GC3Pie basics

Riccardo Murri <riccardo.murri@uzh.ch>

S3IT: Services and Support for Science IT

University of Zurich

July 11–14, 2016

Concepts and glossary

Parts of GC3Pie

GC3Pie consists of three main components:

GC3Libs:
Python library for controlling the life-cycle of computational job collections.

GC3Utils:
This is a small set of low-level utilities exposing the main functionality provided by GC3Libs.

GC3Apps:
A collection of driver scripts to run large job campaigns.

GC3Pie glossary: Application

GC3Pie runs user applications
on clusters and IaaS cloud resources

An `Application` is just a command to execute.

GC3Pie glossary: Application

GC3Pie runs user applications
on clusters and IaaS cloud resources

An `Application` is just a command to execute.

If you can run it in the terminal,
you can run it in GC3Pie.

GC3Pie glossary: Application

GC3Pie runs user applications
on clusters and IaaS cloud resources

An `Application` is just a command to execute.

A single execution of an `Application`
is indeed called a `Run`.

(Other systems might call this a “job”.)

GC3Pie glossary: Task

GC3Pie runs user applications
on clusters and IaaS cloud resources

More generally, GC3Pie runs `Task`s.

`Task`s are a superset of applications,
in that they include workflows.

GC3Pie glossary: Resources

GC3Pie runs user applications
on clusters and IaaS cloud resources

`Resource`s are the computing infrastructures where GC3Pie executes applications.

Resources include: your laptop, the “Hydra” cluster, the Science Cloud, Amazon AWS.

Workflow scaffolding

Let's start coding!

```
from gc3libs.cmdline \
  import SessionBasedScript

if __name__ == '__main__':
  import ex2a
  ex2a.AScript().run()

class AScript(SessionBasedScript):
  """
  Minimal workflow scaffolding.
  """
  def __init__(self):
    super(AScript, self).__init__(
        version='1.0')
  def new_tasks(self, extra):
    return []
```

Download this code into a file named `ex2a.py`

Open it in your favorite text editor.

Exercise 2.A:

Download this code into a file named `ex2a.py`

1. Run the following command:

   ```
   > python ex2a.py --help
   ```

   Where does the program description in the help text come from? Is there anything weird in other parts of the help text?

2. Run the following command:

   ```
   > python ex2a.py
   ```

   What happens?

```python
from gc3libs.cmdline \
  import SessionBasedScript

if __name__ == '__main__':
  import ex2a
  ex2a.AScript().run()

class AScript(SessionBasedScript):
  """
  Minimal workflow scaffolding.
  """
  def __init__(self):
    super(AScript, self).__init__(
        version='1.0')
  def new_tasks(self, extra):
    return []
```

These lines are needed in every session-based script.

See issue 95 for details.

```python
from gc3libs.cmdline \
  import SessionBasedScript

if __name__ == '__main__':
  import ex2a
  ex2a.AScript().run()

class AScript(SessionBasedScript):
  """
  Minimal workflow scaffolding.
  """
  def __init__(self):
    super(AScript, self).__init__(
        version='1.0')
  def new_tasks(self, extra):
    return []
```

For this to work, it is **needed** that this is the actual file name.

```
from gc3libs.cmdline \
  import SessionBasedScript

if __name__ == '__main__':
  import ex2a
  ex2a.AScript().run()

class AScript(SessionBasedScript):
  """
  Minimal workflow scaffolding.
  """
  def __init__(self):
    super(AScript, self).__init__(
        version='1.0')
  def new_tasks(self, extra):
    return []
```

This is the program's help text!

```python
from gc3libs.cmdline \
  import SessionBasedScript

if __name__ == '__main__':
  import ex2a
  ex2a.AScript().run()

class AScript(SessionBasedScript):
  """
  Minimal workflow scaffolding.
  """
  def __init__(self):
    super(AScript, self).__init__(
      version='1.0')
  def new_tasks(self, extra):
    return []
```

A version number is **mandatory**.

```
from gc3libs.cmdline \
  import SessionBasedScript

if __name__ == '__main__':
  import ex2a
  ex2a.AScript().run()

class AScript(SessionBasedScript):
  """
  Minimal workflow scaffolding.
  """
  def __init__(self):
    super(AScript, self).__init__(
        version='1.0')
  def new_tasks(self, extra):
    return []
```

This is the core of the script.

