Loads Documentation

Release 0.2

Mozilla Services

September 27, 2013

CONTENTS

Figure 1: by Juan Pablo Bravo

Loads is a tool to load test your HTTP services, including web sockets. With **Loads**, your load tests are classical Python functional tests which are calling the service(s) you want to exercise.

Loads is not asking you to use an ad-hoc API. The tool offers an integration with 3 existing libraries: Requests, WebTest and ws4py. You just need to write your tests using them, and **Loads** will do the rest.

Loads can run tests from a single box or distributed across many nodes, from the same command line tool. All tests results are coming back to you in real time while the load is progressing.

Since you are using Python to build your tests, you can write very complex scenarii, and use **Loads** options to run them using as many concurrent users as your hardware (or cloud service) allows you.

Here's a really simple example where we check that a local Elastic Search server is answering to HTTP calls:

```
from loads.case import TestCase

class TestWebSite(TestCase):

    def test_es(self):
        res = self.session.get('http://localhost:9200')
        self.assertEqual(res.status_code, 200)
```

The test is also checking that the page is sending back a 200. In case it's not behaving properly, **Loads** will let you know.

Note: If you don't want to write your load tests using Python, or if you want to use any other library to write tests, **Loads** can be extended. See *Plugin-in external runners*.

With such a test, running **Loads** is done by pointing the *test_es* method:

```
$ 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 *Writing load tests* for a complete walkthrough. *Loads commands* provides a detailed documentation on all the options you can use.

If want to contribute to the project see *Contributing*.

CHAPTER

ONE

MORE DOCUMENTATION

1.1 Installation

1.1.1 Prerequisites

Loads is developed and tested with Python 2.7.x and Python 2.6.x. We encourage you to use the latest 2.7 version.

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

You can install **Loads** through Pip:

```
$ pip install loads
```

Or build **Loads** from the Git repo:

```
$ git clone https://github.com/mozilla-services/loads
$ cd loads
$ 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 the **loads-runner** command in your bin directory.

1.1.3 Distributed

To install what's required to start a *distributed run*, it is encouraged to install Circus:

```
$ pip install circus
```

Or if you build Loads from the source, simply run:

```
$ make build_extras
```

Then you can read *Distributed test*.

1.2 Writing load tests

Writing load tests can be done with Requests, WebTest or ws4py. Loads provides a test case class that includes bridges to the three libraries.

> **Warning:** Loads uses Gevent to spawn concurrent users. Most of the time, Gevent will play nicely with your tests and make sure that they are run asynchronously - but in case Loads is not sending the load it's supposed to, it probably means some of your code is blocking the Gevent loop.
> Read *Writing asynchronous tests* to troubleshoot this issue.

1.2.1 Using Requests

Requests is a popular library to query an HTTP service, and is widely used in the Python community.

Let's say you want to load test the Elastic Search root page that's running on your local host.

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

```
from loads.case import TestCase

class TestWebSite(TestCase):

    def test_es(self):
        res = self.session.get('http://localhost:9200')
        self.assertEqual(res.status_code, 200)
```

The *TestCase* class provided by **Loads** 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 that your test works as expected.

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

```
$ bin/loads-runner example.TestWebSite.test_es -u 100 --hits 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 virtual users.

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

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

That's 1200 hits total.

You can use all Requests API to GET, PUT, DELETE, POST or do whatever you need on the service.

Don't forget to control all responses with assertions, so you can catch any issue that may occur on your service on high load.

To do this, use the unit test assert methods provided by Python. Most services will break with 500s errors when they can't cope with the load.

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 provided in the test case class.

Run the echo_server.py file located in Loads' 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):

        results = []

        def callback(m):
            results.append(m.data)
```

```
        ws = self.create_ws('ws://localhost:9000/ws',
                            protocols=['chat', 'http-only'],
                            callback=callback)
        ws.send('something')
        ws.receive()
        ws.send('happened')
        ws.receive()

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

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

See ws4py documentation for more info.

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 ;).

WebTest is really handy to exercise an HTTP service because it includes tools to easily control the responses status code and content.

You just need to use **app** instead of **session** in the test case class. **app** is a *webtest.TestApp* object, providing all the APIs to interact with an HTTP service:

