oct Documentation

Release 0.3.7

karec

April 01, 2016

Contents

Informations

OCT is still in alpha version and is not yet suitable for production, but we working hard on it for make it to the beta ! Follow us for more informations about the next releases
twitter @oct_py

CHAPTER 1

Introduction

1.1 What is OCT

OCT is an agnostic load testing tool. What do we mean by agnostic ? It's simple : OCT provides only the needed tools to distribute and run your tests and compile the results. But the tools and programming languages for writing the tests themselves are up to you.

At this stage of development we only offer a python turret, but if you want to create your own turret implementation in any language, please do it ! We're really open to any suggestions and help.

1.2 Terminology

- HQ for Headquarters: it's the "server" component of OCT and it's tasked with sending start signal, stop signal, collecting results and create reports
- Turret: a turret is the "client" component of OCT. It can be writen in any language and it communicates with the HQ using a zeromq PUSH socket. Each turret owns one or many canons.
- Canons: represent the virtual users, wich means that each canon of the turret will run a test in parallel

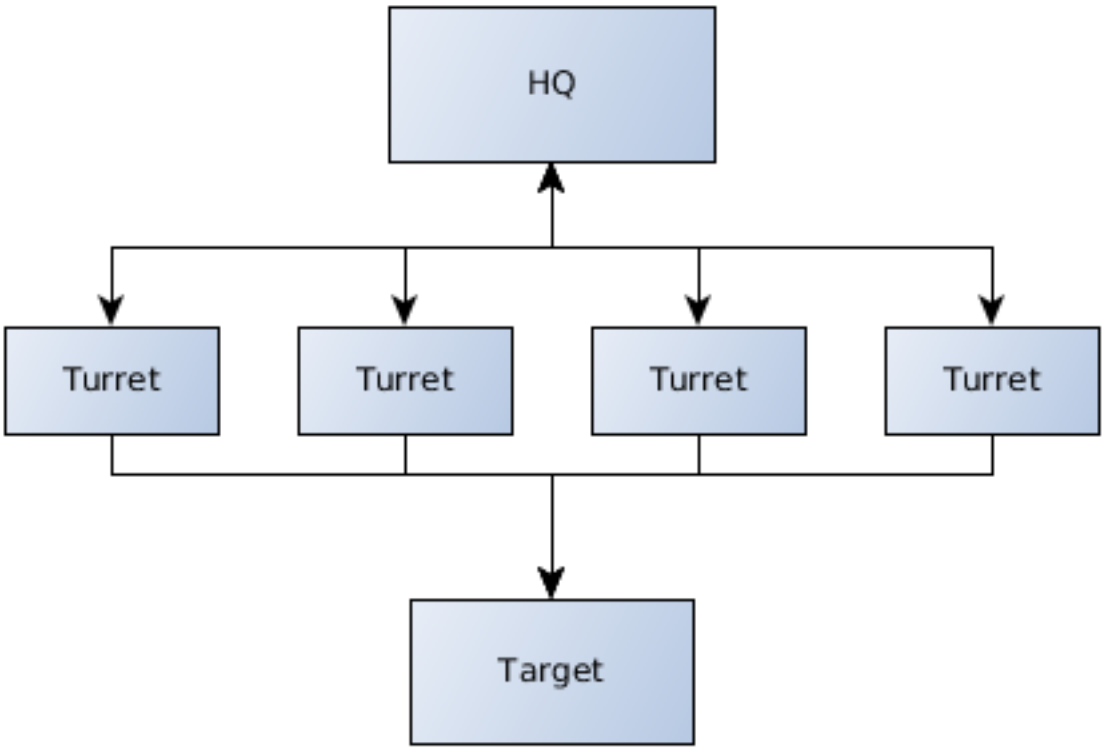
Note: Why do we use this terminology ? It gives a good idea of what's happening when you use OCT. Think about it like that : each turret owns X canons that shoot at the target and report on the result to the HQ

1.3 How it works ?

OCT uses the power of `zeromq` to distribute test on any number of physical machines you need. When running a test the process is very simple :

- OCT starts the HQ. It sends a start message to the turrets and will later collect their results
- The turrets receive the message, start the tests and send results to the HQ
- When the test ends, the HQ sends a stop message to the turrets and process the remaining messages in queue
- OCT will then compile the results and create a html report of them

Want a graph ? Here you go :



So this is it, a bunch of turrets shooting at the target and sending information to the HQ.

