



University of
Zurich ^{UZH}

S3IT

Application control and post-processing

Riccardo Murri <riccardo.murri@uzh.ch>

S3IT: Services and Support for Science IT

University of Zurich

Application run states

Application lifecycle

```
$ ./grayscale.py bfly.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.)

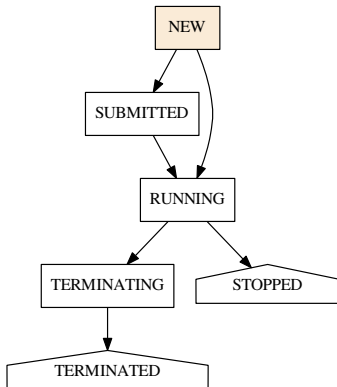
```
>>> print (app.execution.state)
' TERMINATED '
```

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>

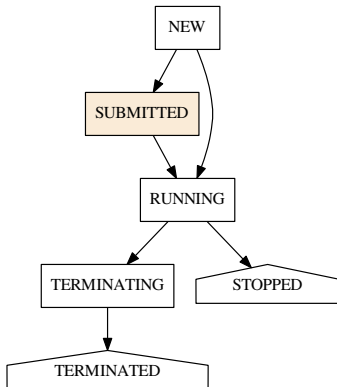
Application lifecycle: state **NEW**



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.

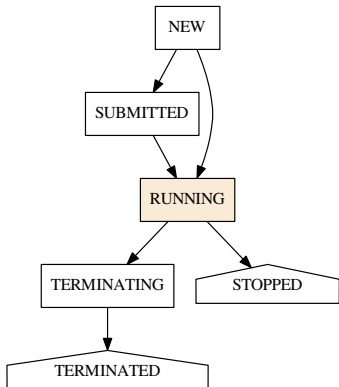
Application lifecycle: state SUBMITTED



SUBMITTED applications have been successfully sent to a computational resource.

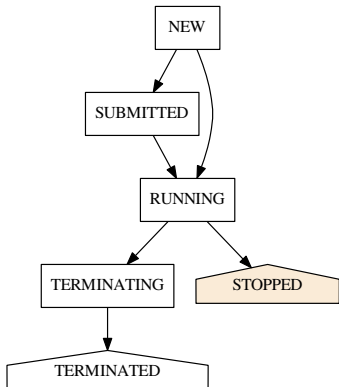
(The transition to *RUNNING* happens automatically, as we do not control the remote execution.)

Application lifecycle: state **RUNNING**



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

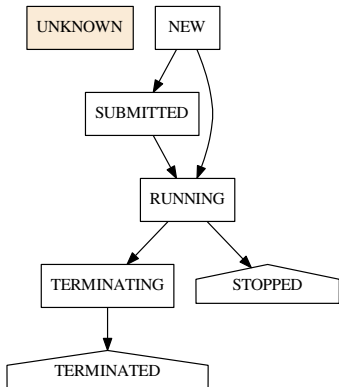
Application lifecycle: state STOPPED



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.

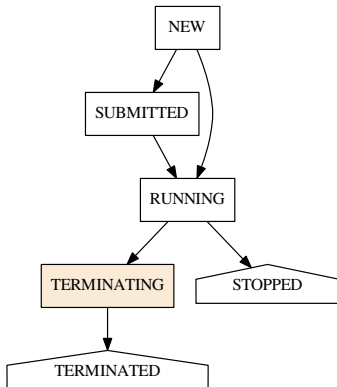
Application lifecycle: state UNKNOWN



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.)

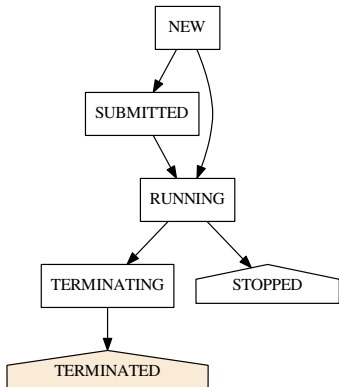
Application lifecycle: state **TERMINATING**



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

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 `stderr` have been captured, the **relative** 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):  
        stderr_file = self.output_dir+"/"+self.stderr  
        stderr_size = os.stat(stderr_file).st_size  
        if stderr_size > 0:  
            gc3libs.log.warn(  
                "Application %s reported errors!", self)
```

Useful in post-processing

These attributes are available in the `terminated()` method:

`self.inputs`

Python dictionary, mapping local (absolute) paths to remote paths (relative to execution directory)

`self.outputs`

Python dictionary, mapping remote paths (relative to execution directory) to *URLs* where they have been copied. In particular, `self.outputs.keys()` is the list of output file names.