```
from loads.case import TestCase

class TestWebSite(TestCase):

    def test_something(self):
        self.assertTrue('tarek' 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://blog.ziade.org
```

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 Adding custom metrics

You can use the **incr_counter** method in your test case to increment a counter. This is useful if you want to count the number of times a particular event happens.

In this example, the **tarek-was-there** counter will be incremented everytime the test is successful:

```
from loads.case import TestCase

class TestWebSite(TestCase):
```

```
    def test_something(self):
        self.assertTrue('tarek' in self.app.get('/'))
        self.incr_counter('tarek-was-there')
```

At the end of the test, you will be able to know how many times the counter was incremented.

1.3 Loads commands

Loads comes with 3 commands:

1. **load-runner**: the test runner
2. **loads-broker**: the master when running in distributed mode
3. **loads-agent**: the slave when running in distributed mode

1.3.1 loads-runner

loads-runner only mandatory argument is the *fully qualified name* (FQN) of the test method you want to call. *Fully Qualified Name* means that you provide a string that contains the package, sub packages, module, class and test name, all separated by dots - like an import statement.

For example, if your test module is called *test_server* and located in the *tests* package under the *project* package, the FQN for the *test_es* method in the *TestSite* class will be: **project.tests.test_server.TestSite.test_es**.

Running that test is done with:

```
$ loads-runner project.tests.test_server.TestSite.test_es
```

Loads imports the *test_server* module, instanciantes the *TestSite* class, then call the *test_es* method.

Every other option in *loads-runner* is optional, as the command provides defaults to run the test locally a single time with a single user.

This is useful for trying out a test, but to do a real load test, you will need more options.

Common options

Loads has 3 options you can use to define how much of a load you are sending.

- **-u / –users**: the number of concurrent users spawned for the test. You can provide several values separated by ":". Example: "10:20:30". In that case, Loads will spawn 10, then 20 then 30 users. That's what we call a **cycle** Defaults to 1.
- **–hits**: the number of times the test is executed per user. Like for **–users**, you can provide a *cycle*. The number of tests will be the cartesian product of hits by users. Defaults to 1.
- **-d / –duration**: number of seconds the test is run. This option is mutually exclusive with –hits. You will have to decide if you want to run test a certain number of times or for a certain amount of time. When using *duration*, Loads will loop on the test for each user indefinitely. Defaults to None.

Distributed mode options

When running in distributed mode, the most important options are **–broker** and **–agents**, that will let you point a cluster and define the number of nodes to use, but they are other options that may be useful to run your test.

- **-b / –broker**: Point to the broker's ZMQ front socket. defaults to *ipc:///tmp/loads-front.ipc*. We call it *front* socket because the broker has many other socket, and this one is used by the broker to receive all queries that are then dispatched to backends.
- **-a / –agents**: Defines the number of nodes you want to use to run a load test. This option triggers the distributed mode: if you use it, then *Loads* makes the assumption that you are in distributed mode. When you use agents, the users/hits/duration options will be sent to each agent, so the number of tests that will be executed is the cartesian product = [agents x users x (hits or duration)]. Defaults to None.
- **–test-dir**: when provided, the broker will ask every agent to create the directory on the slave box, and chdir to it. For example, you can pass a value like "/tmp/mytest". Loads will create all intermediate directories if they don't exist.
- **–python-dep**: points a Python project name, that will be installed on each slave prior to running the test, using pip. You can provide the usual version notation if needed. You can also provide several *–python-dep* arguments if you need them - or None.
- **–include-file**: give that option a filename or a directory and Loads will recursively upload the files on each slave. That option needs to be used with *–test-dir*. You can also use glob-style patterns to include several files. Something like: "*.py" will include all Python files in the current directory. Like *–python-deps** you can provide one or several options, or None.
- **–detach**: when this flag is used, the runner will call the broker and quit immediatly. The test will be running in detached mode. This can also be done by hitting Ctrl-C after the run has started.
- **–attach**: use this flag to reattach a console to an existing run. If several runs are active, you will have to choose which one to get attached to.
- **–ping-broker**: use this flag to display the broker status: the number of workers, the active runs and the broker options.
- **–purge-broker**: use this flag to stop all active runs.
- **–health-check**: use this flag to run an empty test on every agent. This option is useful to verify that every agent is up and responsive.
- **–observer**: you can point a fully qualified name that will be called from the broker when the test is over. *Loads* provides built-in observers: *irc* and *email*. They will send a message on a given channel or to a given recipient when the test is done.
- **–no-patching**: use this flag to prevent Gevent monkey patching. see *Writing asynchronous tests* for more information on this.

Configuration file

Instead of typing a very long command line, you can create a configuration file and have Loads use it.

Here's an example:

