Locust

Release 0.8

September 19, 2017

Contents

Everything you need to know about Locust

About locust

Locust is a scalable load testing framework written in Python

- **Website**: http://locust.io
- **Source code**: http://github.com/locustio/locust
- **Twitter**: @locustio

CHAPTER 1

What is Locust?

Locust is an easy-to-use, distributed, user load testing tool. It is intended for load-testing web sites (or other systems) and figuring out how many concurrent users a system can handle.

The idea is that during a test, a swarm of locusts will attack your website. The behavior of each locust (or test user if you will) is defined by you and the swarming process is monitored from a web UI in real-time. This will help you battle test and identify bottlenecks in your code before letting real users in.

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. Each locust swarming your site is actually running inside its own process (or greenlet, to be correct). This allows you to write very expressive scenarios in Python without complicating your code with callbacks.

Features

- **Write user test scenarios in plain-old Python**

  No need for clunky UIs or bloated XML—just code as you normally would. Based on coroutines instead of callbacks, your code looks and behaves like normal, blocking Python code.

- **Distributed & Scalable - supports hundreds of thousands of users**

  Locust supports running load tests distributed over multiple machines. Being event-based, even one Locust node can handle thousands of users in a single process. Part of the reason behind this is that even if you simulate that many users, not all are actively hitting your system. Often, users are idle figuring out what to do next. Requests per second != number of users online.

- **Web-based UI**

  Locust has a neat HTML+JS user interface that shows relevant test details in real-time. And since the UI is web-based, it's cross-platform and easily extendable.

- **Can test any system**

  Even though Locust is web-oriented, it can be used to test almost any system. Just write a client for what ever you wish to test and swarm it with locusts! It's super easy!

- **Hackable**

  Locust is small and very hackable and we intend to keep it that way. All heavy-lifting of evented I/O and coroutines are delegated to gevent. The brittleness of alternative testing tools was the reason we created Locust.

Background

Locust was created because we were fed up with existing solutions. None of them are solving the right problem and to me, they are missing the point. We've tried both Apache JMeter and Tsung. Both tools are quite OK to use; we've used the former many times benchmarking stuff at work. JMeter comes with a UI, which you might think for a second is a good thing. But you soon realize it's a PITA to "code" your testing scenarios through some point-and-click interface. Secondly, JMeter is thread-bound. This means for every user you want to simulate, you need a separate thread. Needless to say, benchmarking thousands of users on a single machine just isn't feasible.

Tsung, on the other hand, does not have these thread issues as it's written in Erlang. It can make use of the light-weight processes offered by BEAM itself and happily scale up. But when it comes to defining the test scenarios, Tsung is as limited as JMeter. It offers an XML-based DSL to define how a user should behave when testing. I guess you can imagine the horror of "coding" this. Displaying any sorts of graphs or reports when completed requires you to post-process the log files generated from the test. Only then can you get an understanding of how the test went.

Anyway, we've tried to address these issues when creating Locust. Hopefully none of the above pain points should exist.

I guess you could say we're really just trying to scratch our own itch here. We hope others will find it as useful as we do.

Authors

- Jonatan Heyman (@jonatanheyman on Twitter)
- Carl Byström (@cgbystrom on Twitter)
- Joakim Hamrén (@Jahaaja on Twitter)
- Hugo Heyman (@hugoheyman on Twitter)

License

CHAPTER 2

Installation

Locust is available on PyPI and can be installed through pip or easy_install

```
pip install locustio
```

or:

```
easy_install locustio
```

When Locust is installed, a **locust** command should be available in your shell (if you're not using virtualenv—which you should—make sure your python script directory is on your path).

To see available options, run:

```
locust --help
```

Supported Python Versions

Locust supports Python 2.7, 3.3, 3.4, 3.5, and 3.6.

Installing ZeroMQ

If you intend to run Locust distributed across multiple processes/machines, we recommend you to also install **pyzmq**:

```
pip install pyzmq
```

or:

```
easy_install pyzmq
```

Installing Locust on Windows

The easiest way to get Locust running on Windows is to first install pre built binary packages for gevent and greenlet and then follow the above instructions.

You can find an unofficial collection of pre built python packages for windows here: http://www.lfd.uci.edu/~gohlke/pythonlibs/

Note: Running Locust on Windows should work fine for developing and testing your load testing scripts. However, when running large scale tests, it's recommended that you do that on Linux machines, since gevent's performance under Windows is poor.

Installing Locust on OS X

The following is currently the shortest path to installing gevent on OS X using Homebrew.

1. Install Homebrew.
2. Install libev (dependency for gevent):

```
brew install libev
```

3. Then follow the above instructions.

Increasing Maximum Number of Open Files Limit

Every HTTP connection on a machine opens a new file (technically a file descriptor). Operating systems may set a low limit for the maximum number of files that can be open. If the limit is less than the number of simulated users in a test, failures will occur.

Increase the operating system's default maximum number of files limit to a number higher than the number of simulated users you'll want to run. How to do this depends on the operating system in use.

CHAPTER 3

Quick start

Example locustfile.py

Below is a quick little example of a simple **locustfile.py**:

```
from locust import HttpLocust, TaskSet

def login(l):
    l.client.post("/login", {"username":"ellen_key", "password":"education"})

def index(l):
    l.client.get("/")

def profile(l):
    l.client.get("/profile")

class UserBehavior(TaskSet):
    tasks = {index: 2, profile: 1}

    def on_start(self):
        login(self)

class WebsiteUser(HttpLocust):
    task_set = UserBehavior
    min_wait = 5000
    max_wait = 9000
```

Here we define a number of Locust tasks, which are normal Python callables that take one argument (a *Locust* class instance). These tasks are gathered under a *TaskSet* class in the *tasks* attribute. Then we have a *HttpLocust* class which represents a user, where we define how long a simulated user should wait between executing tasks, as well as what *TaskSet* class should define the user's "behaviour". :py:class:‘TaskSet <locust.core.TaskSet>‘s can be nested.

The *HttpLocust* class inherits from the *Locust* class, and it adds a client attribute which is an instance of *HttpSession* that can be used to make HTTP requests.

Another way we could declare tasks, which is usually more convenient, is to use the `@task` decorator. The following code is equivalent to the above:

```
from locust import HttpLocust, TaskSet, task

class UserBehavior(TaskSet):
    def on_start(self):
        """ on_start is called when a Locust start before any task is scheduled """
        self.login()

    def login(self):
        self.client.post("/login", {"username":"ellen_key", "password":"education"})

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

    @task(1)
    def profile(self):
        self.client.get("/profile")

class WebsiteUser(HttpLocust):
    task_set = UserBehavior
    min_wait = 5000
    max_wait = 9000
```

The *Locust* class (as well as *HttpLocust* since it's a subclass) also allows one to specify minimum and maximum wait time—per simulated user—between the execution of tasks (*min_wait* and *max_wait*) as well as other user behaviours.