Return a list of `Application` objects, that GC3Pie will execute.

The `Application` object

Specifying commands to run, I

You need to “describe” an application to GC3Pie, in order for GC3Pie to use it.

This “description” is a blueprint from which many actual command instances can be created.

(A few such “descriptions” are already part of the core library.)

GC3Pie application model

In GC3Pie, an application “description” is an object of the `gc3libs.Application` class (or subclasses thereof).

At a minimum: provide application-specific command-line invocation.

Advanced users can customize pre- and post-processing, react on state transitions, set computational requirements based on input files, influence scheduling. (This is standard OOP: subclass and override a method.)

A basic example: grayscaling

```
$ convert lena.jpg -colorspace gray lena-gray.jpg
```

Grayscaling example, I

Here is how you would tell GC3Pie to run that command-line.

```
from gc3libs import Application

class GrayscaleApp(Application):
  """Convert an image file to grayscale."""
  def __init__(self, img):
    out = "gray-" + basename(img)
    Application.__init__(
      self,
      arguments=[
        "convert", img, "-colorspace", "gray", out],
      inputs=[img],
      outputs=[out],
      output_dir="grayscale.d",
      stdout="stdout.txt")
```

Always inherit from Application

Your application class must inherit from class `gc3libs.Application`

```
from gc3libs import Application

class GrayscaleApp(Application):
  """Convert an image file to grayscale."""
  def __init__(self, img):
    out = "gray-" + basename(img)
    Application.__init__(
      self,
      arguments=[
        "convert", img, "-colorspace", "gray", out],
      inputs=[img],
      outputs=[out],
      output_dir="grayscale.d",
      stdout="stdout.txt")
```

The `arguments` parameter, I

The `arguments=` parameter is the actual command-line to be invoked.

```
class GrayscaleApp(Application):
  """Convert an image file to grayscale."""
  def __init__(self, img):
    out = "gray-" + basename(img)
    Application.__init__(
      self,
      arguments=[
        "convert", img, "-colorspace", "gray", out],
      inputs=[img],
      outputs=[out],
      output_dir="grayscale.d",
      stdout="stdout.txt")
```

The `arguments` parameter, II

The first item in the `arguments` list is the name or path to the command to run.

```
class GrayscaleApp(Application):
  """Convert an image file to grayscale."""
  def __init__(self, img):
    out = "gray-" + basename(img)
    Application.__init__(
      self,
      arguments=[
        "convert", img, "-colorspace", "gray", out],
      inputs=[img],
      outputs=[out],
      output_dir="grayscale.d",
      stdout="stdout.txt")
```

The `arguments` parameter, III

The rest of the list are arguments to the program, as you would type them at the shell prompt.

```
class GrayscaleApp(Application):
  """Convert an image file to grayscale."""
  def __init__(self, img):
    out = "gray-" + basename(img)
    Application.__init__(
      self,
      arguments=[
        "convert", img, "-colorspace", "gray", out ],
      inputs=[img],
      outputs=[out],
      output_dir="grayscale.d",
      stdout="stdout.txt")
```

The `inputs` parameter, I

The `inputs` parameter holds a list of files that you want to *copy* to the location where the command is executed. (Remember: this might be a remote computer!)

```
class GrayscaleApp(Application):
  """Convert an image file to grayscale."""
  def __init__(self, img):
    out = "gray-" + basename(img)
    Application.__init__(
      self,
      arguments=[
        "convert", img, "-colorspace", "gray", out],
      inputs=[img],
      outputs=[out],
      output_dir="grayscale.d",
      stdout="stdout.txt")
```

The `inputs` parameter, II

Input files retain their name during the copy, but not the entire path.

For example:

```
inputs = [
  '/home/rmurri/values.dat',
  '/home/rmurri/stats.csv',
  ]
```

will make files *values.dat* and *stats.csv* available in the command execution directory.