```
[loads]
fqn = example.TestWebSite.test_something
agents = 4

include_file = *.py
            pushtest

test_dir = /tmp/tests
users = 5
duration = 30
```

```
observer = irc
detach = True
```

In this example, we're pushing a load test accross 4 agents.

Using this config file is done with the **–config** option:

```
$ loads-runner --config config.ini
```

1.3.2 loads-broker

XXX

1.3.3 loads-agent

XXX

1.4 Distributed test

> **Warning:** Loads comes with no security whatsoever. If you run a broker, make sure that you secure access to the box because any code can be executed remotely through the loads-runner command.
> The best way to avoid any issue is to protect the server access through firewall rules.

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 spread across several boxes called nodes.

A typical setup is to run a broker on a box, with a few agents, and have dedicated boxes to run many agents. This setup is called a **Loads cluster**.

The typical limiting factor will be the number of sockets each box will be able to open on each node that will call your service. This number can be tweaked by changing the **ulimit** value to a higher number - like 8096. You can read this page for more tips on tweaking your servers.

1.4.1 Setting up a Loads cluster

To run a broker and some agents, we can use Circus - a process supervisor.

To install Circus you can use Pip:

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

If you have any trouble installing Circus, check out its documentation.

Once Circus is installed, you can run it against the provided **loads.ini** configuration file that's located in the Loads source repository in the **con/** directory:

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

This command will run 1 broker and 5 agents

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
```

Let's control that the cluster is functional by pinging the broker for its status:

```
$ bin/loads-runner --ping-broker
Broker running on pid 11154
5 agents registered
endpoints:
- publisher: ipc:///tmp/loads-publisher.ipc
- frontend: ipc:///tmp/loads-front.ipc
- register: ipc:///tmp/loads-reg.ipc
- receiver: ipc:///tmp/loads-broker-receiver.ipc
- heartbeat: ipc:///tmp/hb.ipc
- backend: ipc:///tmp/loads-back.ipc
Nothing is running right now
```

Let's use them now, with the **agents** option, with the example shown in *Writing load tests*:

```
$ 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?

To stop your cluster:

```
$ bin/circusctl quit
```

1.4.2 Adding more agents

XXX

1.4.3 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
```

1.5 Writing asynchronous tests

When **loads-runner** is executing your tests, it calls Gevent monkey patching to make the Python standard library cooperative.

That feature works well when you are making classical socket calls on a service, but some libraries are known to be incompatible with this behavior.

If you encounter some issues, you can deactivate the monkey patching with the **–no-patching** option and work things out manually.

1.5.1 Asynchronous web sockets

XXX

1.6 Design

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 run several of agents to hammer the service you want to load test. That's what we call the *distributed mode*.

Alternatively, you may want to run things from a single process, just to smoke test your service - or simply because you don't need to send a huge load. That's the *non-distributed* mode.

1.6.1 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.