CHAPTER 2

Installation

2.1 OCT-Core

OCT is avaible on pypi so you can install it with pip :

```
pip install oct
```

Or directly from the source :

```
python setup.py install
```

You will also need the python headers for installing some of the dependencies like numpy, and `build-essential` and `python-dev` to compile them

On a debian based system you can install them using apt for example :

```
apt-get install python-dev build-essential
```

Note: The OCT core part have been developed and tested on linux based system only, at this point of the developement process we cannot guarantee you that the oct-core can be installed on a Windows system

2.2 OCT-Turrets

You can actually choose any turret that you need, in any langage. But the `oct` package require the python turrets by default and the "oct-turrets" pypi package will be automaticaly installed with the main `oct` package.

CHAPTER 3

Your first project

OCT exposes several command-line tools to use it, write your tests or run your project.

First we're going to use the `oct-newproject` command for creating our first project.

```
oct-newproject my_project_name
```

This command creates a folder named `my_project_name` containing all the required files to start an OCT project.

Let's take a look at the content of this new folder :

```
|-- config.json
-- templates
|   -- css
|   |   -- style.css
|   -- img
|   -- report.html
|   -- scripts
-- test_scripts
    -- v_user.py
```

Those files are the minimum required by an OCT project. We will analyze them in details in this documentation but let's take it file by file.

3.1 Configuration

The main and more important file here is the *config.json* file, let's take a look at his default content :

```
{
    "run_time": 30,
    "results_ts_interval": 10,
    "progress_bar": true,
    "console_logging": false,
    "testing": false,
    "publish_port": 5000,
    "rc_port": 5001,
    "min_turrets": 1,
    "turrets": [
        {"name": "navigation", "canons": 2, "rampup": 0, "script": "test_scripts/v_user.py"},
        {"name": "random", "canons": 2, "rampup": 0, "script": "test_scripts/v_user.py"}
    ],
    "turrets_requirements": []
}
```

Every key here is useful, but some keys are not required to run a test. Let's take a look at the main ones :

- `run_time`: This key simply sets the time of a test in seconds. So in this case the test will run for 30 seconds.
- `results_ts_interval`: Time interval between two sets of results in seconds. For exemple if we have a run time of 30 seconds and an interval of 10, we will have 3 results sets in the final report
- `testing`: If this key is set to True, the *results.sqlite* file will be created in */tmp* folder
- `publish_port`: The port for the publishing socket of the HQ
- `rc_port`: The port for the result collector (PULL socket) of the HQ
- `min_turrets`: The minimum number of turrets that must be deployed before the HQ sends the start message
- `turrets`: a list of turrets, this key will be use to package turrets with the *oct-pack-turrets* command
- `turrets_requirements`: A list of string containing the required packages for turrets (only for python turrets at this time)

This configuration is simple but will allow you to run simple tests in a local environment.

Now let's take a look at the per-turret configuration :

Each turret can be configured independently, and you can setup different options for each one :

- `name`: the string representation for the turret
- `canons`: The number of canons for this turret (aka virtual users)
- `rampup`: Turrets can spawn their cannon progressively, not each at the same time. Rampup gives a "step" for cannon initialization. The number of cannon spawned by second is equal to the total number of cannons of the turret by rampup - e.g., if you have 30 cannons and a rampup of 15 seconds, it will spawn 2 cannons by seconds. If you do not want to increase the number of cannons in time but start the tests with all cannons ready to fire, leave rampup at 0, as in the exemple.
- `script`: The relative path to the associated test script

3.2 Writing tests

By default, the `oct-new-project` command will create an exemple test script under `test_scripts/v_user.py`, let's take a look at it :

```
from oct_turrets.base import BaseTransaction
import random
import time


class Transaction(BaseTransaction):
    def __init__(self):
        pass

    def run(self):
        r = random.uniform(1, 2)
        time.sleep(r)
        self.custom_timers['Example_Timer'] = r
```

Note: As you can see the default test is writen in python, but each turret can have its own implementation and its own way to write tests. Refer to turrets documentation for more explanations on how to write tests with the selected turret.

