. 1. 2012.
- Presentation of GC3Pie and GC3Libs (PDF) at the NorduGrid conference 2011.

2.4.2 Programming examples

The following slides focus on GC3Libs programming. (Most recent entries first.)

- Global Optimization with GC3Pie (PDF). Poster presented at the EuroSciPy 2013 conference, introducing the GC3Pie numerical optimizer.

- Computational workflows with GC3Pie (PDF). Poster presented at the EGI Community Forum 2012.
- Computational workflows with GC3Pie (PDF). Poster presented at the EuroSciPy 2011 conference.
- Introduction to GC3Pie and its programming model (HTML, PDF); slides presented at the Advanced School on High Performance and Grid Computing at ICTP Trieste.
- Introduction to GC3Libs programming (HTML, PDF); slides presented at the SMSCG Project meeting 2011.
- Introduction to GC3Libs programming (HTML, PDF); slides presented at a private meeting with the Selectome developers.

2.4.3 Use of GC3Pie in scientific applications

These papers and presentations cover specific tools built on top of GC3Pie, or applications of GC3Pie to research in a specific domain. (Most recent entries first.)

- TRAL: Tandem repeat annotation library (Schaper, E., Korsunsky, A., Messina, A., Murri, R., Pečerska, J., Stockinger, H., ... & Anisimova, M.). Bioinformatics, btv306 (2015).
- Selectome update: quality control and computational improvements to a database of positive selection (Moretti, S., Laurenczy, B., Gharib, W. H., Castella, B., Kuzniar, A., Schabauer, H., ... & Robinson-Rechavi, M.). Nucleic acids research, 42(D1), D917-D921, 2014.
- Wireless Mesh Networks and Cloud Computing for Real Time Environmental Simulations (Kropf, P., Schiller, E., Brunner, P., Schilling, O., Hunkeler, D., & Lapin, A.). In Recent Advances in Information and Communication Technology (pp. 1-11). Springer International Publishing. * Real-Time Environmental Monitoring for Cloud-Based Hydrogeological Modeling with HydroGeoSphere (Lapin, A., Schiller, E., Kropf, P., Schilling, O., Brunner, P., Kapic, A. J., ... & Maffioletti, S.). In High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS), 2014 IEEE Intl Conf on (pp. 959-965). IEEE, August 2014.
- Thermal and Hydrological Response of Rock Glaciers to Climate Change: A Scenario Based Simulation Study (Apaloo, J.). University of Waterloo, Canada, 2014.
- Parameter estimation of complex mathematical models of human physiology using remote simulation distributed in scientific cloud (Kulhanek, T., Mateják, M., Silar, J., & Kofranek, J.). In Biomedical and Health Informatics (BHI), 2014 IEEE-EMBS International Conference on (pp. 712-715). IEEE, June 2014.
- Towards a swiss national research infrastructure (Kunszt, P., Maffioletti, S., Flanders, D., Eurich, M., Bohnert, T., Edmonds, A., ... & Schiller, E.). arXiv preprint arXiv:1404.7608.
- User Interaction and Data Management for Large Scale Grid Applications. Journal of Grid Computing (Costantini, A., Gervasi, O., Zollo, F., & Caprini, L.), 12(3), 485-497, 2014.
- Application of large-scale computing infrastructure for diverse environmental research applications using GC3Pie (Maffioletti, S., Dawes, N., Bavay, M., Sarni, S., & Lehning, M.). In EGU General Assembly Conference Abstracts (Vol. 15, p. 13222). April 2013.
- gcodeml: A Grid-enabled Tool for Detecting Positive Selection in Biological Evolution (Moretti, S., Murri, R., Maffioletti, S., Kuzniar, A., Castella, B., Salamin, N., ... & Stockinger, H.). Studies in health technology and informatics, 175, 59-68 (2012).
- A Grid execution model for Computational Chemistry Applications using the GC3Pie framework and AppPot (Costantini, A., Murri, R., Maffioletti, S., Rampino, S., & Laganà, A.). Computational Science and Its Applications–ICCSA 2012. Springer Berlin Heidelberg, 2012. 401-416.
- Running GAMESS jobs with ggamess. Slides presented at a Baldridge Research Group meeting, Sept 2012.