The `inputs` parameter, III

You need to pass the full path name into the `inputs` list, but use only the “base name” in the command invocation.

```
class GrayscaleApp(Application):
  """Convert an image file to grayscale."""
  def __init__(self, img):
    inp = basename(img)
    out = "gray-" + inp
    Application.__init__(
      self,
      arguments=[
        "convert", inp , "-colorspace", "gray", out],
      inputs=[ img ],
      outputs=[out],
      output_dir="grayscale.d",
      stdout="stdout.txt")
```

The `outputs` parameter, I

The `outputs` argument list files that should be copied from the command execution directory back to your computer.

```
class GrayscaleApp(Application):
  """Convert an image file to grayscale."""
  def __init__(self, img):
    inp = basename(img)
    out = "gray-" + inp
    Application.__init__(
      self,
      arguments=[
        "convert", inp, "-colorspace", "gray", out],
      inputs=[img],
      outputs=[out],
      output_dir="grayscale.d",
      stdout="stdout.txt")
```

The `outputs` parameter, II

Output file names are *relative to the execution directory*. For example:

```
outputs = ['result.dat', 'program.log']
```

(Contrast with input files, which must be specified by *absolute path*, e.g., `/home/rmurri/values.dat`)

Any file with the given name that is found in the execution directory will be copied back. (*Where?* See next slides!)

If an output file is *not* found, this is *not* an error. In other words, **output files are optional**.

The `output_dir` parameter, I

The `output_dir` parameter specifies where output filess will be downloaded.

```
class GrayscaleApp(Application):
  """Convert an image file to grayscale."""
  def __init__(self, img):
    inp = basename(img)
    out = "gray-" + inp
    Application.__init__(
      self,
      arguments=[
        "convert", inp, "-colorspace", "gray", out],
      inputs=[img],
      outputs=[out],
      output_dir="grayscale.d",
      stdout="stdout.txt")
```

The *output_dir* parameter, II

By default, GC3Pie does not overwrite an existing output directory: it will move the existing one to a backup name.

So, if `grayscale.d` already exists, GC3Pie will:

1. rename it to `grayscale.d.~1~`
2. create a new directory `grayscale.d`
3. download output files into the new directory

The `stdout` parameter

This specifies that the command's `standard output` should be saved into a file named `stdout.txt` and retrieved along with the other output files.

```
class GrayscaleApp(Application):
  """Convert an image file to grayscale."""
  def __init__(self, img):
    inp = basename(img)
    out = "gray-" + inp
    Application.__init__(
      self,
      arguments=[
        "convert", inp, "-colorspace", "gray", out],
      inputs=[img],
      outputs=[out],
      output_dir="grayscale.d",
      stdout="stdout.txt")
```

(The `stderr` parameter)

There's a corresponding `stderr` option for the command's *standard error* stream.

```
class GrayscaleApp(Application):
  """Convert an image file to grayscale."""
  def __init__(self, img):
    inp = basename(img)
    out = "gray-" + inp
    Application.__init__(
      self,
      arguments=[
        "convert", inp, "-colorspace", "gray", out],
      inputs=[img],
      outputs=[out],
      output_dir="grayscale.d",
      stdout="stdout.txt",
      stderr="stderr.txt" )
```

Mixing `stdout` and `stderr` capture

You can specify **either one** of the `stdout` and `stderr` parameters, **or both**.

If you give both, and they have the same value, then `stdout` and `stderr` will be intermixed just as they are in normal screen output.

Let's run!

In order for a session-based script to execute something, its `new_tasks()` method must return a list of `Application` objects to run.

```
class AScript(SessionBasedScript):
  # ...
  def new_tasks(self, extra):
    # 'self.param.args' is the list
    # of command-line arguments
    input_file = self.params.args[0]
    app = GrayscaleApp(input_file)
    return [app]
```

Exercise 2.B:

Edit the `ex2a.py` file: insert the code to define the `GrayscaleApp` application, and modify the `new_tasks()` method to return one instance of it (as in the previous slide).

Can you convert the `lena.jpg` file to gray-scale using this GC3Pie script?

(You can download the code for `GrayscaleApp` and the "Lena" image file from this URL.)

Exercise 2.C:

Edit the script from Exercise 2.B above and add the ability to convert multiple files: for each file name given on the command line, an instance of `GrayscaleApp` should be run.