Start Locust

To run Locust with the above Locust file, if it was named *locustfile.py* and located in the current working directory, we could run:

```
locust --host=http://example.com
```

If the Locust file is located under a subdirectory and/or named different than *locustfile.py*, specify it using `-f`:

```
locust -f locust_files/my_locust_file.py --host=http://example.com
```

To run Locust distributed across multiple processes we would start a master process by specifying `--master`:

```
locust -f locust_files/my_locust_file.py --master --host=http://example.com
```

and then we would start an arbitrary number of slave processes:

```
locust -f locust_files/my_locust_file.py --slave --host=http://example.com
```

If we want to run Locust distributed on multiple machines we would also have to specify the master host when starting the slaves (this is not needed when running Locust distributed on a single machine, since the master host defaults to 127.0.0.1):

```
locust -f locust_files/my_locust_file.py --slave --master-host=192.168.0.100 --
↪host=http://example.com
```

You may wish to consume your Locust results via a csv file. In this case, there are two ways to do this.

First, when running the webserver, you can retrieve a csv from `localhost:8089/stats/requests/csv` and `localhost:8089/stats/distribution/csv`. Second you can run Locust with a flag which will periodically save the csv file. This is particularly useful if you plan on running Locust in an automated way with the `--no-web` flag:

```
locust -f locust_files/my_locust_file.py --csv=foobar --no-web -n10 -c1
```

You can also customize how frequently this is written if you desire faster (or slower) writing:

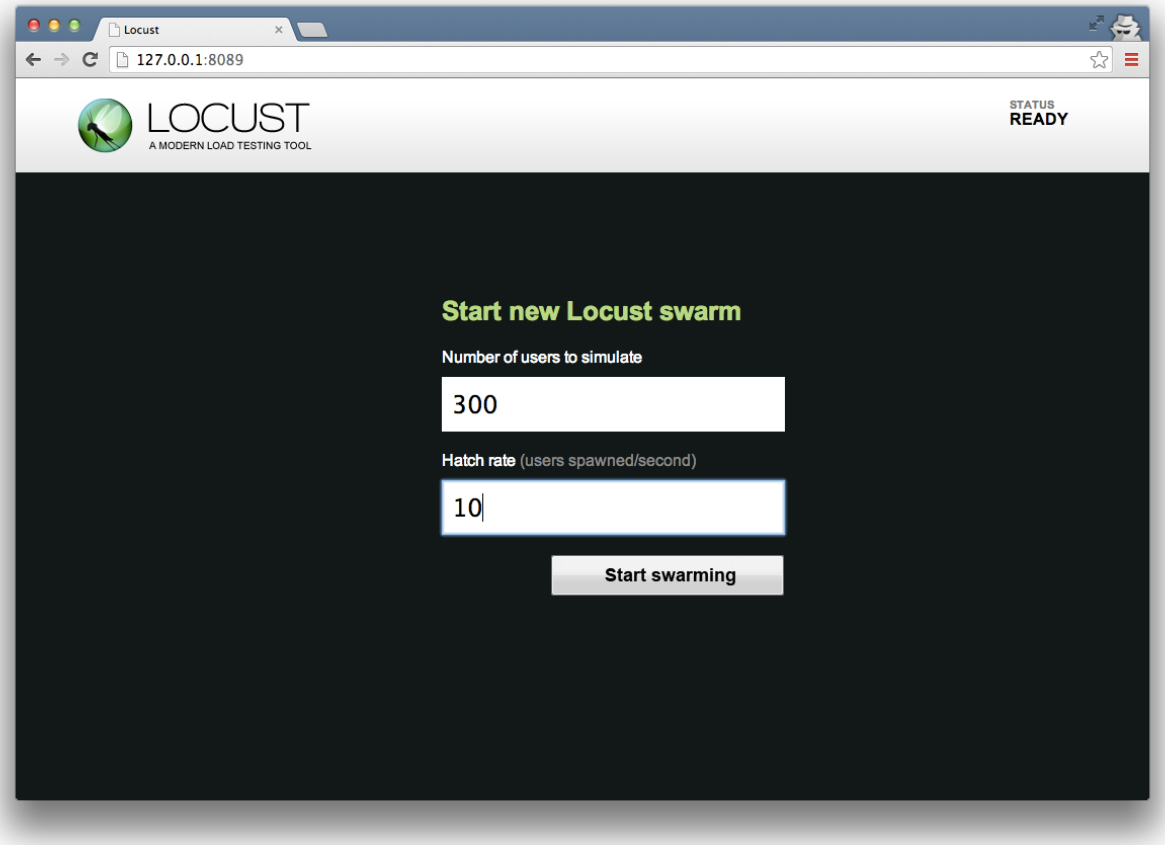
```
import locust.stats
locust.stats.CSV_STATS_INTERVAL_SEC = 5 # default is 2 seconds
```

To see all available options type:

```
locust --help
```

Open up Locust's web interface

Once you've started Locust using one of the above command lines, you should open up a browser and point it to http://127.0.0.1:8089 (if you are running Locust locally). Then you should be greeted with something like this:



CHAPTER 4

Writing a locustfile

A locustfile is a normal python file. The only requirement is that it declares at least one class— let's call it the locust class—that inherits from the class Locust.

The Locust class

A locust class represents one user (or a swarming locust if you will). Locust will spawn (hatch) one instance of the locust class for each user that is being simulated. There are a few attributes that a locust class should typically define.

The `task_set` attribute

The *task_set* attribute should point to a *TaskSet* class which defines the behaviour of the user and is described in more detail below.

The *min_wait* and *max_wait* attributes

In addition to the task_set attribute, one usually wants to declare the *min_wait* and *max_wait* attributes. These are the minimum and maximum time respectively, in milliseconds, that a simulated user will wait between executing each task. *min_wait* and *max_wait* default to 1000, and therefore a locust will always wait 1 second between each task if *min_wait* and *max_wait* are not declared.

With the following locustfile, each user would wait between 5 and 15 seconds between tasks:

```
from locust import Locust, TaskSet, task

class MyTaskSet(TaskSet):
    @task
    def my_task(self):
        print "executing my_task"

class MyLocust(Locust):
```

```
    task_set = MyTaskSet
    min_wait = 5000
    max_wait = 15000
```

The *min_wait* and *max_wait* attributes can also be overridden in a TaskSet class.

The *weight* attribute

You can run two locusts from the same file like so:

```
locust -f locust_file.py WebUserLocust MobileUserLocust
```

If you wish to make one of these locusts execute more often you can set a weight attribute on those classes. Say for example, web users are three times more likely than mobile users:

```
class WebUserLocust(Locust):
    weight = 3
    ....

class MobileUserLocust(Locust):
    weight = 1
    ....
```

The *host* attribute

The host attribute is a URL prefix (i.e. "http://google.com") to the host that is to be loaded. Usually, this is specified on the command line, using the `--host` option, when locust is started. If one declares a host attribute in the locust class, it will be used in the case when no `--host` is specified on the command line.