So this file represent a basic test that will simply wait between 1 or 2 seconds. Not really useful but it give you an exemple on how to write tests and we will keep this example when running our tests in the local setup. For advanced explanations on how to write tests, please see Writing tests

3.3 That's all you need

And that's all you need ! Some configuration and basics tests and that's it.

Of course this will not be enough to test your infrastructure or website, but at this point you should better undersand how OCT work and what you need to run your tests ! In the next part we will talk about writing more complexe tests.

CHAPTER 4

Writing tests

Warning: This section will explain how to write tests, but only based on the **python** turret. But many turrets will have similar implementation

4.1 Basic example

Let's take the default `v_user.py` file :

```
from oct_turrets.base import BaseTransaction
import random
import time


class Transaction(BaseTransaction):
    def __init__(self):
        pass

    def run(self):
        r = random.uniform(1, 2)
        time.sleep(r)
        self.custom_timers['Example_Timer'] = r
```

This raw script will test nothing as it is, so let's work on this simple use case:

We need to test a basic API over the Internet and we want to use the `requests` python library.

So first let's adapt the script to our needs:

```
import time
import requests
from oct_turrets.base import BaseTransaction


class Transaction(BaseTransaction):
    def __init__(self):
        # each canon will only instanciate Transaction once, so each property
        # in the Transaction __init__ method will be set only once so take care if you need to update
        self.url = "http://my-api/1.0/"

    def run(self):
        # For more detailed results we will setup several custom timers
```

```
        start = time.time()
        requests.get(self.url + "echo")
        self.custom_timers['Echo service'] = time.time() - start

        start = time.time()
        requests.get(self.url + "other-service")
        self.custom_timers['other-service'] = time.time() - start
```

So what are we doing here ? We've just imported requests and used it in our script. For each service we've defined a custom timer to see how much time each one will take to answer.

But how to install the dependencies needed by the turrets ? You can simply update your configuration with something like that :

```
{
    "run_time": 30,
    "results_ts_interval": 10,
    "progress_bar": true,
    "console_logging": false,
    "testing": false,
    "publish_port": 5000,
    "rc_port": 5001,
    "min_turrets": 1,
    "turrets": [
        {"name": "navigation", "canons": 2, "rampup": 0, "script": "test_scripts/v_user.py"},
        {"name": "random", "canons": 2, "rampup": 0, "script": "test_scripts/v_user.py"}
    ],
    "turrets_requirements": [
        "requests"
    ]
}
```

If you specify the dependecies in the "turrets_requirements" you will be able to install them for each turret by simply runing :

```
pip install my_turret_package.tar
```

4.2 Setup and Tear down

The previous example is still pretty simple, but you might need things like sessions or cookies. How to manage it knowing that the transaction class will instantiate only once ?

Pretty simple too: we give you two methods in the `BaseTransaction` class to help you : `setup` and `tear_down`

How does it works ? Take a look a this example:

```
import time
import requests
from oct_turrets.base import BaseTransaction


class Transaction(BaseTransaction):
    def __init__(self):
        # each canon will only instanciate Transaction once, so each property
        # in the Transaction __init__ method will be set only once so take care if you need to update
        self.url = "http://my-api/1.0/"
        self.session = None
```

```
    def setup(self):
        self.session = requests.Session()

    def run(self):
        # For more detailed results we will setup several custom timers
        start = time.time()
        self.session.get(self.url + "echo")
        self.custom_timers['Echo service'] = time.time() - start

        start = time.time()
        self.session.get(self.url + "other-service")
        self.custom_timers['other-service'] = time.time() - start

    def tear_down(self):
        self.session.close()
```

And that's it ! Before each `run` iteration, the `setup` method is called, and you've guessed it, `tear_down` is called after the iteration.