`self.output_dir`

Path to the local directory where output files have been downloaded.

Detour: How to ... in Python

... list directory contents

Use the `os.listdir()` function:

```
>>> import os
>>> os.listdir('/etc', 'passwd')
['journalctl', 'wdctl', 'uname', ...]
```

Note that `os.listdir()` returns a list of *relative* file names.

... concatenate a directory and a file name

Use the `os.path.join()` function:

```
>>> import os
>>> os.path.join('/etc', 'passwd')
'/etc/passwd'
>>> os.path.isdir('/non/existent')
False
```

Detour: How to ... in Python

... make a directory

Use the `os.makedirs()` function:

```
>>> import os
>>> os.makedirs('pictures')
```

Note that `os.makedirs()` will raise an error if the directory *already exists*.

... check that a directory exists

Use the `os.path.isdir()` function:

```
>>> import os
>>> os.path.isdir('/tmp')
True
>>> os.path.isdir('/non/existent')
False
```


Exercise 6.A:

In the `colorize.py` script from Exercise 4.A, modify the `ColorizeApp` application to move the output picture file into directory `/home/ubuntu/pictures`. You might need to store the output file name to have it available when the application has terminated running.

(You might want to check out <http://stackoverflow.com/a/8858026/459543> if you're unsure how to move/rename a file with Python.)

Termination status

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 uses some signal codes (not used 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
```

Reference: https://github.com/uzh/gc3pie/blob/master/gc3libs/__init__.py#L1579

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 6.B:

Modify the grayscaling script `ex2c` (or the code it depends upon) so that, when a `GrayscaleApp` task has terminated execution, 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.

Exercise 6.B+: *(Bonus points)* Abstract the verbose terminated method from exercise 6.B into an application class `TermStatusApp`.

Use Python class inheritance to add the `TermStatusApp` functionality into `GrayscaleApp`.

Application-specific configuration

Application classes may be tagged so that parts of the configuration file can be overridden just for them.

Suppose you tag the `GrayscaleApp` class by giving it this name:

```
class GrayscaleApp(Application):  
    application_name = 'grayscale'  
    # [...]
```

then you can provide a specific VM image just for “grayscale” applications:

```
# in the GC3Pie config file:  
[resource/sciencecloud]  
# [...]  
image_id=2b227d15-8f6a-42b0-b744-ed52e5e59f7  
grayscale_image_id=0cca5346-ca12-4cb4-8007-8875c10cce02
```

Other configuration items that can be specialized are:
`instance_type`, `user_data` (cloud), and `prolog_file`,
`epilog_file` (batch-systems).

Detour: How to ... in Python

... take a substring of a string

Use the `text[start:end]` notation:

```
>>> text = 'awesome.m'
>>> text[0:-2]
'awesome'
```

... check for substring inclusion

Use the `in` operator:

```
>>> 'awe' in 'awesome'
True
```

... read whole contents of a file

Use the `read()` on a opened file:

```
>>> data = open('results.csv')
>>> data.read()
'50,50.123,...'
```

Exercise 6.C: (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 -nodesktop -nojvm -r file
```

where *file.m* is the file given to the `MatlabApp()` constructor.

- ▶ captures the standard error output (`stderr`) of the MATLAB script and, if one of the strings “Out of memory.” or “exceeds maximum array size” occurs in it, sets the application `exitcode` to 11.

Verify that it works by running MATLAB script `ra.m` many times over. The script initializes a array of random size: for some values, the size exceeds the amount of available memory.