TaskSet class

If the Locust class represents a swarming locust, you could say that the TaskSet class represents the brain of the locust. Each Locust class must have a *task_set* attribute set, that points to a TaskSet.

A TaskSet is, like its name suggests, a collection of tasks. These tasks are normal python callables and—if we were load-testing an auction website—could do stuff like "loading the start page", "searching for some product" and "making a bid".

When a load test is started, each instance of the spawned Locust classes will start executing their TaskSet. What happens then is that each TaskSet will pick one of its tasks and call it. It will then wait a number of milliseconds, chosen at random between the Locust class' *min_wait* and *max_wait* attributes (unless min_wait/max_wait have been defined directly under the TaskSet, in which case it will use its own values instead). Then it will again pick a new task to be called, wait again, and so on.

Declaring tasks

The typical way of declaring tasks for a TaskSet it to use the *task* decorator.

Here is an example:

```
from locust import Locust, TaskSet, task

class MyTaskSet(TaskSet):
    @task
    def my_task(self):
        print "Locust instance (%r) executing my_task" % (self.locust)

class MyLocust(Locust):
    task_set = MyTaskSet
```

@task takes an optional weight argument that can be used to specify the task's execution ratio. In the following example *task2* will be executed twice as much as *task1*:

```
from locust import Locust, TaskSet, task

class MyTaskSet(TaskSet):
    min_wait = 5000
    max_wait = 15000

    @task(3)
    def task1(self):
        pass

    @task(6)
    def task2(self):
        pass

class MyLocust(Locust):
    task_set = MyTaskSet
```

tasks attribute

Using the @task decorator to declare tasks is a convenience, and usually the best way to do it. However, it's also possible to define the tasks of a TaskSet by setting the *tasks* attribute (using the @task decorator will actually just populate the *tasks* attribute).

The *tasks* attribute is either a list of python callables, or a <*callable : int*> dict. The tasks are python callables that receive one argument—the TaskSet class instance that is executing the task. Here is an extremely simple example of a locustfile (this locustfile won't actually load test anything):

```
from locust import Locust, TaskSet

def my_task(l):
    pass

class MyTaskSet(TaskSet):
    tasks = [my_task]

class MyLocust(Locust):
    task_set = MyTaskSet
```

If the tasks attribute is specified as a list, each time a task is to be performed, it will be randomly chosen from the *tasks* attribute. If however, *tasks* is a dict—with callables as keys and ints as values—the task that is to be executed will be chosen at random but with the int as ratio. So with a tasks that looks like this:

```
{my_task: 3, another_task:1}
```

my_task would be 3 times more likely to be executed than *another_task*.

TaskSets can be nested

A very important property of TaskSets is that they can be nested, because real websites are usually built up in an hierarchical way, with multiple sub-sections. Nesting TaskSets will therefore allow us to define a behaviour that simulates users in a more realistic way. For example we could define TaskSets with the following structure:

- Main user behaviour
  - Index page
  - Forum page
    - Read thread
    - Reply
    - New thread
    - View next page
  - Browse categories
  - Watch movie
  - Filter movies
  - About page

The way you nest TaskSets is just like when you specify a task using the **tasks** attribute, but instead of referring to a python function, you refer to another TaskSet:

```
class ForumPage(TaskSet):
    @task(20)
    def read_thread(self):
        pass

    @task(1)
    def new_thread(self):
        pass

    @task(5)
    def stop(self):
        self.interrupt()

class UserBehaviour(TaskSet):
    tasks = {ForumPage:10}

    @task
    def index(self):
        pass
```

So in the above example, if the ForumPage would get selected for execution when the UserBehaviour TaskSet is executing, then the ForumPage TaskSet would start executing. The ForumPage TaskSet would then pick one of its own tasks, execute it, wait, and so on.

There is one important thing to note about the above example, and that is the call to self.interrupt() in the ForumPage's stop method. What this does is essentially to stop executing the ForumPage task set and the execution will continue

in the UserBehaviour instance. If we didn't have a call to the *interrupt()* method somewhere in ForumPage, the Locust would never stop running the ForumPage task once it has started. But by having the interrupt function, we can—together with task weighting—define how likely it is that a simulated user leaves the forum.

It's also possible to declare a nested TaskSet, inline in a class, using the *@task* decorator, just like when declaring normal tasks:

```
class MyTaskSet(TaskSet):
    @task
    class SubTaskSet(TaskSet):
        @task
        def my_task(self):
            pass
```

The on_start function

A TaskSet class can optionally declare an `on_start` function. If so, that function is called when a simulated user starts executing that TaskSet class.

Referencing the Locust instance, or the parent TaskSet instance

A TaskSet instance will have the attribute *locust* point to its Locust instance, and the attribute *parent* point to its parent TaskSet (it will point to the Locust instance, in the base TaskSet).

Making HTTP requests

So far, we've only covered the task scheduling part of a Locust user. In order to actually load test a system we need to make HTTP requests. To help us do this, the *HttpLocust* class exists. When using this class, each instance gets a `client` attribute which will be an instance of `HttpSession` which can be used to make HTTP requests.

class HttpLocust

Represents an HTTP "user" which is to be hatched and attack the system that is to be load tested.

The behaviour of this user is defined by the task_set attribute, which should point to a *TaskSet* class.

This class creates a *client* attribute on instantiation which is an HTTP client with support for keeping a user session between requests.

client = None

Instance of HttpSession that is created upon instantiation of Locust. The client support cookies, and therefore keeps the session between HTTP requests.

When inheriting from the HttpLocust class, we can use its client attribute to make HTTP requests against the server. Here is an example of a locust file that can be used to load test a site with two URLs; **/** and **/about/**:

```
from locust import HttpLocust, TaskSet, task

class MyTaskSet(TaskSet):
    @task(2)
    def index(self):
        self.client.get("/")

    @task(1)
    def about(self):
        self.client.get("/about/")
```

```
class MyLocust(HttpLocust):
    task_set = MyTaskSet
    min_wait = 5000
    max_wait = 15000
```

Using the above Locust class, each simulated user will wait between 5 and 15 seconds between the requests, and **/** will be requested twice as much as **/about/**.

The attentive reader will find it odd that we can reference the HttpSession instance using *self.client* inside the TaskSet, and not *self.locust.client*. We can do this because the *TaskSet* class has a convenience property called client that simply returns self.locust.client.

Using the HTTP client

Each instance of HttpLocust has an instance of *HttpSession* in the *client* attribute. The HttpSession class is actually a subclass of `requests.Session` and can be used to make HTTP requests, that will be reported to Locust's statistics, using the *get*, *post*, *put*, *delete*, *head*, *patch* and *options* methods. The HttpSession instance will preserve cookies between requests so that it can be used to log in to websites and keep a session between requests. The client attribute can also be referenced from the Locust instance's TaskSet instances so that it's easy to retrieve the client and make HTTP requests from within your tasks.