Note: The setup and the tear_down method are not included in the stats sent to the HQ, so the actions will not be included in the `scriptrun_time` statistic

---

CHAPTER 5

Packaging your turrets

Warning: This section will explain how to package turrets, but only based on the **python** turret. But most turrets will have similar implementation

So that's it ? You've written all your tests and you're ready to start to fire at your target ? Well let's prepare your turrets for deployement !

5.1 Auto packaging

Warning: This example only works for python based turrets. Please refer to your turret documentation if you use anything else

OCT provides a simple way to package your turrets and set them ready to fire, the `oct-pack-turrets` command. It generates tar files based on your configuration file. Those tar files are the turrets, ready to fire at your command.

You can use it like this :

```
oct-pack-turrets /path/to/oct/project
```

A sucessful packing should return the following output :

```
Added config.json
Added setup.py
Added test_scripts/v_user.py
Archive ./navigation.tar created
========================================
Added config.json
Added setup.py
Added test_scripts/v_user.py
Archive ./random.tar created
========================================
```

In addition if some optionnal keys of the configuration are not set, you could see something like that :

```
WARNING: hq_address configuration key not present, the value will be set to default value
```

You will see a WARNING line per missing key. Also if a required key is not set the command will throw an exception like that :

```
oct.core.exceptions.OctConfigurationError: Error: the required configuration key <key> is not define
```

Where <key> is the missing key

5.2 Installing and starting the turrets

Now that your turrets are packaged, you can install them using pip for example :

```
pip install navigation.tar
```

This command will install all required packages listed under the turrets_requirements configuration key, plus the oct-turrets package itself.

Once the installation is finished you can start your turret using the oct-turrets-start like that :

```
oct-turrets-start --tar navigation.tar
```

And if everything is fine you should see this message :

```
[2015-12-21 18:02:09,295: INFO | oct_turrets.turret] starting turret
```

You are now ready to fire at the target !

CHAPTER 6

Running Tests

So that's it ? Your turrets are running and ready to fire at the target ? Si let's do it ! Leeeeeerooooy....

6.1 Configuration

Before running the tests, don't forget to update your configuration if your turrets are running on a different IP address from the master.

6.2 Starting the test

Just type:

```
oct-run /path/to/oct/project
```

And that's it, your test will start and your turrets will now fire at the target. If everything is going ok you should see an output like:

```
Warmup
waiting for 1 turrets to connect
waiting for 0 turrets to connect
turrets: 1, elapsed: 20.0   transactions: 4906  timers: 4906  errors: 0
```

So... That's it ? And yes that's it ! You've successfuly run your first OCT test !

Once the test have ended you should see the following output :

```
Processing all remaining messages...


analyzing results...

transactions: 4906
errors: 0

test start: 2015-12-21 18:23:06
test finish: 2015-12-21 18:23:26

Report generated in 2.75798082352 seconds
created: ././results/results_2015.12.21_18.23.05_162132/results.html

done.
```

CHAPTER 7

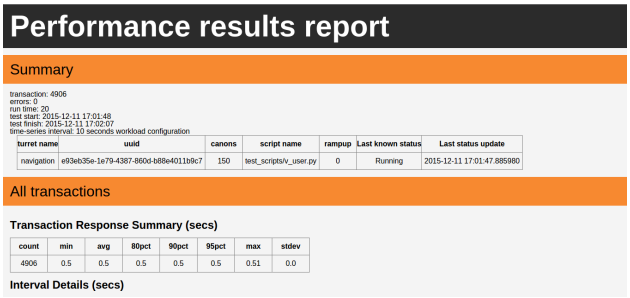
Collecting the results

The tests have ended and the report has been created ? Let's take a look at it !

7.1 Results.html file

This is the main and the more explicit part of the results, it will give you all major information about the tests, plus some graphs to help you read the results.

A default result page looks like this :



For each custom timer, a section will be created (like the "All transactions" section) and the associated graphs will be created.