Global post-processing, I

Further options for customizing a session-based script:

`before_main_loop(self)`

to execute some code *before* the main loop starts.

`after_main_loop(self)`

to execute some code *after* the main loop, i.e., before the script quits. A list of all Application objects is available in the `self.session.tasks.values()` list.

Global post-processing, II

Example: compute statistical distribution of termination statuses:

```
def after_main_loop(self):  
    # check that all tasks are terminated  
    can_postprocess = True  
    for task in self.session.tasks.values():  
        if task.execution.state != 'TERMINATED':  
            can_postprocess = False  
            break  
    if can_postprocess:  
        # do stuff... (see next slide)
```

Global post-processing, III

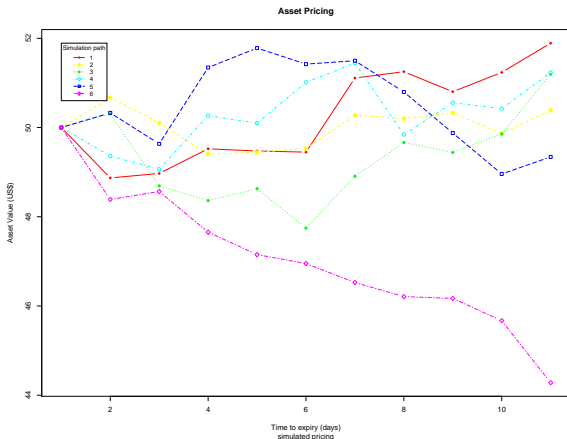
Example: compute statistical distribution of termination statuses (cont'd):

```
def after_main_loop(self):  
    # ... (see prev slide)  
    if can_postprocess:  
        status_counts = defaultdict(int)  
        for app in self.session.tasks.values():  
            termstatus = app.execution.returncode  
            status_counts[termstatus] += 1
```

Variable `self.session.tasks` holds a mapping
JobID \Rightarrow Application; thus
`self.session.tasks.values()` is a list of all the
Application instances returned by `new_tasks`

Detour: asset pricing, I

The script `simAsset.R` simulates asset pricing over a certain amount of time. Different pricing paths are generated using a **1D Brownian motion**, all starting from the same initial price.



Detour: asset pricing, II

You can run the `simAsset.R` script with these positional parameters:

- S_0 stock price today (e.g., 50)
- μ expected return (e.g., 0.04)
- σ volatility (e.g., 0.1)
- δ size of time steps (e.g., 0.273)
- e days to expiry (e.g., 1000)
- N number of simulation paths to generate

For example:

```
$ Rscript simAsset.R 50 0.04 0.1 0.27 10 4
```

Each run of `simAsset.R` produces two output files:

- results.csv** table of generated data: each column is a simulation path, each row is a time step;
- results.pdf** plot of the above.

Detour: How to ... in Python

... compute the sum of a list of numbers

Use the built-in `sum()` function:

```
>>> sum([1, 2, 3])  
6
```

... get the number of items in a list

Use the built-in `len()` function:

```
>>> len(['a', 'b', 'c'])  
3
```

... convert a string to number

Use the built-in `float()` or `int()` functions:

```
>>> int('3')  
3
```

Detour: How to ... in Python

... read a CSV file

Use the `csv` module from the standard library:

```
import csv

path = '/some/file.csv'
data = open(path)
rows = csv.reader(data)
for row in rows:
    # process row
```

... extract fields from CSV rows

When using the `csv` module, rows are just tuples of values. Use `row[i]` to access the i -th field in the row.

Exercise 6.D: *(Difficult)*

Write a `sim_asset.py` program that:

- ▶ takes the same command-line positional arguments as `simAsset.R`, *plus* an additional integer trailing parameter P ;
- ▶ runs `simAsset.R` (in parallel) P times with the given arguments (so, effectively simulates $N \cdot P$ price paths);
- ▶ reads all the generated `results.csv` files, and
- ▶ computes and prints the average value of the asset at the end of the simulated time, across all $N \cdot P$ price paths.

(For easier reading CSV files, you can use the standard `csv` Python module, see: <https://docs.python.org/2/library/csv.html>)