Introduction on GC3Pie

GC3: Grid Computing Competence Center,
University of Zurich

Oct. 1, 2012

What is GC3Pie and why do we need something like that ?

high-throughput (HTC) use cases I

Run application on a range of different inputs. Each input is a different file (or a set of files).

Then collect output files and post-process them, e.g., gather some statistics.

Typically implemented by a set of `sh` / `perl` scripts to drive execution on a local cluster.

high-throughput (HTC) use cases II

Need to chain together different execution steps (workflow)

Execution flow is not uniform (e.g. do not have to repeat the same action over every input)

Execution flow is determined at runtime (dynamic dependency)

Potential issues

1. **Portability:** Cannot run on a different cluster without rewriting all the scripts.
2. **Code reuse:** Scripts are often very tied to a certain purpose, so they are difficult to reuse.
3. **Heavy maintenance:** the more a script does its job well, the more you'll find yourself adding `generic` features and maintaining requests from other users.

Recurring patterns for an HTC driver script

1. **Access** to computational resources
2. **Supervise** execution of collection of jobs
3. **Handling** of error conditions individually
4. **Post-process** and store results

What is GC3Pie ?

GC3Pie is a Python toolkit:

it provides the building blocks to write Python scripts to run large computational campaigns and

to combine several tasks into a dynamic workflow.

What is GC3Pie ?

GC3Pie consists of three main components:

GC3Libs:

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

GC3Apps:

A collection of driver scripts to run large job campaigns.

GC3Utils:

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

An example: ggamess

```
import gc3libs
from gc3libs.application.gamess
     import GamessApplication
from gc3libs.cmdline
     import SessionBasedScript

class  GGamessScript(SessionBasedScript):
  def __init__(self):
    SessionBasedScript.__init__(
       self,
       application = GamessApplication,
       input_filename_pattern = '*.inp'
    )

if __name__ == '__main__':
  GGamessScript().run()
```

GC3Pie for developers

Programming model based on customization of base classes through inheritance (Template method pattern)

Different level of interfaces depending on the control required

SessionBasedScript is the highest level of abstraction

How is GC3Pie different? (I)

GC3Pie runs specific **applications**, not generic jobs.

That is, GC3Pie exposes `Application` classes whose programming interface is adapted to the specific task/computation a scientific application performs.

You can add your own application by specializing the generic `Application` class.

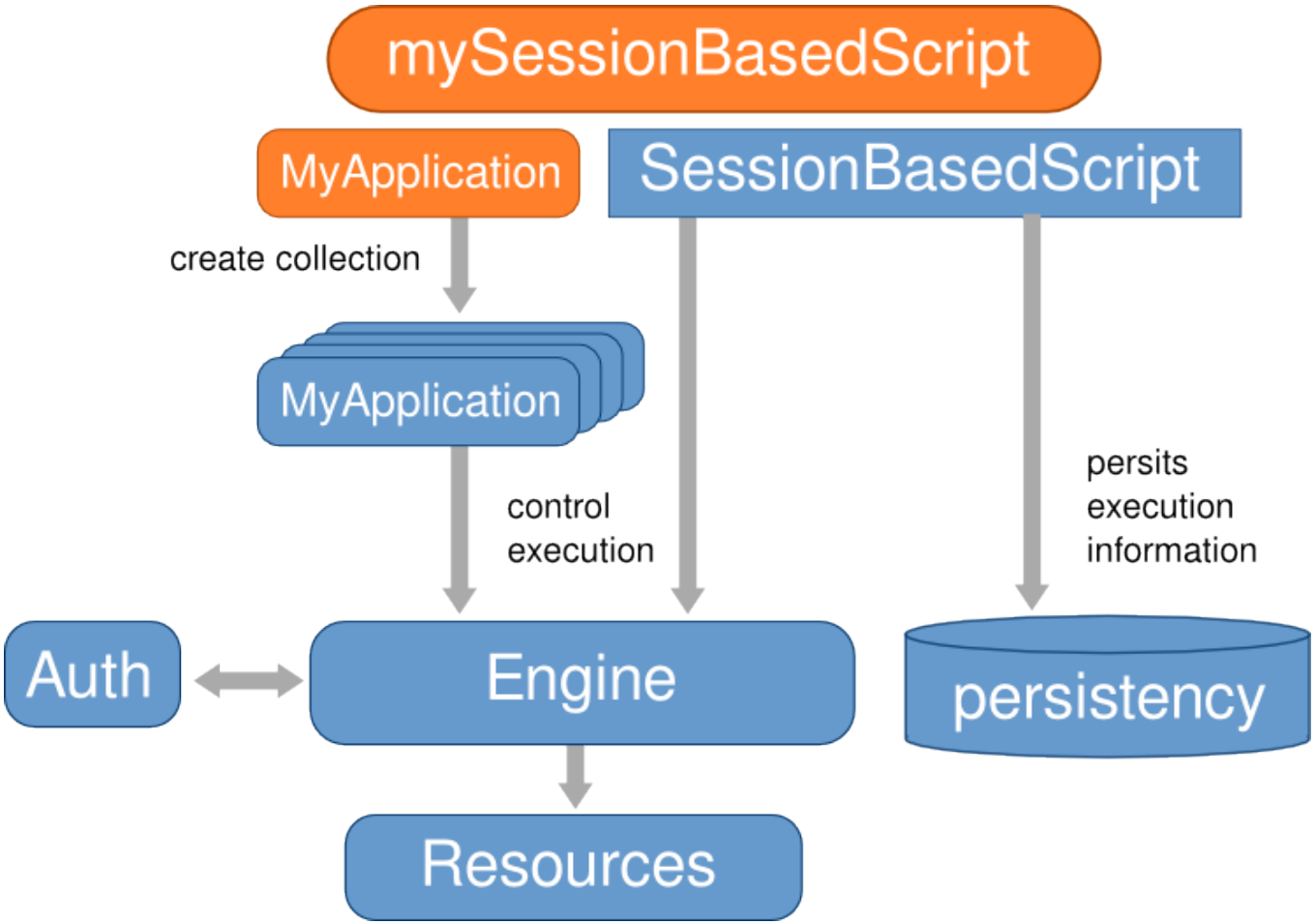
How is GC3Pie different? (II)

