Loads Documentation

Release 0.1

Mozilla Services

September 27, 2013

CONTENTS

Loads is a tool to load test your HTTP services.

With **Loads**, your load tests are classical Python unit tests which are calling the service(s) you want to send load to.

It also comes with a command line to run the actual load.

Loads tries its best to avoid reinventing the wheel, so we offer integration with 3 existing libraries: **Requests**, **WebTest** and **ws4py**.

Here's a really simple test example:

```
from loads.case import TestCase

class TestWebSite(TestCase):

    def test_es(self):
        self.session.get('http://localhost:9200')
```

If you don't want to write your load tests in python, or if you want to use any other library to describe the testing, **Loads** allows you to use your own formalism. see :doc:zmq-api.

With such a test, running loads simply consists of doing:

```
$ bin/loads-runner example.TestWebSite.test_es
[=============================================================================]  100%

Hits: 1
Started: 2013-06-14 12:15:42.860586
Duration: 0.03 seconds
Approximate Average RPS: 39
Average request time: 0.01s
Opened web sockets: 0
Bytes received via web sockets : 0

Success: 1
Errors: 0
Failures: 0
```

See *User Guide* for more options and information.

CHAPTER

ONE

MORE DOCUMENTATION

1.1 Installation

1.1.1 Prerequisites

Loads is developed and tested with Python 2.7.

Loads uses ZeroMQ and Gevent, so you need to have libzmq and libev on your system. You also need the Python headers.

Under Debuntu:

```
$ sudo apt-get install libev-dev libzmq-dev python-dev
```

And under Mac OS X, using Brew:

```
$ brew install libev
$ brew install zeromq
$ brew install python
```

Make sure you have a C compiler, and then pip:

```
$ curl -O https://raw.github.com/pypa/pip/master/contrib/get-pip.py
$ sudo python get-pip.py
```

This will install pip globally on your system.

The next step is to install Virtualenv:

```
$ sudo pip install virtualenv
```

This will also install it globally on your system.

1.1.2 Basic installation

Now we can build **Loads** locally:

```
$ make build
```

This will compile Gevent 1.0rc2 using Cython, and all the dependencies required by Loads - into a local virtualenv.

That's it. You should then find **load-runner** in your bin directory.

1.1.3 Distributed

To install what's required to run distributed tests, you need to run:

```
$ make build_extras
```

1.2 User Guide

1.2.1 Using Loads with Requests

Let's say you want to load test the Elastic Search root page on your system, just to be sure.

Write a unittest like this one and save it in an **example.py** file:

```
from loads.case import TestCase

class TestWebSite(TestCase):

    def test_es(self):
        self.session.get('http://localhost:9200')
```

The *TestCase* class provided by **Load** has a *session* attribute you can use to interact with an HTTP server. It's a **Session** instance from Requests.

Now run **loads-runner** against it:

```
$ bin/loads-runner example.TestWebSite.test_es
[=============================================================================]  100%

Hits: 1
Started: 2013-06-14 12:15:42.860586
Duration: 0.03 seconds
Approximate Average RPS: 39
Average request time: 0.01s
Opened web sockets: 0
Bytes received via web sockets : 0

Success: 1
Errors: 0
Failures: 0
```

This will execute your test just once - so you can control it works well.

Now, try to run it using 100 virtual users (-u), each of them running the test 10 times (-c):

```
$ bin/loads-runner example.TestWebSite.test_es -u 100 -c 10
[=============================================================================]  100%
Hits: 1000
Started: 2013-06-14 12:15:06.375365
Duration: 2.02 seconds
Approximate Average RPS: 496
Average request time: 0.04s
Opened web sockets: 0
Bytes received via web sockets : 0

Success: 1000
Errors: 0
Failures: 0
```

Congrats, you've just sent a load of 1000 hits, using 100 concurrent threads.

Now let's run a series of 10, 20 then 30 users, each one running 20 hits:

```
$ bin/loads-runner loads.examples.test_blog.TestWebSite.test_something --hits 20 -u 10:20:30
```

That's 1200 hits total.

1.2.2 Using Loads with ws4py

Loads provides web sockets API through the **ws4py** library. You can initialize a new socket connection using the **create_ws** method.