The graphs are currently in SVG format and use javascript to make reading and interpretating the result easier.

7.2 Regenerate results

Sometimes you may need to regenerate the html report with all graphs from an sqlite file. OCT got a tool that allows you to do this.

You can simply use the `oct-rebuild-results` like this for example:

```
oct-rebuild-results . results.sqlite config.json
```

Note: The oct-rebuild-results command will only work on an already created results folder that contains only the sqlite results and optionnaly the configuration.

CHAPTER 8

Writing your own turret

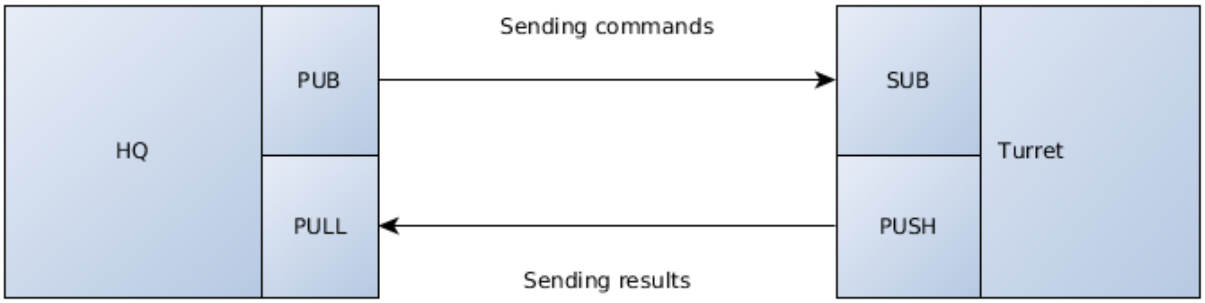
You use OCT but the python turret doesn't fit your needs ? Or you need a library avaible only in one language ? Or maybe you just want to create a turret with your favorite programing language ? No problem, this guide is here for you !

Note: You can base your turret on the python turret, source code avaible here

8.1 Global workflow

OCT uses zeromq for communication between the HQ and the turrets. That means that you can write a turret in any language with a zeromq binding (zeromq bindings)

But before writing your code, you need to understand how turrets must communicate with the HQ. Here is a schema to explain it :



So as you can see this is pretty simple, the HQ send orders to the turrets using a PUB/SUB pattern, and the turrets will send results to the master using a PUSH/PULL pattern.

Note: The python turret also uses a push/pull pattern to enable communication between cannons and the turret itself. All cannons have an inproc socket connected to the turret process

8.2 Requirements

Before going any further, you need to know what a turret must be able to do :

- Reading a turret configuration file (see below)
- Spawning N number of canons (set in the configuration file)
- Managing rampup
- Importing test files and run it
- Sending well formated json messages to the HQ

8.3 The turret configuration

As you can see in the python-turret example (in the GitHub repository), a turret must be able to read and understand this type of configuration file :

```
{
    "name": "navigation",
    "canons": 50,
    "rampup": 10,
    "script": "v_user.py",
    "hq_address": "127.0.0.1",
    "hq_publisher": 5000,
    "hq_rc": 5001
}
```

The configuration is pretty simple - and yes this is the full configuration needed for a turret to run.

Let's explain all keys :

- `name`: the display name of the turret for the report
- `canons`: the number of canons to spawn (remember, canons == virtual users)
- `rampup`: the rampup value in seconds
- `script`: the path/name of the test script to load
- `hq_address`: the IP address of the HQ
- `hq_publisher`: the port of the PUB socket of the HQ
- `hq_rc`: the port of the PULL socket of the HQ

All keys are required.

