GC3Pie - The Session Based Script

GC3: Grid Computing Competence Center,
University of Zurich

Oct. 1, 2012

Exercise 3.C

Update the exercice 3.B so that it runs 10 copies of the **CpuinfoApplication** and collect statistics about CPU models (how many unique “model name” strings)

Why exercise 3.C failed?

- **localhost** resource does not allow more than two jobs at the same time.

Why exercise 3.C failed?

- **localhost** resource does not allow more than two jobs at the same time.
  - $\Rightarrow$ **Engine**, a more advanced *version* of **Core**, is able to manage a list of jobs and to deal with these situations.

Why exercise 3.C failed?

- **localhost** resource does not allow more than two jobs at the same time.
  - $\Rightarrow$ **Engine**, a more advanced *version* of **Core**, is able to manage a list of jobs and to deal with these situations.
- if the script is killed, all information about the jobs are lost.

Why exercise 3.C failed?

- **localhost** resource does not allow more than two jobs at the same time.

  $\Rightarrow$ **Engine**, a more advanced *version* of **Core**, is able to manage a list of jobs and to deal with these situations.

- if the script is killed, all information about the jobs are lost.

  $\Rightarrow$ a **Session** is a *persistent collection of jobs*. They are saved on the filesystem or a DB.

Why exercise 3.C failed?

- **localhost** resource does not allow more than two jobs at the same time.
  - ⇒ **Engine**, a more advanced *version* of **Core**, is able to manage a list of jobs and to deal with these situations.
- if the script is killed, all information about the jobs are lost.
  - ⇒ a **Session** is a *persistent collection of jobs*. They are saved on the filesystem or a DB.
- some logic is common to any script, including code to *glue* all together and to parse command line options.

Why exercise 3.C failed?

- **localhost** resource does not allow more than two jobs at the same time.
  - ⇒ **Engine**, a more advanced *version* of **Core**, is able to manage a list of jobs and to deal with these situations.
- if the script is killed, all information about the jobs are lost.
  - ⇒ a **Session** is a *persistent collection of jobs*. They are saved on the filesystem or a DB.
- some logic is common to any script, including code to *glue* all together and to parse command line options.
  - ⇒ a **SessionBasedScript** automatically create an **Engine**, group all the jobs into a **Session**, accept some commonly used options and much more.

Creating a SessionBasedScript is easy

Create a file named `demoscript.py`:

```
from gc3libs import Application
from gc3libs.cmdline import SessionBasedScript

class Gdemo(SessionBasedScript):
    """ Gdemo script """
    version='1.0'

    def new_tasks(self, extra):
        return [
            Application(['/bin/hostname'], [], [],
                        stdout='stdout.txt', **extra)
            ]

if __name__ == "__main__":
    from demo import Gdemo
    Gdemo().run()
```

Creating a SessionBasedScript is easy

```
from gc3libs import Application
from gc3libs.cmdline import SessionBasedScript

class Gdemo(SessionBasedScript):
    """ Gdemo script """
    version='1.0'

    def new_tasks(self, extra):
        return [
            Application(['/bin/hostname'], [], [],
                        stdout='stdout.txt', **extra)
            ]

if __name__ == "__main__":
    from demo import Gdemo
    Gdemo().run()
```

Creating a SessionBasedScript is easy

```
from gc3libs import Application
from gc3libs.cmdline import SessionBasedScript

class Gdemo(SessionBasedScript):
    """ Gdemo script """
    version='1.0'

    def new_tasks(self, extra):
        return [
            Application(['/bin/hostname'], [], [],
                        stdout='stdout.txt', **extra)
            ]

if __name__ == "__main__":
    from demo import Gdemo
    Gdemo().run()
```

Creating a SessionBasedScript is easy

```
from gc3libs import Application
from gc3libs.cmdline import SessionBasedScript

class Gdemo(SessionBasedScript):
    """ Gdemo script """
    version='1.0'

    def new_tasks(self, extra):
        return [
            Application(['/bin/hostname'], [], [],
                        stdout='stdout.txt', **extra),
            ]

if __name__ == "__main__":
    from demo import Gdemo
    Gdemo().run()
```

Running the script

```
kenny:~$ python demoscript.py -C 1
[...]
          NEW    0/1      (0.0%)
      RUNNING    0/1      (0.0%)
      STOPPED    0/1      (0.0%)
    SUBMITTED    0/1      (0.0%)
   TERMINATED    1/1    (100.0%)
TERMINATING    0/1      (0.0%)
      UNKNOWN    0/1      (0.0%)
           ok    1/1    (100.0%)
        total    1/1    (100.0%)
```

Running the script

```
kenny:~$ python demoscript.py -C 1
[...]
          NEW    0/1     (0.0%)
      RUNNING    0/1     (0.0%)
      STOPPED    0/1     (0.0%)
    SUBMITTED    0/1     (0.0%)
   TERMINATED    1/1   (100.0%)
TERMINATING    0/1     (0.0%)
      UNKNOWN    0/1     (0.0%)
           ok    1/1   (100.0%)
        total    1/1   (100.0%)
```

In the current directory you will find two directories:

demoscript the directory containing the session data.