1.6.2 What happens during a distributed run

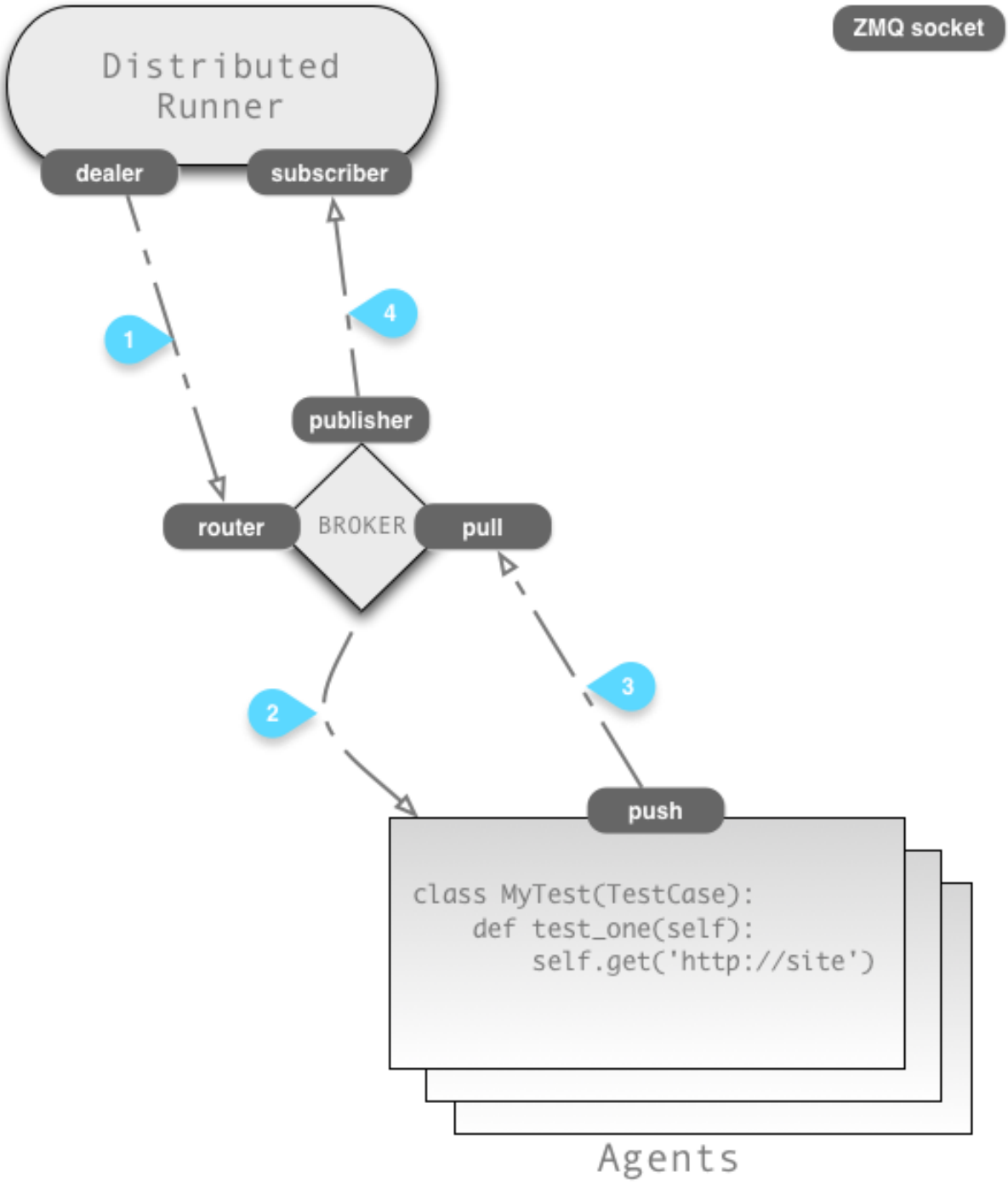
When you run in distributed mode, you have a distributed runner (the *broker*) which defer the execution to one or several *agents*.

These agents are simple runners that will redirect their results to the broker using a ZeroMQ stream.

The relay can be found in the *loads/relay.py* module. It's a drop-in replacement for the *TestResult* class.

The broker gets back the results and store them in a database, then publishes them in turn, so the caller can get them.

A schema might help you to get things right:



All the communication is handled through ZeroMQ sockets, as you can see in the diagram.

In more details:

1. The distributed runner sends a message to the broker, asking it to run the tests on N agents.
2. The broker selects available agents and send them the job. Every agent starts a loads-runner instance in slave mode
3. The broker receives the results back from every agent.

