Locust for Performance Testing

Now-a-days, Enterprises moved away from traditional load testing approach to more of a 'code to test' approach by switching to locust.io for performance testing. Locust is also useful for IoT performance testing as it can integrate various python libraries to include various IoT protocols like MQTT. since the client behaviour can be entirely specified by us using regular python its highly flexible tool and also allows distributed deployment for high scale testing. It lets you write tests against your web application which mimic your user's behaviour or any type of IoT device behaviour. This ensures the cloud applications we write are able to handle a high load and remain performant.

Why another tool??

Why do we need to look into this new tool when there are lots of well-established tools available in the market for decades like JMeter, LoadRunner, ApacheBench, Gatling, Tsung, etc…

Some major features of Locust are:

- Allows a user to write code in plain python and not limited to UI/XML based ones making it highly flexible and allowing to recreate exact input behaviours easily.
- Ability to include various Python libraries making it easy to include things like IoT protocol vary easily
- Supports distributed user.
- Supports default HTML + JS based reporting.
- Open to test other systems by writing clients.
- Supports user to write code in other languages using **locust slave**, like boomer if needed.
- Open source under the MIT license.
- Locust is completely event-based, and therefore it's possible to support thousands of concurrent users on a single machine. In contrast to many other event-based apps, it doesn't use callbacks. Instead, it uses light-weight processes, through gevent.
- Green-let threads allows a user to write an expressive scenario in Python without complicating your code with callbacks.

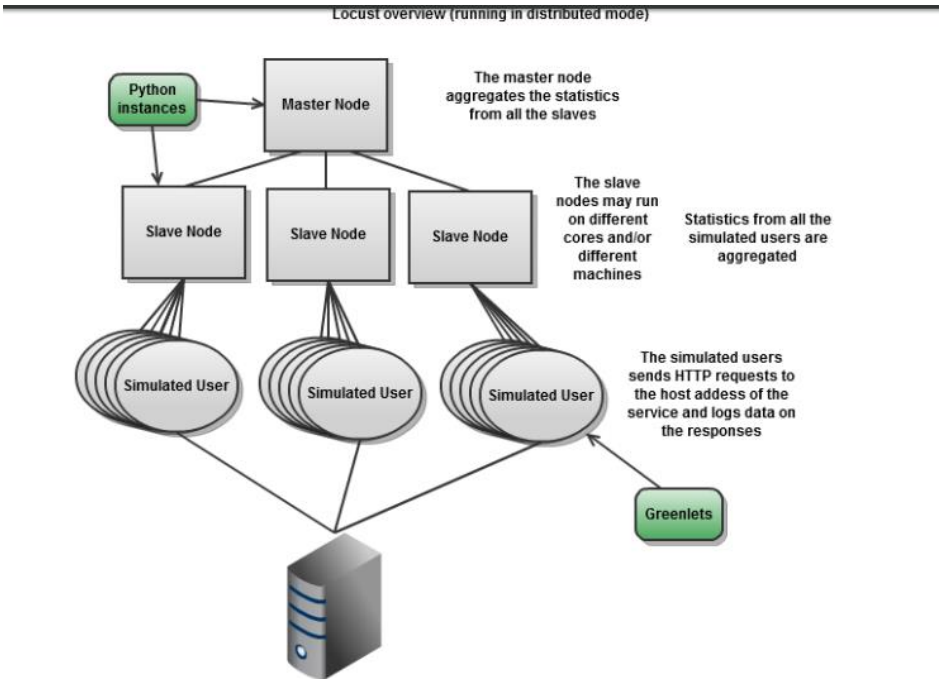
A quick comparison between Locust and Jmeter is as below

Topics	**Locust**	**Jmeter**
Platform/Operating System	Any	Any
Open Source/License	Yes. MIT Licensed	Yes. Apache 2.0 Licensed

Supported Protocols	Primarily HTTP and supports everything using a custom script.	HTTP,JDBC, FTP, LDAP, SMTP and more.
Support of "Test as Code"	Strong(Python)	Moderate(Java)
Concurrent Users	Efficient and powerful	Moderate
Resources Consumption	Less (1X)	More(GUI – 30X Non-GUI – 14X)
Ramp-up Flexibility	Customization using code	Yes.
GUI	No	Yes
Execution Monitoring	Console Web	Console File Graphs Desktop client Custom plugins
Recording Functionality	No	Yes
Easy to Use VCS	Yes.	No

Locust Architecture:

The execution flow starts with the main function of the python file(main.py). Locust class is parsed and associated weight tasks(functions) are parsed and stored. Locust browser client works with python standard library urllib2 to manage the HTTP requests. The "browser" that is used by the simulated users is defined in clients.py. When a request is executed the time until a response has been received is recorded. Locust is implemented using events. When a request is finished, an event is fired to allow the stats module to store a response time or possibly a failure. Each simulated users run in Green-let threads. A green-let is actually not a real thread but it makes sense to think of it as such.



How To Write Tests

One of the nicest features of Locust is that configuration is done via "Plain Old Python." You simply create a file named `locustfile.py` and all configuration for your load tester and its tests is done there.

