pythia-core Documentation

Release 1.0

Sébastien Combéfis

Jun 10, 2019

User's Documentation

Pythia is a framework deployed as an online platform whose goal is to teach programming and algorithm design. The platform executes the code in a safe environment and its main advantage is to provide intelligent feedback to its users to suppor their learning. More details about the whole project can be found on the official website of Pythia.

Pythia-core is the backbone of the Pythia framework. It manages a pool of UML virtual machines and is in charge of the safe execution of low-level jobs. Pythia-core is written in Go and can be easily distributed on several machines or in the cloud.

CHAPTER 1

Quick install

Since the pythia-core framework uses UML-based virtual machines, it can only be run on Linux.

Start by installing required dependencies:

- Make (4.0 or later)
- Go (1.2.1 or later)
- SquashFS tools (`squashfs-tools`)
- Embedded GNU C Library (`libc6-dev-i386`)

Then, clone the Git repository, and launch the installation:

```
> git clone --recursive https://github.com/pythia-project/pythia-core.git
> cd pythia-core
> make
```

Once successfully installed, you can try to execute a simple task:

```
> cd out
> touch input.txt
> ./pythia execute -input="input.txt" -task="tasks/hello-world.task"
```

and you will see, among others, `Hello world!` printed in your terminal.

CHAPTER 2

Contents

This documentation is split into two parts: the first one is targeter to users and the second one is for developers. In any case, we recommend you to first read the user's documentation to understand how to use and test the framework.

2.1 Presentation

The pythia-core framework makes it possible to execute code in a safe environment. It can be used to grade codes written by non-trusted people, such as students in a learning environment, for example. The framework can execute jobs provided with optional inputs and that will produce outputs upon completion.

2.1.1 Queue and pools

The pythia-core framework is built on two mains components, namely one queue and several pools. As shown on Figure 1, the *queue* is the entry point of the framework which receives the jobs to be executed from the outside. It then dispatches the jobs to pools (of execution) and waits for the result to send it back to the job's submitter. A *pool* launches a new virtual machine for each job it is asked to execute, so that to execute it in a safe and controlled environment.

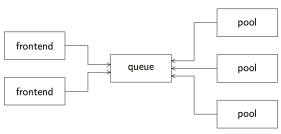

Fig. 1: Figure 1: The pythia-core framework is composed of two main components, namely a queue and several pools.

The queue is the main component and is waiting for incoming connections. It means that the other components, that is, the pools and the frontends, have to first connect to the queue before being able to offer their services.

2.1.2 Environment and task filesystems

Jobs that are executed by the virtual machines launched by the pools. They are composed of several elements as shown on Figure 2. The *environment* filesystem contains all the files that are necessary to execute the job (compilers, interpretors, configuration files, etc.) The *task* filesystem contains all the files that are specific to the job. In particular, it contains a *control* file which is launched at the job startup.

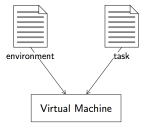

Fig. 2: Figure 2: A job is composed of an environment and a task files.

Several predefined environments are available in the environments repository on the GitHub page of the Pythia project organisation.

2.1.3 Task description

In addition to the environment and task filesystems, the virtual machine can be configured with a set of constraints. The configuration is stored in a *.task* file containing a JSON object. The following table lists the parameters that can be configured:

Key	Subkey	Description
`environment`		Environment to use to execute the job
`taskfs`		Path to the task filesystem
`limits`	`time`	Maximum execution time allowed (seconds)
	`memory`	Maximum amount on main memory (Mo)
	`disk`	Size of the disk memory (percentage)
	`output`	Maximum size of the output (number of characters)

2.1.4 Hello World example

Let us examine a simple example of a job whose execution simply returns *Hello World!*. This example can be found in the `tasks` directory of the pythia-core repository on GitHub. The first thing to do is to define the task filesystem. The first file, namely `hello.sh`, is just a shell script that prints `Hello World!` on the standard output.

```
#!/bin/sh
echo "Hello world!"
```

The second file, namely `control`, must contain the sequence of executables to launch. It is the only file that is mandatory in a task filesystem. The one of this example just calls the `hello.sh` script. Note that the task filesystem is mounted in the `/task` directory inside the virtual machine.

```
/task/hello.sh
```

Finally, constraints related to the execution environment of the job are specified in the `hello-world.task` file. The job uses the `busybox` environment (which provides several stripped-down Unix tools including `sh`) and the task filesystem is contained in the `hello-world.sfs` file. The execution time is limited to 60 seconds, the main memory to 32 Mo, the disk size to 50% and the length of the output to 1024 characters.

```
{
  "environment": "busybox",
  "taskfs": "hello-world.sfs",
  "limits": {
    "time": 60,
    "memory": 32,
    "disk": 50,
    "output": 1024
  }
}
```

2.2 Task execution

Now that you have a global idea of how the pythia-core framework is architectured, let us examine how to create a new task and then submit a job to execute it with the framework.