Here's a simple example that makes a GET request to the */about* path (in this case we assume *self* is an instance of a *TaskSet* or *HttpLocust* class:

```
response = self.client.get("/about")
print "Response status code:", response.status_code
print "Response content:", response.content
```

And here's an example making a POST request:

```
response = self.client.post("/login", {"username":"testuser", "password":"secret"})
```

Safe mode

The HTTP client is configured to run in safe_mode. What this does is that any request that fails due to a connection error, timeout, or similar will not raise an exception, but rather return an empty dummy Response object. The request will be reported as a failure in Locust's statistics. The returned dummy Response's *content* attribute will be set to None, and its *status_code* will be 0.

Manually controlling if a request should be considered successful or a failure

By default, requests are marked as failed requests unless the HTTP response code is OK (2xx). Most of the time, this default is what you want. Sometimes however—for example when testing a URL endpoint that you expect to return 404, or testing a badly designed system that might return *200 OK* even though an error occurred—there's a need for manually controlling if locust should consider a request as a success or a failure.

One can mark requests as failed, even when the response code is OK, by using the *catch_response* argument and a with statement:

```
with client.get("/", catch_response=True) as response:
    if response.content != "Success":
        response.failure("Got wrong response")
```

Just as one can mark requests with OK response codes as failures, one can also use **catch_response** argument together with a *with* statement to make requests that resulted in an HTTP error code still be reported as a success in the statistics:

```
with client.get("/does_not_exist/", catch_response=True) as response:
    if response.status_code == 404:
        response.success()
```

Grouping requests to URLs with dynamic parameters

It's very common for websites to have pages whose URLs contain some kind of dynamic parameter(s). Often it makes sense to group these URLs together in Locust's statistics. This can be done by passing a *name* argument to the *HttpSession's* different request methods.

Example:

```
# Statistics for these requests will be grouped under: /blog/?id=[id]
for i in range(10):
    client.get("/blog?id=%i" % i, name="/blog?id=[id]")
```

Common libraries

Often, people wish to group multiple locustfiles that share common libraries. In that case, it is important to define the *project root* to be the directory where you invoke locust, and it is suggested that all locustfiles live somewhere beneath the project root.

A flat file structure works out of the box:

- project root
  - commonlib_config.py
  - commonlib_auth.py
  - locustfile_web_app.py
  - locustfile_api.py
  - locustfile_ecommerce.py

The locustfiles may import common libraries using, e.g. `import commonlib_auth`. This approach does not cleanly separate common libraries from locust files, however.

Subdirectories can be a cleaner approach (see example below), but locust will only import modules relative to the directory in which the running locustfile is placed. If you wish to import from your project root (i.e. the location where you are running the locust command), make sure to write `sys.path.append(os.getcwd())` in your locust file(s) before importing any common libraries—this will make the project root (i.e. the current working directory) importable.

- project root
  - __init__.py
  - common/
    - __init__.py
    - config.py
    - auth.py

- locustfiles/
  - __init__.py
  - web_app.py
  - api.py
  - ecommerce.py

With the above project structure, your locust files can import common libraries using:

```
sys.path.append(os.getcwd())
import common.auth
```

CHAPTER 5

Using other languages

A Locust master and a Locust slave communicate by exchanging msgpack messages, which is supported by many languages. So, you can write your Locust tasks in any languages you like. For convenience, some libraries do the job as a slave runner. They run your Locust tasks, and report to master regularly.

Boomer

Boomer is a Locust slave runner written in golang.

CHAPTER 6

Running Locust distributed

Once a single machine isn't enough to simulate the number of users that you need, Locust supports running load tests distributed across multiple machines.

To do this, you start one instance of Locust in master mode using the `--master` flag. This is the instance that will be running Locust's web interface where you start the test and see live statistics. The master node doesn't simulate any users itself. Instead you have to start one or —most likely—multiple slave Locust nodes using the `--slave` flag, together with the `--master-host` (to specify the IP/hostname of the master node).

A common set up is to run a single master on one machine, and then run one slave instance per processor core, on the slave machines.

Note: Both the master and each slave machine, must have a copy of the locust test scripts when running Locust distributed.

Example

To start locust in master mode:

```
locust -f my_locustfile.py --master
```

And then on each slave (replace `192.168.0.14` with IP of the master machine):

```
locust -f my_locustfile.py --slave --master-host=192.168.0.14
```

Options

--master

Sets locust in master mode. The web interface will run on this node.

--slave

Sets locust in slave mode.

--master-host=X.X.X.X

Optionally used together with `--slave` to set the hostname/IP of the master node (defaults to 127.0.0.1)

--master-port=5557

Optionally used together with `--slave` to set the port number of the master node (defaults to 5557). Note that locust will use the port specified, as well as the port number +1. So if 5557 is used, locust will use both port 5557 and 5558.

--master-bind-host=X.X.X.X

Optionally used together with `--master`. Determines what network interface that the master node will bind to. Defaults to * (all available interfaces).

--master-bind-port=5557

Optionally used together with `--master`. Determines what network ports that the master node will listen to. Defaults to 5557. Note that locust will use the port specified, as well as the port number +1. So if 5557 is used, locust will use both port 5557 and 5558.

CHAPTER 7

Testing other systems using custom clients

Locust was built with HTTP as its main target. However, it can easily be extended to load test any request/response based system, by writing a custom client that triggers *request_success* and *request_failure* events.

Sample XML-RPC Locust client

Here is an example of a Locust class, **XmlRpcLocust**, which provides an XML-RPC client, **XmlRpcClient**, and tracks all requests made:

```
import time
import xmlrpclib

from locust import Locust, TaskSet, events, task


class XmlRpcClient(xmlrpclib.ServerProxy):
    """
    Simple, sample XML RPC client implementation that wraps xmlrpclib.ServerProxy and
    fires locust events on request_success and request_failure, so that all requests
    gets tracked in locust's statistics.
    """
    def __getattr__(self, name):
        func = xmlrpclib.ServerProxy.__getattr__(self, name)
        def wrapper(*args, **kwargs):
            start_time = time.time()
            try:
                result = func(*args, **kwargs)
            except xmlrpclib.Fault as e:
                total_time = int((time.time() - start_time) * 1000)
                events.request_failure.fire(request_type="xmlrpc", name=name,␣
↪response_time=total_time, exception=e)
            else:
                total_time = int((time.time() - start_time) * 1000)
                events.request_success.fire(request_type="xmlrpc", name=name,␣
↪response_time=total_time, response_length=0)
```

```
                # In this example, I've hardcoded response_length=0. If we would want␣
↪the response length to be
                # reported correctly in the statistics, we would probably need to␣
↪hook in at a lower level

        return wrapper


class XmlRpcLocust(Locust):
    """
    This is the abstract Locust class which should be subclassed. It provides an XML-
↪RPC client
    that can be used to make XML-RPC requests that will be tracked in Locust's␣
↪statistics.
    """
    def __init__(self, *args, **kwargs):
        super(XmlRpcLocust, self).__init__(*args, **kwargs)
        self.client = XmlRpcClient(self.host)


class ApiUser(XmlRpcLocust):

    host = "http://127.0.0.1:8877/"
    min_wait = 100
    max_wait = 1000

    class task_set(TaskSet):
        @task(10)
        def get_time(self):
            self.client.get_time()

        @task(5)
        def get_random_number(self):
            self.client.get_random_number(0, 100)
```

If you've written Locust tests before, you'll recognize the class called *ApiUser* which is a normal Locust class that has a *TaskSet* class with *tasks* in its *task_set* attribute. However, the *ApiUser* inherits from *XmlRpcLocust* that you can see right above ApiUser. The *XmlRpcLocust* class provides an instance of XmlRpcClient under the *client* attribute. The *XmlRpcClient* is a wrapper around the standard library's `xmlrpclib.ServerProxy`. It basically just proxies the function calls, but with the important addition of firing *locust.events.request_success* and *locust.events.request_failure* events, which will make all calls reported in Locust's statistics.