Application-N1 the directory containing the output of the application.

The session directory

- It contains internal data used by gc3pie.
- You can specify a different name using the option `-s SESSION_NAME`
- If a session already exists, the script will **not** create new jobs, but instead, will update the status of the jobs in the current session.

The bottom line is...

The session directory

- It contains internal data used by gc3pie.
- You can specify a different name using the option `-s SESSION_NAME`
- If a session already exists, the script will **not** create new jobs, but instead, will update the status of the jobs in the current session.

The bottom line is...

don't touch it!

The output directory

- If you don't specify an output directory for your job, the **SessionBasedScript** class will do it for you.
- If an output directory already exists, this will be *renamed* and never overwritten.
- If you pass the option `-o DIRECTORY` to the script, all the output dirs will be saved inside that directory

In our case, the output directory `Application-N1` will contain a file `stdout.txt` with the output of the application.

The output directory

- If you don't specify an output directory for your job, the **SessionBasedScript** class will do it for you.
- If an output directory already exists, this will be *renamed* and never overwritten.
- If you pass the option `-o DIRECTORY` to the script, all the output dirs will be saved inside that directory

In our case, the output directory `Application-N1` will contain a file `stdout.txt` with the output of the application.

Don't trust me, check yourself

Session Based Script - command line options

–help	show an help message and exits
-C NUM	Keep running, monitoring jobs and possibly submitting new ones or fetching results every NUM seconds. Exit when all jobs are finished.
-o DIR	Output files from all jobs will be collected in the specified DIRECTORY path.
-s PATH	Store the session information in the directory at PATH
-r NAME	Submit jobs to a specific computational resources. NAME is a resource name or comma-separated list of such names.
-J NUM	Set the max NUMber of jobs (default: 50) running at the same time.

Passing requirements to the application

Some options are used to specify some requirements of the applications:

-c NUM Set the number of CPU cores required for each job.

-m GB Set the amount of memory required per execution core

-w DURATION Set the time limit for each job; default is 8 hours.

Passing requirements to the application

Some options are used to specify some requirements of the applications:

-c NUM Set the number of CPU cores required for each job.

-m GB Set the amount of memory required per execution core

-w DURATION Set the time limit for each job; default is 8 hours.

and are automatically passed to the application, if you remember to do it!

```
def new_tasks(self, extra):
    return [
        Application(['/bin/hostname'], [], [],
                    stdout='stdout.txt', **extra),
            ]
```

Exercise 5.A

- Create a **GHelloWorld** application that writes the string `Hello, World!` into a file.
- Create a **GHelloScript** script that runs 20 instances of the **GHelloWorld** application.

How to add command line options

To setup new arguments you must override the `setup_options` method of the script.

```
def setup_options(self):
    self.add_param('-x', '--option', dest='varname',
                   default="Default value",
                   help="Meaningful help string"
                   "which will be printed"
                   "by the --help option")
```

- Supports short and/or long options.
- if `dest='varname'` then the content will be available inside the script as `self.params.varname`

Reference: http://docs.python.org/dev/library/argparse.html#argparse.ArgumentParser.add_argument

Exercise 5.B

Starting from the Exercise 5.A:

- add an option `--string` which accept a string argument, which is the string that will be printed by the application instead of `Hello, World!`
- add an option `--copies` that accept an integer argument (by default, 1), and modify `new_tasks` so that it will run `copies` number of the **GHelloWorld** application.

SessionBasedScript - short recap

What the father class will do for you:

- It reads and parses the GC3Pie configuration file.
- It creates an **Engine** class.
- It creates a **Session** to persist jobs.
- It parses commonly used command line arguments.
- It submit jobs, check their status, fetch their output when they are finished.
- It automatically sets the following parameters:
  - `output_dir`
  - `requested_cores`
  - `requested_memory`
  - `requested_walltime`
  - `jobname`

SessionBasedScript - customization

To customize the script you have to modify:

`setup_options(self)` to add command line options

`new_tasks(self, extra)` method to return a list of **Application**-like instances. Here, you can access command line options via `self.params.option_name`

`before_main_loop(self)` to execute some code *before* the submission of the jobs.

`after_main_loop(self)` to execute some code *after* the main loop. A list of all Application objects is available in the `self.session.tasks.values()` list.

Exercise 5.C

Create a script which will run a variable number of copies of the **CpuinfoApplication** of the `cpuinfo.py` script and will print the results at the end. Remember to:

- in `setup_options` add a command line option `--copies`.
- in `new_tasks` read the `self.params.copies` attribute to know how many applications to run.
- in `after_main_loop` check if all the application are done, and eventually print the results.