Application lifecycle

```
$ ./grayscale.py lena.jpg
[...]
        NEW   1/1   (100.0%)
    RUNNING   0/1    (0.0%)
    STOPPED   0/1    (0.0%)
  SUBMITTED   0/1    (0.0%)
 TERMINATED   0/1    (0.0%)
TERMINATING   0/1    (0.0%)
    UNKNOWN   0/1    (0.0%)
      total   1/1   (100.0%)
```

`Application` objects can be in one of several states.

(A session-based script prints a table of all managed applications and their states.)

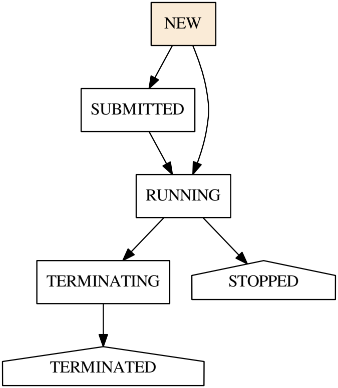
```
>>> print(app.execution.state)
'TERMINATED'
```

The current state is stored in the `.execution.state` instance attribute.

Reference:

http://gc3pie.readthedocs.io/en/master/programmers/api/gc3libs.html#gc3libs.Run.state

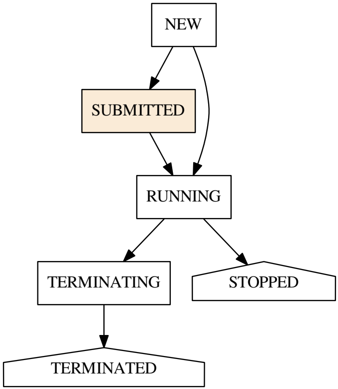
Application lifecycle: state NEW



NEW is the state of “just created” Application objects.

The Application has not yet been sent off to a compute resource: it only exists locally.

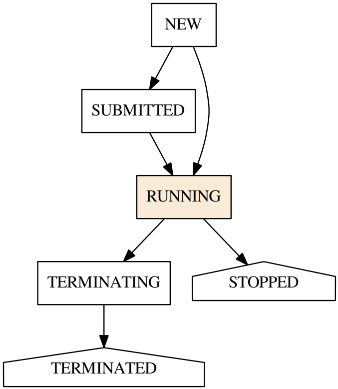
Application lifecycle: state SUBMITTED



SUBMITTED applications have been successfully sent to a computational resource.

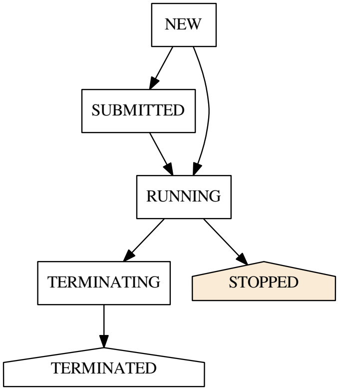
(The transition to *RUNNING* happens automatically, as we do not control the remote execution.)

Application lifecycle: state RUNNING



RUNNING state happens when the computational job associated to an application starts executing on the computational resource.

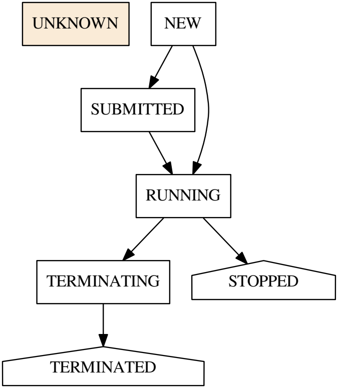
Application lifecycle: state STOPPED



A task is in *STOPPED* state when its execution has been blocked at the remote site and GC3Pie cannot recover automatically.

User or sysadmin intervention is required for a task to get out of *STOPPED* state.

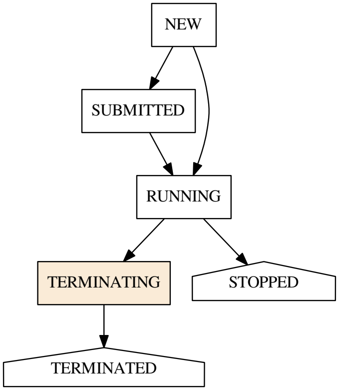
Application lifecycle: state UNKNOWN



A task is in *UNKNOWN* state when GC3Pie can no longer monitor it at the remote site.