Here's an implementation of an XML-RPC server that would work as a server for the code above:

```
import random
import time
from SimpleXMLRPCServer import SimpleXMLRPCServer


def get_time():
    time.sleep(random.random())
    return time.time()

def get_random_number(low, high):
    time.sleep(random.random())
    return random.randint(low, high)

server = SimpleXMLRPCServer(("localhost", 8877))
```

```
print "Listening on port 8877..."
server.register_function(get_time, "get_time")
server.register_function(get_random_number, "get_random_number")
server.serve_forever()
```

CHAPTER 8

API

Locust class

class Locust

Represents a "user" which is to be hatched and attack the system that is to be load tested.

The behaviour of this user is defined by the task_set attribute, which should point to a *TaskSet* class.

This class should usually be subclassed by a class that defines some kind of client. For example when load testing an HTTP system, you probably want to use the *HttpLocust* class.

max_wait = 1000

Maximum waiting time between the execution of locust tasks

min_wait = 1000

Minimum waiting time between the execution of locust tasks

task_set = None

TaskSet class that defines the execution behaviour of this locust

weight = 10

Probability of locust being chosen. The higher the weight, the greater is the chance of it being chosen.

HttpLocust class

class HttpLocust

Represents an HTTP "user" which is to be hatched and attack the system that is to be load tested.

The behaviour of this user is defined by the task_set attribute, which should point to a *TaskSet* class.

This class creates a *client* attribute on instantiation which is an HTTP client with support for keeping a user session between requests.

client = None
Instance of HttpSession that is created upon instantiation of Locust. The client support cookies, and therefore keeps the session between HTTP requests.

TaskSet class

class TaskSet (*parent*)
Class defining a set of tasks that a Locust user will execute.

When a TaskSet starts running, it will pick a task from the *tasks* attribute, execute it, call it's wait function which will sleep a random number between *min_wait* and *max_wait* milliseconds. It will then schedule another task for execution and so on.

TaskSets can be nested, which means that a TaskSet's *tasks* attribute can contain another TaskSet. If the nested TaskSet it scheduled to be executed, it will be instantiated and called from the current executing TaskSet. Execution in the currently running TaskSet will then be handed over to the nested TaskSet which will continue to run until it throws an InterruptTaskSet exception, which is done when *TaskSet.interrupt()* is called. (execution will then continue in the first TaskSet).

client
Reference to the `client` attribute of the root Locust instance.

interrupt (*reschedule=True*)
Interrupt the TaskSet and hand over execution control back to the parent TaskSet.

If *reschedule* is True (default), the parent Locust will immediately re-schedule, and execute, a new task

This method should not be called by the root TaskSet (the one that is immediately, attached to the Locust class' *task_set* attribute), but rather in nested TaskSet classes further down the hierarchy.

locust = None
Will refer to the root Locust class instance when the TaskSet has been instantiated

max_wait = None
Maximum waiting time between the execution of locust tasks. Can be used to override the max_wait defined in the root Locust class, which will be used if not set on the TaskSet.

min_wait = None
Minimum waiting time between the execution of locust tasks. Can be used to override the min_wait defined in the root Locust class, which will be used if not set on the TaskSet.

parent = None
Will refer to the parent TaskSet, or Locust, class instance when the TaskSet has been instantiated. Useful for nested TaskSet classes.

schedule_task (*task_callable*, *args=None*, *kwargs=None*, *first=False*)
Add a task to the Locust's task execution queue.

Arguments:

- task_callable: Locust task to schedule
- args: Arguments that will be passed to the task callable
- kwargs: Dict of keyword arguments that will be passed to the task callable.
- first: Optional keyword argument. If True, the task will be put first in the queue.

tasks = []
List with python callables that represents a locust user task.

If tasks is a list, the task to be performed will be picked randomly.

If tasks is a *(callable,int)* list of two-tuples, or a {callable:int} dict, the task to be performed will be picked randomly, but each task will be weighted according to it's corresponding int value. So in the following case *ThreadPage* will be fifteen times more likely to be picked than *write_post*:

```
class ForumPage(TaskSet):
    tasks = {ThreadPage:15, write_post:1}
```

task decorator

task(*weight=1*)

Used as a convenience decorator to be able to declare tasks for a TaskSet inline in the class. Example:

```
class ForumPage(TaskSet):
    @task(100)
    def read_thread(self):
        pass

    @task(7)
    def create_thread(self):
        pass
```

HttpSession class

class HttpSession(*base_url*, **args*, ***kwargs*)

Class for performing web requests and holding (session-) cookies between requests (in order to be able to log in and out of websites). Each request is logged so that locust can display statistics.

This is a slightly extended version of python-request's `requests.Session` class and mostly this class works exactly the same. However the methods for making requests (get, post, delete, put, head, options, patch, request) can now take a *url* argument that's only the path part of the URL, in which case the host part of the URL will be prepended with the HttpSession.base_url which is normally inherited from a Locust class' host property.

Each of the methods for making requests also takes two additional optional arguments which are Locust specific and doesn't exist in python-requests. These are:

Parameters

- **name** – (optional) An argument that can be specified to use as label in Locust's statistics instead of the URL path. This can be used to group different URL's that are requested into a single entry in Locust's statistics.
- **catch_response** – (optional) Boolean argument that, if set, can be used to make a request return a context manager to work as argument to a with statement. This will allow the request to be marked as a fail based on the content of the response, even if the response code is ok (2xx). The opposite also works, one can use catch_response to catch a request and then mark it as successful even if the response code was not (i.e 500 or 404).

delete(*url*, ***kwargs*)

Sends a DELETE request. Returns `Response` object.

Parameters

- **url** – URL for the new `Request` object.

- ****kwargs** – Optional arguments that `request` takes.

Return type requests.Response

get(*url*, ***kwargs*)
Sends a GET request. Returns `Response` object.

Parameters

- **url** – URL for the new `Request` object.
- ****kwargs** – Optional arguments that `request` takes.

Return type requests.Response

head(*url*, ***kwargs*)
Sends a HEAD request. Returns `Response` object.

Parameters

- **url** – URL for the new `Request` object.
- ****kwargs** – Optional arguments that `request` takes.