4. The broker publishes the results so the distributed runner can get them.

1.6.3 The TestResult object

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

Hopefully, people that are used to write Python tests should be familiar with these API and they can use Loads' TestCase class in lieu of the usual *unittest.TestCase class*.

Loads' *TestCase* class is located in *loads/case.py*, and implements the same APIs than unittest's one.

The extra benefit of keeping our class compatible with unittest is that you can also run Loads tests with third party test runners like Nose or unittest2. They will be recognized as classical functional tests.

1.6.4 The Runners

As mentioned earlier, Loads provides more than one Runner implementation. Each of these classes share an implicit interface, allowing us to have more than one implementation of a runner.

For instance, you can see that we have a *Runner* and a *DistributedRunner*. The former is a "local" runner: it is able to run the tests locally and output the results directly or proxy them to a ZMQ backend.

The latter, the *DistributedRunner*, runs the tests on a Loads cluster, using a *broker* and one or more *agents*.

A runner has a constructor, which takes an `arg` argument, a dict, with all the startup options it may need. It is then started with the *execute* method.

If you want to add a specific behavior, you may need to subclass *LocalRunner* and change its *_execute* method (notice how it's prefixed with an underscore). This method is where all the actual execution happens.

1.7 Plugin-in external runners

By default, Loads is built in a way which makes it possible to have tests runners written in any languages. To do that, it uses ZeroMQ to do communicate.

This document describes the protocol you need to implement if you want to create your own runner.

1.7.1 Existing implementations

Currently, there is only a Python implementation and a JavaScript implementation (using the Mocha test framework for the latter). The JS runner is provided in a separate project named loads.js.

If you have implemented your own runner, feel free to submit us a patch or a pull request.

1.7.2 The protocol

Each message sent to Loads needs to 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 agent id with every message. Each agent id should be different from each other.

- You should also send the id of the run.
- Additionally, each message contains a **data_type**, with the type of the data.

A message generally looks like this:

```
{
    data_type: 'something',
    agent_id: '1',
    run_id: '1234',
    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.

With loads, you can run cycle of runs. For instance, if you pass 10:50:100 for the users, it will start with 10 users in parallel, and then 50 and finally 100.

Loads status contains information about the current number of users we have to run for the cycle we are in (50, for instance), and the user we are currently taking care of (could be 12). Same applies for the hits.

It contains, in this order:

- hits: the number of hits that will be running on this cycle.
- user: the number of users that will be running on this cycle.
- current_hit: the current hit we're running.
- current_user: the current user 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.

1.7.3 Data types

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)

You should **not** send the *startTestRun* and *stopTestRun* messages.

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.

1.8 Glossary

agent, agents A process, running on a distant machine, waiting to run the tests (send the requests) to create some actual load on the system under test.

broker The process which routes the jobs to the agents. It contains a broker controller and a broker database.

distributed run, distributed test When a test is run in distributed mode, meaning that all the commands goes trought the broker and one or more agents.

observers Some python code in charge of notifying people via various channels (irc, email, etc.). Observers are running on the broker.

outputs Some python code in charge of generating reports (in real time or not) about a Loads run. Outputs are running on the client side.

runner The code that will actually run the test suite for you.

system under test The website or service you want to test with Loads.

virtual users When running a test, you can chose the number of users you want to have in parallel. This is called the number of virtual users.

workers Each agent can spawn a number of workers, which does the actual queries. The agent isn't sending itself the queries, it creates a worker which does it instead.

1.9 Contributing

Loads is an open source project and we welcome contributors.

We usually hang on IRC on Freenode in the #mozilla-circus channel.

- Source code: https://github.com/mozilla-services/loads
- Documentation: http://loads.readthedocs.org

1.10 Outputs

By default, Loads reports the status of the load in real time on the 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.

At the moment, we're supporting the following outputs:

- **file** if you want to have all the calls reported to a file. This is useful for later analysis but doesn't do much.
- **funkload** generates a funkload compatible report. These reports can then be used with the the *fl-build-report* <*filename*> command-line tool to generate reports about the load.
- **null** in case you want to silent the outputs.