Run the echo_server.py file located in the examples directory, then write a test that uses a web socket against it:

```
from loads.case import TestCase

class TestWebSite(TestCase):

    def test_something(self):
        def callback(m):
            results.append(m.data)

        ws = self.create_ws('ws://localhost:9000/ws',
                            callback=callback)
        ws.send('something')
        ws.receive()
        ws.send('happened')
        ws.receive()

        while len(results) < 2:
            time.sleep(.1)

        self.assertEqual(results, ['something', 'happened'])
```

1.2.3 Using Loads with WebTest

If you are a **WebTest** fan, you can use it instead of Requests. If you don't know what webtest is, you should have a look at it ;).

You just need to use **app** instead of **session** in the test class, that's a *webtest.TestApp* object, providing all the APIs to interact with a web application:

```
from loads.case import TestCase

class TestWebSite(TestCase):

    def test_something(self):
        self.assertTrue('Search' in self.app.get('/'))
```

Of course, because the server root URL will change during the tests, you can define it outside the tests, on the command line, with **–server-url** when you run your load test:

```
$ bin/loads-runner example.TestWebSite.test_something --server_url http://localhost:9200
```

Changing the server URL

It may happen that you need to change the server url when you're running the tests. To do so, change the *server_url* attribute of the app object:

```
self.app.server_url = 'http://new-server'
```

1.2.4 Distributed test

If you want to send a lot of load, you need to run a distributed test. A distributed test uses multiple agents to do the requests. The agents can be on the same machine, or on a different physical hardware.

The **Loads** command line is able to interact with several **agents** through a **broker**.

To run a broker and some agents, let's use Circus.

Install Circus:

```
$ bin/pip install circus
```

And run it against the provided **loads.ini** configuration file that's located in the Loads source repository in **conf**:

```
$ bin/circusd --daemon conf/loads.ini
```

Here is the content of the *loads.ini* file:

```
[circus]
check_delay = 5
httpd = 0
statsd = 1
debug = 0

[watcher:broker]
cmd = bin/loads-broker
warmup_delay = 0
numprocesses = 1

[watcher:agents]
cmd = bin/loads-agent
warmup_delay = 0
numprocesses = 5
copy_env = 1
```

What happened? You have just started a Loads broker with 5 agents.

Let's use them now, with the **agents** option:

```
$ bin/load-runner example.TestWebSite.test_something -u 10:20:30 -c 20 --agents 5
[=================================================================]  100%
```

Congrats, you have just sent 6000 hits from 5 different agents. Easy, no?

Detach mode

When you are running a long test in distributed mode, you might want to detach the console and come back later to check the status of the load test.

To do this, you can simply hit Ctrl+C. **Loads** will ask you if you want to detach the console and continue the test, or simply stop it:

```
$ bin/load-runner example.TestWebSite.test_something -u 10:20:30 -c 20 --agents 5
^C
...
Duration: 2.04 seconds
Hits: 964
Started: 2013-07-22 07:12:30.139814
Approximate Average RPS: 473
Average request time: 0.00s
Opened web sockets: 0
Bytes received via web sockets : 0

Success: 964
Errors: 0
Failures: 0

Do you want to (s)top the test or (d)etach ? d
```

Then you can use **–attach** to reattach the console:

```
$ bin/loads-runner --attach
[                                        ]   4%
Duration: 43.68 seconds
Hits: 19233
Started: 2013-07-22 07:12:30.144859
Approximate Average RPS: 0
Average request time: 0.00s
Opened web sockets: 0
Bytes received via web sockets : 0

Success: 0
Errors: 0
Failures: 0

Do you want to (s)top the test or (d)etach ? s
```

1.2.5 Outputs

By default, loads reports the status of the load in real time onthe standard output of the client machine. Depending what you are trying to achieve, that may or may not be what you want.

Loads comes with a pluggable "output" mechanism: it's possible to define your own output format if you need so.

You can change this behaviour with the –output option of the *loads-runner* command line.

1.3 Under the hood — How loads is designed

Hopefully, it's not really complicated to dig into the code and have a good overview of how *loads* is designed, but sometimes a good document explaining how things are done is a good starting point, so let's try!

You can run loads either in *distributed mode* or in *non-distributed* mode. The vast majority of the time, you want to spawn a number of agents and let them hammer the site you want to test. That's what we call the distributed mode. Alternatively, you may want to run things in a single process, for instance while writing your functional tests, that's the *non-distributed* mode.