Return type requests.Response

options(*url*, ***kwargs*)
Sends a OPTIONS request. Returns `Response` object.

Parameters

- **url** – URL for the new `Request` object.
- ****kwargs** – Optional arguments that `request` takes.

Return type requests.Response

patch(*url*, *data=None*, ***kwargs*)
Sends a PATCH request. Returns `Response` object.

Parameters

- **url** – URL for the new `Request` object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the `Request`.
- ****kwargs** – Optional arguments that `request` takes.

Return type requests.Response

post(*url*, *data=None*, *json=None*, ***kwargs*)
Sends a POST request. Returns `Response` object.

Parameters

- **url** – URL for the new `Request` object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the `Request`.
- **json** – (optional) json to send in the body of the `Request`.
- ****kwargs** – Optional arguments that `request` takes.

Return type requests.Response

put(*url*, *data=None*, ***kwargs*)
Sends a PUT request. Returns `Response` object.

Parameters

- **url** – URL for the new `Request` object.
- **data** – (optional) Dictionary, bytes, or file-like object to send in the body of the `Request`.
- ****kwargs** – Optional arguments that `request` takes.

Return type requests.Response

request(*method*, *url*, *name=None*, *catch_response=False*, ***kwargs*)
Constructs and sends a `requests.Request`. Returns `requests.Response` object.

Parameters

- **method** – method for the new `Request` object.
- **url** – URL for the new `Request` object.
- **name** – (optional) An argument that can be specified to use as label in Locust's statistics instead of the URL path. This can be used to group different URL's that are requested into a single entry in Locust's statistics.
- **catch_response** – (optional) Boolean argument that, if set, can be used to make a request return a context manager to work as argument to a with statement. This will allow the request to be marked as a fail based on the content of the response, even if the response code is ok (2xx). The opposite also works, one can use catch_response to catch a request and then mark it as successful even if the response code was not (i.e 500 or 404).
- **params** – (optional) Dictionary or bytes to be sent in the query string for the `Request`.
- **data** – (optional) Dictionary or bytes to send in the body of the `Request`.
- **headers** – (optional) Dictionary of HTTP Headers to send with the `Request`.
- **cookies** – (optional) Dict or CookieJar object to send with the `Request`.
- **files** – (optional) Dictionary of `'filename': file-like-objects` for multipart encoding upload.
- **auth** – (optional) Auth tuple or callable to enable Basic/Digest/Custom HTTP Auth.
- **timeout** (*float or tuple*) – (optional) How long to wait for the server to send data before giving up, as a float, or a (connect timeout, read timeout) tuple.
- **allow_redirects** (*bool*) – (optional) Set to True by default.
- **proxies** – (optional) Dictionary mapping protocol to the URL of the proxy.
- **stream** – (optional) whether to immediately download the response content. Defaults to `False`.
- **verify** – (optional) if `True`, the SSL cert will be verified. A CA_BUNDLE path can also be provided.
- **cert** – (optional) if String, path to ssl client cert file (.pem). If Tuple, ('cert', 'key') pair.

Response class

This class actually resides in the python-requests library, since that's what Locust is using to make HTTP requests, but it's included in the API docs for locust since it's so central when writing locust load tests. You can also look at the `Response` class at the requests documentation.

class Response

The Response object, which contains a server's response to an HTTP request.

apparent_encoding

The apparent encoding, provided by the chardet library.

close()

Releases the connection back to the pool. Once this method has been called the underlying raw object must not be accessed again.

Note: Should not normally need to be called explicitly.

content

Content of the response, in bytes.

cookies = None

A CookieJar of Cookies the server sent back.

elapsed = None

The amount of time elapsed between sending the request and the arrival of the response (as a timedelta). This property specifically measures the time taken between sending the first byte of the request and finishing parsing the headers. It is therefore unaffected by consuming the response content or the value of the stream keyword argument.

encoding = None

Encoding to decode with when accessing r.text.

headers = None

Case-insensitive Dictionary of Response Headers. For example, headers['content-encoding'] will return the value of a 'Content-Encoding' response header.

history = None

A list of Response objects from the history of the Request. Any redirect responses will end up here. The list is sorted from the oldest to the most recent request.

is_permanent_redirect

True if this Response one of the permanent versions of redirect.

is_redirect

True if this Response is a well-formed HTTP redirect that could have been processed automatically (by Session.resolve_redirects()).

iter_content(*chunk_size=1*, *decode_unicode=False*)

Iterates over the response data. When stream=True is set on the request, this avoids reading the content at once into memory for large responses. The chunk size is the number of bytes it should read into memory. This is not necessarily the length of each item returned as decoding can take place.

chunk_size must be of type int or None. A value of None will function differently depending on the value of *stream*. stream=True will read data as it arrives in whatever size the chunks are received. If stream=False, data is returned as a single chunk.

If decode_unicode is True, content will be decoded using the best available encoding based on the response.

iter_lines(*chunk_size=512*, *decode_unicode=None*, *delimiter=None*)

Iterates over the response data, one line at a time. When stream=True is set on the request, this avoids reading the content at once into memory for large responses.

Note: This method is not reentrant safe.

---

json(***kwargs*)
Returns the json-encoded content of a response, if any.

Parameters ****kwargs** – Optional arguments that `json.loads` takes.

Raises **ValueError** – If the response body does not contain valid json.

links
Returns the parsed header links of the response, if any.

next
Returns a PreparedRequest for the next request in a redirect chain, if there is one.

ok
Returns True if `status_code` is less than 400.

This attribute checks if the status code of the response is between 400 and 600 to see if there was a client error or a server error. If the status code, is between 200 and 400, this will return True. This is **not** a check to see if the response code is `200 OK`.

raise_for_status()
Raises stored `HTTPError`, if one occurred.

raw = None
File-like object representation of response (for advanced usage). Use of `raw` requires that `stream=True` be set on the request.

reason = None
Textual reason of responded HTTP Status, e.g. "Not Found" or "OK".

request = None
The `PreparedRequest` object to which this is a response.

status_code = None
Integer Code of responded HTTP Status, e.g. 404 or 200.

text
Content of the response, in unicode.

If Response.encoding is None, encoding will be guessed using `chardet`.

The encoding of the response content is determined based solely on HTTP headers, following RFC 2616 to the letter. If you can take advantage of non-HTTP knowledge to make a better guess at the encoding, you should set `r.encoding` appropriately before accessing this property.

url = None
Final URL location of Response.

ResponseContextManager class

class ResponseContextManager(*response*)
A Response class that also acts as a context manager that provides the ability to manually control if an HTTP request should be marked as successful or a failure in Locust's statistics

This class is a subclass of `Response` with two additional methods: *`success`* and *`failure`*.

failure(*exc*)
Report the response as a failure.

exc can be either a python exception, or a string in which case it will be wrapped inside a CatchResponseError.