8.4 Sockets configuration

To communicate with the headquarters, you will need only two zmq sockets :

- A sub socket listening on the general, empty string '' channel and on '<turret_uniq_id>' channel (for direct orders)
- A push socket to send results to the master

For example, in the python turret the sockets are created this way :

```
self.context = zmq.Context()

self.master_publisher = self.context.socket(zmq.SUB)
self.master_publisher.connect("tcp://{}:{}".format(self.config['hq_address'], self.config['hq_publish
```

```
self.master_publisher.setsockopt_string(zmq.SUBSCRIBE, '')
self.master_publisher.setsockopt_string(zmq.SUBSCRIBE, self.uuid)

self.result_collector = self.context.socket(zmq.PUSH)
self.result_collector.connect("tcp://{}:{}".format(self.config['hq_address'], self.config['hq_rc']))
```

You need to listen to the `master_publisher` socket to retrieve commands from the master. These commands can be :

- `start`: tells the turret to start the tests
- `status_request`: headquarters ask for the status of the turret (RUNNING, WAITING, etc.)
- `kill`: tells the turret to shutdown
- `stop`: tells the turret to stop tests and clean everything to be in ready status again

8.5 HQ commands format

The HQ will send commands in JSON format. All command messages will contain 2 keys : `command` and `msg`.

For example :

```
{
    "command": "stop",
    "msg": "premature stop"
}
```

8.6 Tell the HQ that your turret is ready to fire

The master need to know if your turret is ready or not. Why ? Because the HQ can be set up to wait for `n` number of turrets before starting the tests.

But don't worry, it's pretty simple to tell the master that your turret is ready, you only need to send a json message with the `PUSH` socket of your turret.

The status message SHOULD contain all of the following fields:

- `turret`: the name of the turret (eg: navigation, connection, etc.)
- `status`: the current status of the turret (ready, waiting, running, etc.)
- `uuid`: the unique identifier of the turret
- `rampup`: the rampup setting of the turret
- `script`: the test script associated with the turret
- `canons`: the number of canons on the turret

A complete json status message will look like this:

```
{
    "turret": "navigation",
    "status": "READY",
    "uuid": "d7b8a1cc-639a-405c-9b16-62ce5cd66f36",
    'rampup': "30",
    'script': "tests/navigation.py",
```

```
    'canons': "250"
}
```

Note: The status messages are not fixed, since it will only be used in the final html report for displaying the latest known status of each turret. But it's important to update it, since a crashing turret will obviously impact final results

8.7 Results messages format

All results messages that will be sent to the HQ should have the same pattern. Note that if the HQ receive a badly formatted message, it will fail silently and you will lose those data.

But don't worry, once again the pattern of the message is pretty simple :

```
{
    "turret_name": "my_turret"
    "elapsed": 12.48, // total elapsed time in seconds
    "epoch": 1453731738 // timestamp
    "scriptrun_time": 1.2, // the time it took to execute the current transaction (aka the response t
    "error": "My custom error", // the error string. Empty if there are no errors
    "custom_timers": {
        "Example_timer": 0.6, // An example custom timer
        "Other timer": 0.8
    }
}
```

See ? Pretty simple, isn't it ?

This message will be sent throught the `push` socket of the turret and will be received by the `pull` socket of the master.

Warning: The master use the `recv_json()` method to retreive messages comming from the turret, so take care to sent message using the appropriate `send_json()` method

8.8 Error management

The way turrets must manage errors is pretty simple :

- If the error is inside the test scripts, the turret should keep running
- If the error happens at the turret level, the turret should send a notification to the master before dying

So, what happens when an error is thrown inside the test script ? Simple, your turret should log it and send it to the master in the `error` key of the reponse message. This way, the user could be informed if something went wrong, but the test will continue to run.

And now, if the error appears at the turret level, how to tell the HQ that your turret is dead ? Pretty simple again, a simple status message with the new status of your turret :

```
{
    "turret": "navigation",
    "status": "Aborted",
    "uuid": "d7b8a1cc-639a-405c-9b16-62ce5cd66f36",
    'rampup': "30",
```

```
    'script': "tests/navigation.py",
    'canons': "250"
}
```

If you sent this message, in the final html report the user will be able to see that one turret is dead and at what moment the turret as stopped

8.9 Writing your own packaging system

For this you're pretty free to implement it the way you want / need it. But don't forget that the goal of the packaging system is to provide simple way to distribute turret in one command line.

Don't forget to document the way your user can packages their turrets and how they can run it !