(As this might be due to network failures, jobs *can* get out of *UNKNOWN* automatically.)

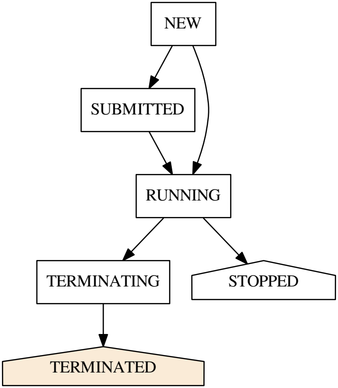
Application lifecycle: state TERMINATING



TERMINATING state when a computational job has finished running, for whatever reason.

(Transition to *TERMINATED* only happens when `fetch_output` is called.)

Application lifecycle: state TERMINATED



A job is *TERMINATED* when its final output has been retrieved and is available locally.

The exit code of *TERMINATED* jobs can be inspected to find out whether the termination was successful or unsuccessful, or if the program was forcibly ended.

Post-processing features, I

When the remote computation is done, the `terminated` method of the application instance is called.

The path to the output directory is available as `self.output_dir`; if `stdout` and `srderr` have been captured, the paths to the capture files are available as `self.stdout` and `self.stderr`.

Post-processing features, II

For example, the following code logs a warning message if the standard error output is non-empty:

```
class MyApp(Application):
  # ...
  def terminated(self):
    error_file = self.output_dir+"/"+self.stderr
    error_size = os.stat(error_file).st_size
    if error_size > 0:
      gc3libs.log.warn(
        "Application %s reported errors!", self)
```

Exercise 2.D:
Modify the `GrayscaleApp` application to print a message "`Conversion of '`*`filename`*`' done.`" whenever running the `convert` program terminates.

A successful run or not?

There's a *single TERMINATED state*, whatever the task outcome. You have to inspect the "return code" to determine the cause of "task death".

Attribute '.execution.returncode' provides a numeric termination status (with the same format and meaning as the POSIX termination status).

The termination status combines two fields: the "termination signal" and the "exit code".

Termination signal, I

The `.execution.signal` instance attribute is non-zero if the program was killed by a signal (e.g., memory error / segmentation fault).

The `.execution.signal` instance attribute is zero only if the program run until termination. (**Beware!** This does not mean that it run *correctly*: just that it halted by itself.)

Termination signal, II

Read `man 7 signal` for a list of OS signals and their numeric values.

Note that GC3Pie overloads some signal codes (unused by the OS) to represent its own specific errors.

For instance, if program `app` was cancelled by the user, `.execution.signal` will take the value 121:

```
>>> print(app.execution.signal)
121
```

Exit code

The `.execution.exitcode` instance attribute holds the numeric exitcode of the executed command, or `None` if the command has not finished running yet.

Note that the `.execution.exitcode` is guaranteed to have a valid value only if the `.execution.signal` attribute has the value 0.

The `.execution.exitcode` is the same exitcode that you would see when running a command directly in the terminal shell. (By convention, code 0 is successful termination, every other value indicates an error.)

Exercise 2.E:

Write a `TermStatusApp` application, which is like a generic `Application` class with the addition that —upon termination— it prints:

- whether the program has been killed by a signal, and the signal number;
- whether the program has terminated by exiting, and the exit code.

Verify that it works by plugging the class into the "grayscale" session-based script.

Exercise 2.F: *(Difficult)*
MATLAB has the annoying habit of exiting with code 0 even when some error occurred.

Write a `MatlabApp` application, which:

- is constructed by giving the path to a MATLAB '`.m`' script file, like this: `app = MatlabApp("ra.m");`
- Runs the following command:

  `matlab -nosplash -nodesktop -nojvm` *`file.m`*

  where *file.m* is the file given to the `MatlabApp()` constructor.
- captures the standard error output (`stderr`) of the MATLAB script and, if the string "`Out of memory.`" occurs in it, sets the application exitcode to `11`.

Verify that it works by running a MATLAB script that allocates an array of random size. (For some random values, the size will exceed the amount of available memory.)