Example:

```
with self.client.get("/", catch_response=True) as response:
    if response.content == "":
        response.failure("No data")
```

success()
Report the response as successful

Example:

```
with self.client.get("/does/not/exist", catch_response=True) as response:
    if response.status_code == 404:
        response.success()
```

InterruptTaskSet Exception

exception InterruptTaskSet(*reschedule=True*)
Exception that will interrupt a Locust when thrown inside a task

Event hooks

The event hooks are instances of the **locust.events.EventHook** class:

class EventHook
Simple event class used to provide hooks for different types of events in Locust.

Here's how to use the EventHook class:

```
my_event = EventHook()
def on_my_event(a, b, **kw):
    print "Event was fired with arguments: %s, %s" % (a, b)
my_event += on_my_event
my_event.fire(a="foo", b="bar")
```

Available hooks

The following event hooks are available under the **locust.events** module:

request_success = <locust.events.EventHook object>
request_success is fired when a request is completed successfully.

Listeners should take the following arguments:

- *request_type*: Request type method used
- *name*: Path to the URL that was called (or override name if it was used in the call to the client)
- *response_time*: Response time in milliseconds
- *response_length*: Content-length of the response

request_failure = <locust.events.EventHook object>
request_failure is fired when a request fails

Event is fired with the following arguments:

- *request_type*: Request type method used
- *name*: Path to the URL that was called (or override name if it was used in the call to the client)
- *response_time*: Time in milliseconds until exception was thrown
- *exception*: Exception instance that was thrown

locust_error = <locust.events.EventHook object>

locust_error is fired when an exception occurs inside the execution of a Locust class.

Event is fired with the following arguments:

- *locust_instance*: Locust class instance where the exception occurred
- *exception*: Exception that was thrown
- *tb*: Traceback object (from sys.exc_info()[2])

report_to_master = <locust.events.EventHook object>

report_to_master is used when Locust is running in –slave mode. It can be used to attach data to the dicts that are regularly sent to the master. It's fired regularly when a report is to be sent to the master server.

Note that the keys "stats" and "errors" are used by Locust and shouldn't be overridden.

Event is fired with the following arguments:

- *client_id*: The client id of the running locust process.
- *data*: Data dict that can be modified in order to attach data that should be sent to the master.

slave_report = <locust.events.EventHook object>

slave_report is used when Locust is running in –master mode and is fired when the master server receives a report from a Locust slave server.

This event can be used to aggregate data from the locust slave servers.

Event is fired with following arguments:

- *client_id*: Client id of the reporting locust slave
- *data*: Data dict with the data from the slave node

hatch_complete = <locust.events.EventHook object>

hatch_complete is fired when all locust users has been spawned.

Event is fire with the following arguments:

- *user_count*: Number of users that was hatched

quitting = <locust.events.EventHook object>

quitting is fired when the locust process in exiting

CHAPTER 9

Extending Locust

Locust comes with a number of events that provides hooks for extending locust in different ways.

Event listeners can be registered at the module level in a locust file. Here's an example:

```
from locust import events

def my_success_handler(request_type, name, response_time, response_length, **kw):
    print "Successfully fetched: %s" % (name)

events.request_success += my_success_handler
```

Note: It's highly recommended that you add a wildcard keyword argument in your listeners (the **kw in the code above), to prevent your code from breaking if new arguments are added in a future version.

See also:

To see all available event, please see *Event hooks*.

Adding Web Routes

Locust uses Flask to serve the web UI and therefore it is easy to add web end-points to the web UI. Just import the Flask app in your locustfile and set up a new route:

```
from locust import web

@web.app.route("/added_page")
def my_added_page():
    return "Another page"
```

You should now be able to start locust and browse to http://127.0.0.1:8089/added_page

CHAPTER 10

Changelog

0.8

- Support Python 3
- Dropped support for Python 2.6
- Added *–no-reset-stats* option for controling if the statistics should be reset once the hatching is complete
- Added charts to the web UI for requests per second, average response time, and number of simulated users.
- Updated the design of the web UI.
- Added ability to write a CSV file for results via command line flag
- Added the URL of the host that is currently being tested to the web UI.
- We now also apply gevent's monkey patching of threads. This fixes an issue when using Locust to test Cassandra (https://github.com/locustio/locust/issues/569).
- Various bug fixes and improvements

0.7.5

- Use version 1.1.1 of gevent. Fixes an install issue on certain versions of python.

0.7.4

- Use a newer version of requests, which fixed an issue for users with older versions of requests getting ConnectionErrors (https://github.com/locustio/locust/issues/273).
- Various fixes to documentation.

0.7.3

- Fixed bug where POST requests (and other methods as well) got incorrectly reported as GET requests, if the request resulted in a redirect.
- Added ability to download exceptions in CSV format. Download links has also been moved to it's own tab in the web UI.

0.7.2

- Locust now returns an exit code of 1 when any failed requests were reported.
- When making an HTTP request to an endpoint that responds with a redirect, the original URL that was requested is now used as the name for that entry in the statistics (unless an explicit override is specified through the *name* argument). Previously, the last URL in the redirect chain was used to label the request(s) in the statistics.
- Fixed bug which caused only the time of the last request in a redirect chain to be included in the reported time.
- Fixed bug which caused the download time of the request body not to be included in the reported response time.
- Fixed bug that occurred on some linux dists that were tampering with the python-requests system package (removing dependencies which requests is bundling). This bug only occured when installing Locust in the python system packages, and not when using virtualenv.
- Various minor fixes and improvements.

0.7.1

- Exceptions that occurs within TaskSets are now catched by default.
- Fixed bug which caused Min response time to always be 0 after all locusts had been hatched and the statistics had been reset.
- Minor UI improvements in the web interface.
- Handle messages from "zombie" slaves by ignoring the message and making a log entry in the master process.

0.7

HTTP client functionality moved to HttpLocust

Previously, the Locust class instantiated a `HttpSession` under the client attribute that was used to make HTTP requests. This funcionality has now been moved into the `HttpLocust` class, in an effort to make it more obvious how one can use Locust to *load test non-HTTP systems*.

To make existing locust scripts compatible with the new version you should make your locust classes inherit from HttpLocust instead of the base Locust class.

msgpack for serializing master/slave data

Locust now uses msgpack for serializing data that is sent between a master node and it's slaves. This adresses a possible attack that can be used to execute code remote, if one has access to the internal locust ports that are used for master-slave communication. The reason for this exploit was due to the fact that pickle was used.

> **Warning:** Anyone who uses an older version should make sure that their Locust machines are not publicly accessible on port 5557 and 5558. Also, one should never run Locust as root.

Anyone who uses the *report_to_master* and *slave_report* events, needs to make sure that any data that is attached to the slave reports is serializable by msgpack.

requests updated to version 2.2

Locust updated requests to the latest major release.

Note: Requests 1.0 introduced some major API changes (and 2.0 just a few). Please check if you are using any internal features and check the documentation: Migrating to 1.x and Migrationg to 2.x

---

gevent updated to version 1.0

gevent 1.0 has now been released and Locust has been updated accordingly.

Big refactoring of request statistics code

Refactored `RequestStats`.