GC3Pie can run applications in parallel, or sequentially, or any combination of the two, and do arbitrary processing of data in the middle.

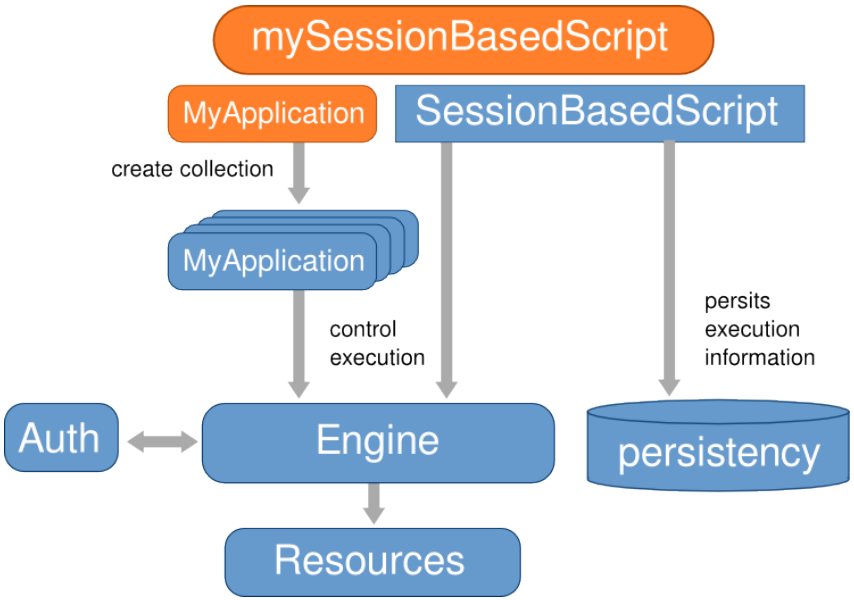
Think of workflows, except you can write them in the Python programming language.

Which means, you can create them dynamically at runtime, adapting the schema to your problem.