Here's an example `locustfile.py`, which defines a simple user behavior which consists of a single "task" which gets a specific webpage:

```
from locust import HttpLocust, TaskSet, task

class UserBehavior(TaskSet):

    @task
    def get_something(self):
        self.client.get("/something")

class WebsiteUser(HttpLocust):
    task_set = UserBehavior
```

We can add a second task as follows:

```
class UserBehavior(TaskSet):

    @task
    def get_something(self):
        self.client.get("/something")
```

```
    @task
    def get_something_else(self):
        self.client.get("/something-else")
```

When the above `UserBehavior` is run, Locust will randomly choose between each of the tasks and run them. If you want to **weight** different tasks differently, so that one is run twice as much as the other, you can add weighting as follows:

```
class UserBehavior(TaskSet):

    @task(2)
    def get_something(self):
        self.client.get("/something")

    @task(1)
    def get_something_else(self):
        self.client.get("/something-else")
```

These task weights are ratios across all defined tasks, so here `get_something` would happen twice as often during the load test.

You can also write tasks which compose other tasks, to perform a sequential or serial set of tasks in a specific order. This lets you define a user flow through multiple requests. For example:[^1]

```
class UserBehavior(TaskSet):

    @task
    def get_something(self):
        self.client.get("/something")

    @task
    def get_something_else(self):
        self.client.get("/something-else")

    @task
    def get_two_things(self):
        self.get_something()
        self.get_something_else()
```

A `TaskSet` class can optionally declare an `on_start` function, which is called when a simulated user starts executing that `TaskSet` class. This can be used to log in or apply credentials once before beginning the load test:

```
class UserBehavior(TaskSet):

    def on_start(self):
        self.client.post("/login", {
            'username': 'foo', 'password': 'bar'
        })
```

```
    @task
    def get_something(self):
        self.client.get("/something")
```

Running Locally

To run Locust, you run the `locust` command in the same directory as your `locustfile.py`:

```
$ locust --host=http://localhost:5000
```

To update your locust.io tests, you have to open up the Python file in your favorite IDE or even a text editor. To update your JMeter tests, you have to (first download and then) open up the clunky JMeter GUI - or take a wild stab at a direct XML update.

When you have a large number of similar tests - which most of the micro-services are beginning to get to - the need for common code/components in your tests becomes apparent. This is easy to implement in your locust.io tests; your performance tests can become just as modular as any of the other code in your repository. Not so easy in JMeter. There's little scope for re usability there, especially across services, where there is essentially zero reuse.

Running Distributed

Running locally is fine for basic testing and getting started with Locust, but most applications will not receive a significant load if you're just running it from your local machine. It's almost always necessary to run it in distributed mode. This is easy to do with a couple AWS nodes.

After installing Locust and moving your `locustfile.py` to all nodes, you can start the "master" node:

```
$ locust --host=http://localhost:5000 --master
```

Then start any "slave" nodes, giving them a reference to the master node:

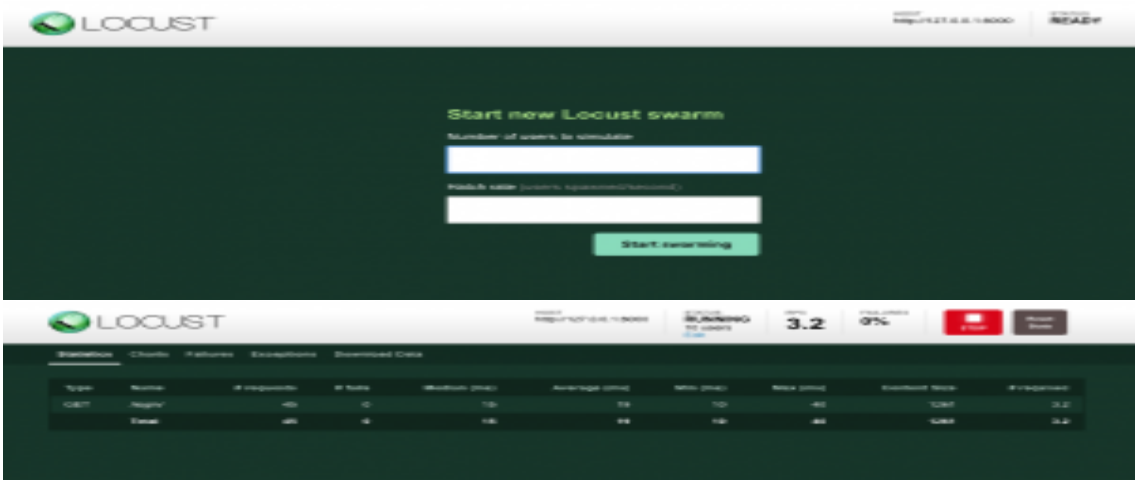
```
$ locust --host=http://localhost:5000 --slave\
  --master-host=192.168.10.100
```

Downsides

As nice as Locust is, there are a small number of downsides. First, statistics are pretty bad or nonexistent for the results of the test, and could be better (e.g., there are no graphs, and you can't correlate an increased failure rate with a higher load without running multiple tests). Second, it's sometimes hard to get details about error responses beyond the status. Finally, it's non-trivial to do non-HTTP or non-RESTful requests (although this is admittedly rare).

The UI

The interface will open a page with two inputs. In the first of them, “Number of users to simulate”, we have to enter the number of users (e.g. 10.0). in the second, “Hatch rate”, you specify the number of users that will be created in a second (e.g. 1). When running a test with given values, you will see the test statistic.



Locust was created to solve some specific issues of current existing performance tools and it did it great. Using this framework and some Python experience you can write performance scripts pretty fast, store them within your Python project, spend minimum time on maintenance without additional GUI applications and simulate thousands of test users even on very slow machines.

If your code base is Python, it’s a shoe-in for the best tool you can be using, due to the opportunity to pull in data, models, or domain logic from your existing code base, but even if you’re not using Python, you can easily integrate it. Put your code to the test!