1.3.1 What happens during a non-distributed run

1. You invoke the *loads.runner.Runner* class.
2. A *loads.case.TestResult* object is created. This object is a data collector, it is passed to the test suite (*TestCase*), the loads *Session* object and the websocket manager. Its very purpose is to collect the data from these sources. You can read more in the section named *TestResult* below.
3. We create any number of outputs (standard output, html output, etc.) in the runner and register them to the test_result object.
4. The *loads.case.TestCase* derivated-class is built and we pass it the test_result object.
5. A number of threads / gevent greenlets are spawned and the tests are run one or multiple times.
6. During the tests, both the requests' *Session*, the test case itself and the websocket objects report their progress in real time to test_result. When there is a need to disambiguate the calls, a loads_status object is passed along.

   It contains data about the hits, the total number of users, the current user and the current hit.
7. Each time a call is made to the test_result object to add data, it notifies its list of observers to be sure they are up to date. This is helpful to create reports in real time, as we get data, and to provide a stream of info to the end users.

1.3.2 What happens during a distributed run

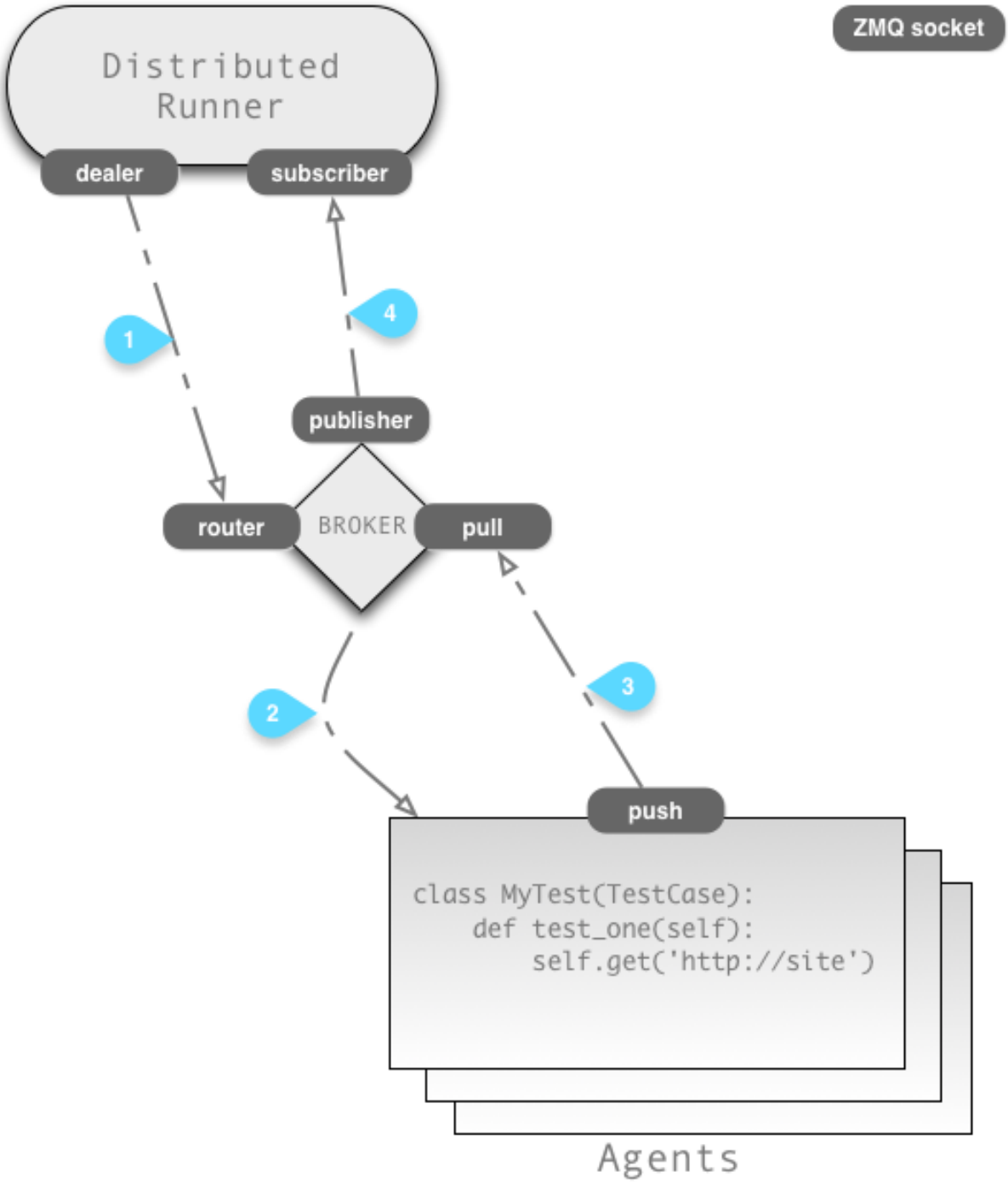
When you run in distributed mode, you have a distributed runner (master) which, rather than running the tests locally, asks an *Agent* to run them. It is possible to run a number of agents at the same time.