Plus, the packaging avaible in the core of OCT will be rewritten to be more generic as soon as possible.

8.10 Document your turret

Simply put: please, document your turret !

We expect to create a list to reference all available turrets, and if your turrets doesn't have a documentation, we will refuse to list it.

But keep in mind that for many case, a simple README is enough. At the very least, tell your users how to install and start your turret.

CHAPTER 9

API Reference

9.1 oct.core package

9.1.1 Submodules

9.1.2 oct.core.exceptions module

exception oct.core.exceptions.**FormNotFoundException**
Bases: exceptions.Exception

Raised in case of FormNotFound with browser

exception oct.core.exceptions.**LinkNotFound**
Bases: exceptions.Exception

Raised in case of link not found in current html document

exception oct.core.exceptions.**NoFormWaiting**
Bases: exceptions.Exception

Raised in case of action required form if no form selected

exception oct.core.exceptions.**NoUrlOpen**
Bases: exceptions.Exception

Raised in case of no url open but requested inside browser class

exception oct.core.exceptions.**OctConfigurationError**
Bases: exceptions.Exception

Provide an oct configuration error

exception oct.core.exceptions.**OctGenericException**
Bases: exceptions.Exception

Provide generic exception for reports

9.1.3 oct.core.hq module

class oct.core.hq.**HightQuarter**(*publish_port*, *rc_port*, *stats_handler*, *config*)
Bases: object

The main hight quarter that will receive informations from the turrets and send the start message

Parameters

- **publish_port** (*int*) – the port for publishing information to turrets
- **rc_port** (*int*) – the result collector port for collecting results from the turrets
- **stats_handler** (StatsHandler) – the stats handler writer
- **config** (*dict*) – the configuration of the test

run()
Run the hight quarter, lunch the turrets and wait for results

wait_turrets(*wait_for*)
Wait until wait_for turrets are connected and ready

9.1.4 oct.core.main module

9.1.5 oct.core.test_loader module

9.1.6 Module contents

9.2 oct.results package

9.2.1 Submodules

9.2.2 oct.results.models module

class oct.results.models.**Result**(**args*, ***kwargs*)
Bases: peewee.Model

Define a result model

DoesNotExist
alias of ResultDoesNotExist

custom_timers = <peewee.TextField object>

elapsed = <peewee.FloatField object>

epoch = <peewee.FloatField object>

error = <peewee.TextField object>

id = <peewee.PrimaryKeyField object>

scriptrun_time = <peewee.FloatField object>

to_dict()

turret_name = <peewee.CharField object>

class oct.results.models.**Turret**(**args*, ***kwargs*)
Bases: peewee.Model

Define a turret model

DoesNotExist
alias of TurretDoesNotExist

canons = <peewee.IntegerField object>

id = <peewee.PrimaryKeyField object>

name = <peewee.TextField object>

rampup = <peewee.IntegerField object>

save(**args*, ***kwargs*)

script = <peewee.TextField object>

status = <peewee.TextField object>

to_dict()

updated_at = <peewee.DateTimeField object>

uuid = <peewee.TextField object>

oct.results.models.**set_database**(*db_path*, *proxy*, *config*)
Initialize the peewee database with the given configuration

Parameters

- **db_path** (*str*) – the path of the sqlite database
- **proxy** (*peewee.Proxy*) – the peewee proxy to initialise
- **config** (*dict*) – the configuration dictionnary

9.2.3 oct.results.report module

class oct.results.report.**ReportResults**(*run_time*, *interval*)
Bases: object

Represent a report containing all tests results

Parameters

- **run_time** (*int*) – the run_time of the script
- **interval** (*int*) – the time interval between each group of results

compile_results()
Compile all results for the current test

9.2.4 oct.results.writer module

class oct.results.writer.**ReportWriter**(*results_dir*, *parent*)
Bases: object

A class representing a report, used to output the result

Parameters

- **results_dir** (*str*) – the output directory for the report
- **parent** (*str*) – the parent directory

set_statics()
Create statics directory and copy files in it

write_report(*template*)
Write the compiled jinja template to the results file

Parameters **str** (*template*) – the compiled jinja template

9.2.5 oct.results.ouput module

9.2.6 oct.results.stats_handler module

class oct.results.stats_handler.**StatsHandler**(*output_dir*, *config*)
Bases: object