2.2.1 Task filesystem

As previously mentioned, a task is just a set of files in a given directory. This latter is then *compressed* into an `.sfs` file using the SquashFS filesystem. Two elements must be kept in mind when creating a task filesystem:

- It must contain a `control` file with the sequence of executables to launch.
- The task filesystem will be mounted in the `/task` directory inside the virtual machine.

The *Hello World* example presented in the *presentation section* is composed of two files is the `hello-world` directory:

```
hello-world/
   control
   hello.sh
```

To build a *SquashFS filesystem* for this directory, you just have to use `mksquashfs`. The following command will create a `hello-world.sfs` file with the content of the `hello-world` directory:

```
> mksquashfs hello-world hello-world.sfs -all-root -comp lzo -noappend
```

Note that you can extract the files contained in a SquashFS file with the `unsquashfs` command:

```
> unsquashfs -d hello-world hello-world.sfs
```

2.2.2 Task execution

As previously mentioned, a task can be easily executed by the pythia-core framework with the `execute` subcommand. For that to be possible, an additional `.task` file containing the configuration of the task must be created. This file refers in particular to the `.sfs` file with the SquashFS filesystem of the task. The input to provide for the execution of the task is stored in a text file, which can be empty.

The task can be executed as a job in the pythia-core framework with the following command, assuming that the `pythia` executable is in your `PATH` and that you are in the directory containing the three files `hello-world.sfs`, `hello-world.task` and `input.txt`:

```
> pythia execute -input="input.txt" -task="hello-world.task"
Warning: unable to read configuration file: open config.json: no such file or␣
↪directory
Status: success
Output: Hello world!
```

As you can observe, the `execute` subcommand produces two pieces of information: the status of the execution and the output produced by the task. We can also notice a warning about a missing configuration file, described hereafter.

Configuration

The options of the `pythia` executables can be specified directly through the command line or thanks to a `config.json` file. This file is not mandatory but can be useful when *setting up the pythia-core framework*. The configuration is a JSON object composed of a global section and one specific section for each available subcommand. To know what are the possible configuration options, please refer to the *usage page* of this documentation.

Hereafter is an example of a `config.json` file:

```
{
  "global": {
    "queue": "127.0.0.1:9000"
  },
  "components": [
    {
      "component": "queue",
      "capacity": "500"
    },
    {
      "component": "pool"
    }
  ]
}
```

Execution status

There are *seven different status* for the execution of a task, summarised in the table hereafter. Depending on the status, the output takes different values. The standard output (`stdout`) referred to in the *output* column of the table corresponds to the one generated by the execution of the job.

Status	Description	Output
`success`	Finished	stdout
`timeout`	Timed out	stdout so far
`overflow`	stdout too big	capped stdout
`abort`	Aborted by abort message	–
`crash`	Sandbox crashed	stdout
`error`	Error (maybe temporary)	error message
`fatal`	Unrecoverable error (e.g. misformatted task)	error message

Standard input

The only way external information can be provided to a task is through its standard input. When executing a task with the `execute` subcommand, you can specify the path to a text file whose content will be poured in the standard input

of the task when executed.

To better understand, let us look at another example that can be found in the `tasks` directory of the pythia-core repository on GitHub. The *Hello input* example reads all the lines of text from its standard input and says hello to them. The `hello.sh` script launched by the `control` file reads the standard input and echoes the hello greetings on the standard output:

```
#!/bin/sh
while read input; do
    echo "Hello ${input}!"
done
```

Let us execute the task with the following `input.txt` file:

```
Sébastien
Virginie
```

The execution results in the printing of two lines of text in the standard output, greeting Sébastien and Virginie:

```
> pythia execute -input="input.txt" -task="hello-input.task"
Status: success
Output: Hello Sébastien!
Hello Virginie!
```

2.3 Framework setup

We now know how to create a new task and to execute it directly with the pythia-core framework. In this section we examine how to setup the pythia-core framework, that is, starting the queue and the pools and submitting tasks to the queue for execution.