These agents are just simple runners, but instead of reporting everything locally, using a *TestResult* object, they relay all the data to the master instance using a 0MQ stream.

It means that the code in *loads/relay.py* is a drop-in replacement for a TestResult object.

Once the results are back to the master, it populates its local *test_runner*, which will in turn call the outputs to generate the reports.

A schema might help you to get things right:



All the inter-process communications (IPC) are handled by ZeroMQ, as you can see on the schema. Here is the caption:

1. The distributed loads runner (**the master**) sends a message to the broker, asking it to run the tests on N agents.
2. The broker selects the spare agents and send them the job. The agents start a loads-runner instance in slave mode (**the slave**), proxying all the calls to the *test_result* objects to the zmq push socket.
3. The **master** receives the calls and pass them to its local *test_results* instance.

1.3.3 The TestResult object

The TestResult object follows the APIs of unittest. That's why you can see methods such as *addSuccess*, *addFailure*, etc.

It is done this way so that you actually can just replace the normal unittest object by the one coming from loads, and gather data this way.

If you have a look at what you can find in *loads/case.py*, you will find that we create a *TestResultProxy* object. This is done so that the test_result object we pass to the TestCase have the exact same APIs than the one in unittest (it used to contain extra arguments).

1.4 Using loads in a different language

Loads is built in a way that makes it possible to have runners written in different languages. It's perfectly possible to have a runner in javascript or ruby, sending data to loads.

This is made possible by the use of zeromq to send inter-process (or even inter-machines!) messages.

This means you can write your load tests with whatever language you want, as long as the test-runner sends back its results in the zmq pipeline, respecting the format described in this document.

1.4.1 Implementations in other languages

- Integration of loads with javascript, named loads.js

1.4.2 Loads messaging format

The messages sent to **loads** always contain a *data_type* key, which describes the type of that that's being sent.

The messages respect the following rules:

- All the data is JSON encoded.
- Dates are expressed in ISO 8601 format, (YYYY-MM-DDTHH:MM:SS)
- You should send along the worker id with every message. Each worker id should be different from each other.

A message generally looks like this:

```
{
    data_type: 'something',
    worker_id: '1',
    other_key_1: 'foo'
    other_key_2: 'bar'
}
```

loads_status

Some messages take a *loads_status* argument. *loads_status* is a list of values concerning the current status of the load. It contains, in this order:

- cycles: the number of cycles that will be running in total
- user: the number of users per cycle)

- current_cycle: the cycle we are currently in
- current_user: the current user that's doing the requests

errors / exceptions

When errors / exceptions are caught, they are serialised and sent trough the wire, as well. When you see an *exc **, it is a list containing this:

- A string representation of the exception
- A string representation of the exception class
- The traceback / Stack trace.

1.4.3 Data types

Before and after you run the tests, you need to tell that you're doing so:

- startTestRun()
- stopTestRun()

Tests

When using loads, you usually run a test suite. Tests start, stop, succeed and fail. Here are the APIs you can use:

- addFailure(test_name, *exc **, loads_status)
- addError(test_name, *exc **, loads_status)
- addSuccess(test_name, loads_status)
- startTest(test_name, loads_status)
- stopTest(test_name, loads_status)

Requests

To track requests, you only have one method, named "add_hit" with the following parameters:

- *url*, the URL of the request, for instance http://notmyidea.org
- *method*, the HTTP method (GET, POST, PUT, etc.)
- *status*, the response of the call (200)
- *started*, the time when it started
- *elapsed*, the number of seconds (decimal) the request took to run
- loads_status, as already described

Sockets

If you're also able to track what's going on with the socket connections, then you can use the following messages:

- socket_open()
- socket_close()
- socket_message(size) # the size, in bytes, that were transmitted via the websocket.