This class will handle results and stats comming from the turrets

Parameters **output_dir** (*str*) – the output directory for the results

write_remaining()
Write the remaning stack content

write_result(*data*)
Write the results received to the database

Parameters **data** (*dict*) – the data to save in database

Returns None

write_turret(*data*)
Write the turret information in database

Parameters **data** (*dict*) – the data of the turret to save

Returns The turret object after save

9.2.7 oct.results.graphs module

oct.results.graphs.**get_local_time**(*index*)
Localize datetime for better output in graphs

Parameters **index** (*pandas.DateTimeIndex*) – pandas datetime index

Returns aware time objet

Return type datetime.time

oct.results.graphs.**resp_graph**(*dataframe*, *image_name*, *dir='./'*)
Response time graph for bucketed data

Parameters

- **dataframe** (*pandas.DataFrame*) – dataframe containing all data
- **image_name** (*str*) – the output file name
- **dir** (*str*) – the output directory

Returns None

oct.results.graphs.**resp_graph_raw**(*dataframe*, *image_name*, *dir='./'*)
Response time graph for raw data

Parameters

- **dataframe** (*pandas.DataFrame*) – the raw results dataframe
- **image_name** (*str*) – the output file name
- **dir** (*str*) – the output directory

Returns None

oct.results.graphs.**tp_graph**(*dataframe*, *image_name*, *dir='./'*)
Throughput graph

Parameters

- **dataframe** (*pandas.DataFrame*) – dataframe containing all data
- **dir** (*str*) – the output directory

Returns None

9.2.8 Module contents

9.3 oct.tools package

9.3.1 Submodules

9.3.2 oct.tools.rebuild_results module

oct.tools.rebuild_results.**main**()

9.3.3 oct.tools.results_to_csv module

oct.tools.results_to_csv.**main**()

oct.tools.results_to_csv.**results_to_csv**(*result_file*, *output_file*, *delimiter=';'*)
Take a sqlite filled database of results and return a csv file

Parameters

- **result_file** (*str*) – the path of the sqlite database
- **output_file** (*str*) – the path of the csv output file
- **delimiter** (*str*) – the desired delimiter for the output csv file

9.3.4 Module contents

9.4 oct.utilities package

9.4.1 Submodules

9.4.2 oct.utilities.configuration module

oct.utilities.configuration.**configure**(*project_name*, *cmd_opts*, *config_file=None*)
Get the configuration of the test and return it as a config object

Returns the configured config object

Return type Object

oct.utilities.configuration.**configure_for_turret**(*project_name*, *config_file*)
Load the configuration file in python dict and check for keys that will be set to default value if not present

Parameters

- **project_name** (*str*) – the name of the project
- **config_file** (*str*) – the path of the configuration file

Returns the loaded configuration

Return type dict

9.4.3 oct.utilities.newproject module

oct.utilities.newproject.**create_project**(*project_name*)
Create a new oct project

Parameters **project_name** (*str*) – the name of the project

oct.utilities.newproject.**main**()

9.4.4 oct.utilities.pack module

oct.utilities.pack.**main**()

oct.utilities.pack.**pack_turret**(*turret_config*, *tmp_config_file*, *tmp_setup*, *base_config_path*, *path=None*)
pack a turret into a tar file based on the turret configuration

Parameters

- **turret_config** (*dict*) – the turret configuration to pack
- **tmp_config_file** (*str*) – the path of the temp config file
- **base_config_path** (*str*) – the base directory of the main configuration file

9.4.5 oct.utilities.run module

oct.utilities.run.**main**()
Main function to run oct tests.

oct.utilities.run.**run**(*cmd_args*)
Start an oct project

Parameters **cmd_args** (*Namespace*) – the commande-line arguments

9.4.6 Module contents

O

S

T

U

W