2.3.1 Starting the queue

The main component of the pythia-core framework is the *queue*. Once started, the other components have to connect to the queue to register themselves and start using the services provided by the queue. The queue is characterised by two parameters:

- the address and port number it is listening to (127.0.0.1:9000 by default);
- the maximum number of tasks waiting for execution (capacity of 500 by default).

Starting the queue with default parameters is as easy as:

```
> pythia queue
Listening to 127.0.0.1:9000
```

2.3.2 Connecting pools

Once the queue is started, you have to start *execution pools* and connect them to the queue, so that this latter can use them to execute tasks. A pool is characterised by five parameters:

- the address and port of the queue to connect to (127.0.0.1:9000 by default);
- the maximum number of parallel running sandboxes (capacity of 1 by default);
- the directory where to find enviroments (`vm` by default)

- the directory where to find tasks (`tasks` by default)
- the path to the UML executable (`vm/uml` by default)

Starting a new pool and connecting it to the queue is as easy as:

```
> pythia pool
Connected to queue 127.0.0.1:9000
```

In the console of the queue, you can also notice that a new pool has been connected to the queue:

```
> pythia queue
Listening to 127.0.0.1:9000
Client 0: connected.
Client 0: pool capacity 1
```

You can start as many pools as you want, as far as your machine is powerful enough to withstand the load. The queue will automatically balance the tasks as equally as possible between all the pools that are connected to it.

2.3.3 Submitting a task with the server

Once the queue and pools are launched and correctly setup, it is possible to submit a task to the pythia-core framework by connecting as a frontend to the queue and sending the execution request. It is a different way to proceed compared to the direct execution with the `execute` subcommand where information about the task to execute are provided through a `.task` file.

To make it easier to connect to the queue and send it request, a `server` submodule is available. This submodule launches an HTTP web server that communicates directly with the queue, reading information about the task from the `.task` file. You can also specify the text to pass to the standard input of the task with a parameter. A server is characterised by one parameter:

- the port number it is listening to (80 by default)

Launching a new frontend server is as easy as:

```
> pythia server
Server listening on 8080
```

Once the server is started, you can simply launch the execution of a task with the cURL program, for example:

```
> curl --data '{"tid": "hello-input", "response": "Sébastien\nVirginie\n"}' http://
↪localhost:8080/execute
Hello Sébastien!
Hello Virginie!
```

2.4 Usage

The pythia-core framework is contained in a single executable file simply named `pythia`. The different components of the framework can be launched with subcommands. There are currently four available components in the pythia-core framework. Here is a summary about how to use the main executable:

```
Usage: ./pythia [global options]
       ./pythia [global options] component [options]

Launches the pythia platform with the components specified in the configuration
```

(continues on next page)

(continued from previous page)

```
file (first form) or a specific pythia component (second form).

Available components:
  server       Front-end component allowing execution of pythia tasks
  execute      Execute a single job (for debugging purposes)
  pool         Back-end component managing a pool of sandboxes
  queue        Central queue back-end component

Global options:
  -conf string
     configuration file (default "config.json")
  -queue string
     queue address (default "127.0.0.1:9000")
```

2.4.1 Execute

The `execute` subcommand launches a new job to execute a task:

```
Usage: ./pythia [global options] execute [options]

Execute a single job (for debugging purposes)

Options:
  -envdir string
     environments directory (default "vm")
  -input string
     path to the input file (mandatory)
  -task string
     path to the task description (mandatory)
  -tasksdir string
     tasks directory (default "tasks")
  -uml string
     path to the UML executable (default "vm/uml")
```

2.4.2 Queue

The `queue` subcommand launches a queue that receives demands to execute tasks:

```
Usage: ./pythia [global options] queue [options]

Central queue back-end component

Options:
  -capacity int
     queue capacity (default 500)
```

2.4.3 Pool

The `pool` subcommand launches an execution pool to run jobs in UML virtual machines:

```
Usage: ./pythia [global options] pool [options]

Back-end component managing a pool of sandboxes

Options:
  -capacity int
     max parallel sandboxes (default 1)
  -envdir string
     environments directory (default "vm")
  -tasksdir string
     tasks directory (default "tasks")
  -uml string
     path to the UML executable (default "vm/uml")
```

2.4.4 Server

The `server` subcommand launches a frontend server:

```
Usage: ./pythia [global options] server [options]

Front-end component allowing execution of pythia tasks

Options:
  -port int
     server port (default 8080)
```

2.5 Architecture

2.6 Communication messages