GC3Pie Execution model



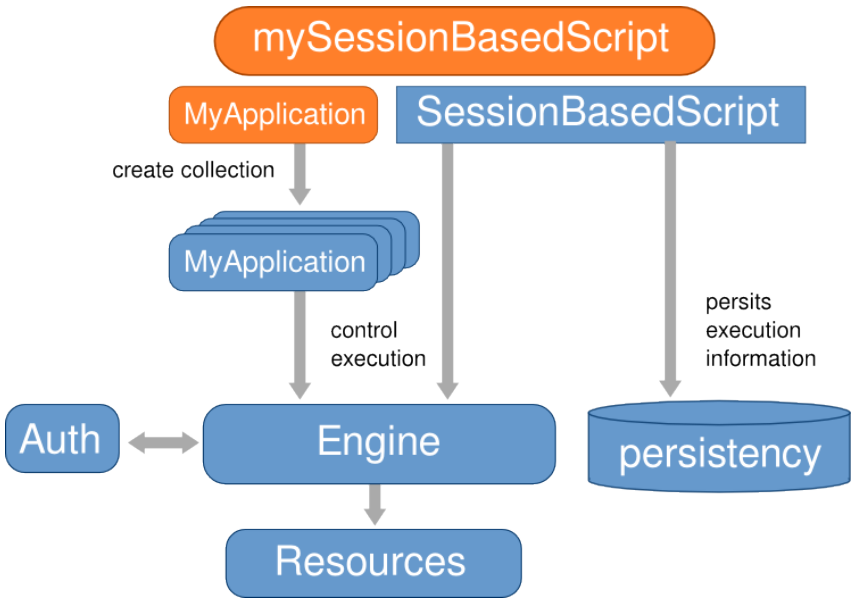
GC3Pie Execution model



An application is a subclass of the `gc3libs.Application` class.

Applications can be grouped in collections

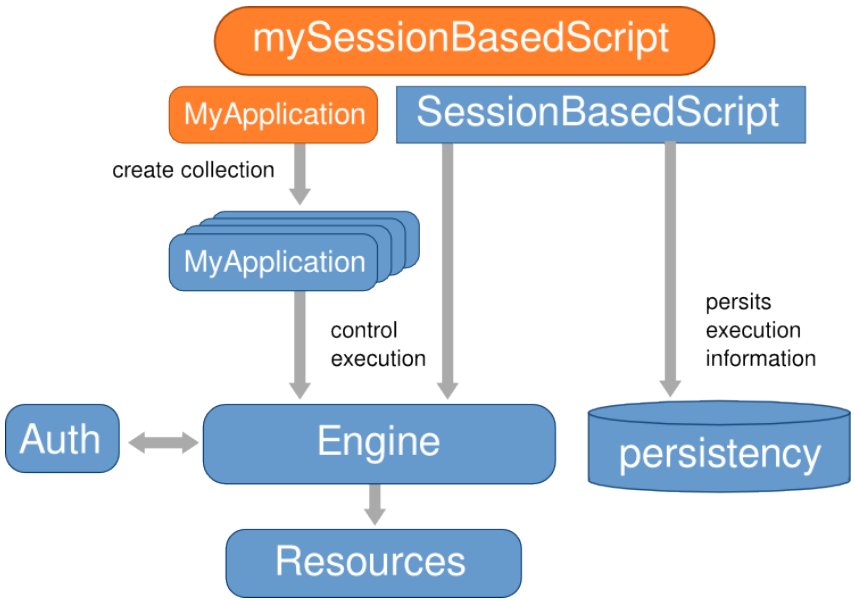
GC3Pie Execution model



Execution of collections is delegated to an **`Engine`**.
(two modes supported: synchronous and asynchronous)

Execution Engine handles the access to computational resources (also verifying the proper authentication mechanism)

GC3Pie Execution model



A convenient class SessionBasedScript contains already most of the control logic for instructing the execution engine

`SessionBasedScript` takes also care of persisting execution information

A simple high-throughput script structure...

1. Initialize computational resource (e.g., authentication step)
2. Prepare files for submission
3. Submit jobs
4. Monitor job status (loop)
5. React on failures (e.g. resubmit)
6. Retrieve results
7. Postprocess and display

A high-throughput script with GC3Pie, revisited

1. *Create a gc3libs.core.Core instance*
2. *Create a gc3libs.persistence.FilesystemStore instance*
3. *Create a gc3libs.core.Engine instance*
4. *Load saved jobs into it*
5. Create *new* instance(s) of the application class
6. *Let engine manage jobs until all are done*
7. ~~Retrieve results~~ (the `Engine` does it)
8. Postprocess and display

Steps 1-4, and 6-7 are automatically done by the `SessionBasedScript` class.

Job dependency management

An `Engine` manages all jobs concurrently. What if there are inter-application dependencies?

GC3Pie provides Task composition support (workflow), created programmatically from Python code.

Which means, no graphical editor. But also means you can create workflows on-the-fly as your computation proceeds.