- The MP2 binding energy of the ethene dimer and its dependence on the auxiliary basis sets: a benchmark study using a newly developed infrastructure for the processing of quantum chemical data (Glöß, A., Brändle, M. P., Klopper, W., & Lüthi, H. P.). Molecular Physics, 110(19-20), 2523-2534 (2012).
- Three tools for high-throughput computing with GAMESS (PDF). Slides presented at a Baldridge Research Group meeting, May 2011.
- Enabling High-Throughput Computational Chemistry on the Grid (PDF). Poster presented at the EGI User Forum 2011.
- GRunDB: a tool for validating QM algorithms in GAMESS-US (PDF). Slides presented at the Swiss Grid Day 2010.
- GC3Pie and related tools for high-throughput computational chemistry (PowerPoint PPT slides). Presentation held at the Databases in Quantum Chemistry workshop, September 22-25, 2010 in Zaragoza, Spain.

2.5 List of contributors to GC3Pie

This is a list of people that have contributed to GC3Pie, in any form: be it enhancements to the code or testing out releases and new features, or simply contributing suggestions and proposing enhancements. To them all, our gratitude for trying to make GC3Pie a better tool.

The list is sorted by last name. Please send an email to <*gc3pie-dev@googlegroups.com*> for corrections.

- Tyanko Aleksiev <tyanko.alexiev@gmail.com>
- Niko Ehrenfeuchter <nikolaus.ehrenfeuchter@unibas.ch>
- Benjamin Jonen <benjamin.jonen@gmail.com>
- Sergio Maffioletti <sergio.maffioletti@gc3.uzh.ch>
- Daniel McDonald <daniel.mcdonald@uzh.ch>
- Antonio Messina <arcimboldo@gmail.com>
- Mark Monroe <markjmonroe@yahoo.com>
- Riccardo Murri <riccardo.murri@gmail.com>
- Tom Osika <tom.osika@kitware.com>
- Michael Packard <mrghort@gmail.com>
- Xin Zhou <xin.zhou1983@gmail.com>

2.6 Glossary

API Acronym of *Application Programming Interface*. An API is a description of the way one piece of software asks another program to perform a service (quoted from: http://www.computerworld.com/s/article/43487/Application_Programming_Interface which see for a more detailed explanation).

Command-line The sequence of words typed at the terminal prompt in order to run a specified application.

Command-line option Arguments to a command (i.e., words on the command line) that select variants to the usual behavior of the command. For instance, a command-line option can request more verbose reporting.

Traditionally, UNIX command-line options consist of a dash (−), followed by one or more lowercase letters, or a double-dash (−−) followed by a complete word or compound word.

For example, the words `-h` or `--help` usually instruct a command to print a short usage message and exit immediately after.

Core A single computing unit. This was called a *CPU* until manufacturers started packing many processing units into a single package: now the term CPU is used for the package, and *core* is one of the several independent processing units within the package.

CPU Time The total time that computing units (processor *core*) are actively executing a *job*. For single-threaded jobs, this is normally *less* then the actual duration ('wall-clock time' or *walltime*), because some time is lost in I/O and system operations. For parallel jobs the CPU time is normally larger than the duration, because several processor cores are active on the job at the same time; the quotient of the CPU time and the duration measures the efficiency of the parallel job.

Job A computational job is a single run of a non-interactive application. The prototypical example is a run of GAMESS on a single input file.

Persistent Used in the sense of *preserved across program stops and system reboots*. In practice, it just means that the relevant data is stored on disk or in some database.

Resource Short for *computational resource*: any cluster or Grid where a job can run.

State A one-word indication of a computational *job* execution status (e.g., `RUNNING` or `TERMINATED`). The terms *state* and *status* are used interchangeably in GC3Pie documentation.

STDERR Abbreviation for "standard error stream"; it is the sequence of all text messages that a command prints to inform the user of problems or to report on operations progress. The Linux/UNIX system allows two separate output streams, one for output proper, named *STDOUT*, and STDERR for "error messages". It is entirely up to the command to tag a message as "standard output" or "standard error".

STDOUT Abbreviation for "standard output stream". It is the sequence of all characters that constitute the output of a command. The Linux/UNIX system allows two separate output streams, one for output proper, and one for "error messages", dubbed *STDERR*. It is entirely up to the command to tag a message as "standard output" or "standard error".

Session A *persistent* collection of GC3Pie tasks and jobs. Sessions are used by *The GC3Apps software* to store job status across program runs. A session is specified by giving the filesystem path to a *session directory*: the directory contains some files with meta-data about the tasks that comprise the session. It is also possible to *simulate* a session by specifying a *task store URL* (path to a filesystem directory where the jobs are stored, or connection URL to a database); in this case the session meta-data will be reconstructed from the set of tasks in the store.

Walltime Short for *wall-clock time*: indicates the total running time of a *job*.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index