- Created `StatsEntry` which represents a single stats entry (URL).

Previously the `RequestStats` was actually doing two different things:

- It was holding track of the aggregated stats from all requests
- It was holding the stats for single stats entries.

Now RequestStats should be instantiated and holds the global stats, as well as a dict of StatsEntry instances which holds the stats for single stats entries (URLs)

Removed support for avg_wait

Previously one could specify avg_wait to `TaskSet` and `Locust` that Locust would try to strive to. However this can be sufficiently accomplished by using min_wait and max_wait for most use-cases. Therefore we've decided to remove the avg_wait as it's use-case is not clear or just too narrow to be in the Locust core.

Removed support for ramping

Previously one could tell Locust, using the –ramp option, to try to find a stable client count that the target host could handle, but it's been broken and undocumented for quite a while so we've decided to remove it from the locust core and perhaps have it reappear as a plugin in the future.

Locust Event hooks now takes keyword argument

When *Extending Locust* by listening to *Event hooks*, the listener functions should now expect the arguments to be passed in as keyword arguments. It's also highly recommended to add an extra wildcard keyword arguments to listener functions, since they're then less likely to break if extra arguments are added to that event in some future version. For example:

```
from locust import events

def on_request(request_type, name, response_time, response_length, **kw):
    print "Got request!"

locust.events.request_success += on_request
```

The *method* and *path* arguments to `request_success` and `request_failure` are now called *request_type* and *name*, since it's less HTTP specific.

Other changes

- You can now specify the port on which to run the web host
- Various code cleanups
- Updated gevent/zmq libraries
- Switched to unittest2 discovery
- Added option –only-summary to only output the summary to the console, thus disabling the periodic stats output.
- Locust will now make sure to spawn all the specified locusts in distributed mode, not just a multiple of the number of slaves.
- Fixed the broken Vagrant example.
- Fixed the broken events example (events.py).
- Fixed issue where the request column was not sortable in the web-ui.
- Minor styling of the statistics table in the web-ui.
- Added options to specify host and ports in distributed mode using –master-host, –master-port for the slaves, –master-bind-host, –master-bind-port for the master.
- Removed previously deprecated and obsolete classes WebLocust and SubLocust.
- Fixed so that also failed requests count, when specifying a maximum number of requests on the command line

0.6.2

- Made Locust compatible with gevent 1.0rc2. This allows user to step around a problem with running Locust under some versions of CentOS, that can be fixed by upgrading gevent to 1.0.
- Added `parent` attribute to TaskSet class that refers to the parent TaskSet, or Locust, instance. Contributed by Aaron Daubman.

0.6.1

- Fixed bug that was causing problems when setting a maximum number of requests using the **-n** or **–num-request** command line parameter.

0.6

Warning: This version comes with non backward compatible changes to the API. Anyone who is currently using existing locust scripts and want to upgrade to 0.6 should read through these changes.

`SubLocust` replaced by `TaskSet` and `Locust` class behaviour changed

Locust classes does no longer control task scheduling and execution. Therefore, you no longer define tasks within Locust classes, instead the Locust class has a *task_set* attribute which should point to a *TaskSet* class. Tasks should now be defined in TaskSet classes, in the same way that was previously done in Locust and SubLocust classes. TaskSets can be nested just like SubLocust classes could.

So the following code for 0.5.1:

```
class User(Locust):
    min_wait = 10000
    max_wait = 120000

    @task(10)
    def index(self):
        self.client.get("/")

    @task(2)
    class AboutPage(SubLocust):
        min_wait = 10000
        max_wait = 120000

        def on_init(self):
            self.client.get("/about/")

        @task
        def team_page(self):
            self.client.get("/about/team/")

        @task
        def press_page(self):
            self.client.get("/about/press/")

        @task
        def stop(self):
            self.interrupt()
```

Should now be written like:

```
class BrowsePage(TaskSet):
    @task(10)
    def index(self):
        self.client.get("/")
```

```
    @task(2)
    class AboutPage(TaskSet):
        def on_init(self):
            self.client.get("/about/")

        @task
        def team_page(self):
            self.client.get("/about/team/")

        @task
        def press_page(self):
            self.client.get("/about/press/")

        @task
        def stop(self):
            self.interrupt()

class User(Locust):
    min_wait = 10000
    max_wait = 120000
    task_set = BrowsePage
```

Each TaskSet instance gets a *locust* attribute, which refers to the Locust class.

Locust now uses Requests

Locust's own HttpBrowser class (which was typically accessed through *self.client* from within a locust class) has been replaced by a thin wrapper around the requests library (http://python-requests.org). This comes with a number of advantages. Users can now take advantage of a well documented, well written, fully fledged library for making HTTP requests. However, it also comes with some small API changes wich will require users to update their existing load testing scripts.

Gzip encoding turned on by default

The HTTP client now sends headers for accepting gzip encoding by default. The **–gzip** command line argument has been removed and if someone want to disable the *Accept-Encoding* that the HTTP client uses, or any other HTTP headers you can do:

```
class MyWebUser(Locust):
    def on_start(self):
        self.client.headers = {"Accept-Encoding":""}
```

Improved HTTP client

Because of the switch to using python-requests in the HTTP client, the API for the client has also gotten a few changes.

- Additionally to the *get*, *post*, *put*, *delete* and *head* methods, the *HttpSession* class now also has *patch* and *options* methods.
- All arguments to the HTTP request methods, except for **url** and **data** should now be specified as keyword arguments. For example, previously one could specify headers using:

```
client.get("/path", {"User-Agent":"locust"}) # this will no longer work
```

And should now be specified like:

```
client.get("/path", headers={"User-Agent":"locust"})
```

- In general the whole HTTP client is now more powerful since it leverages on python-requests. Features that we're now able to use in Locust includes file upload, SSL, connection keep-alive, and more. See the python-requests documentation for more details.
- The new *HttpSession* class' methods now return python-request Response objects. This means that accessing the content of the response is no longer made using the **data** attribute, but instead the **content** attribute. The HTTP response code is now accessed through the **status_code** attribute, instead of the **code** attribute.

HttpSession methods' catch_response argument improved and allow_http_error argument removed

- When doing HTTP requests using the **catch_response** argument, the context manager that is returned now provides two functions, *success* and *failure* that can be used to manually control what the request should be reported as in Locust's statistics.

  class ResponseContextManager(*response*)
  A Response class that also acts as a context manager that provides the ability to manually control if an HTTP request should be marked as successful or a failure in Locust's statistics

  This class is a subclass of Response with two additional methods: *success* and *failure*.

  failure(*exc*)
  Report the response as a failure.

  exc can be either a python exception, or a string in which case it will be wrapped inside a CatchResponseError.

  Example:

  ```
  with self.client.get("/", catch_response=True) as response:
      if response.content == "":
          response.failure("No data")
  ```

  success()
  Report the response as successful

  Example:

  ```
  with self.client.get("/does/not/exist", catch_response=True) as response:
      if response.status_code == 404:
          response.success()
  ```

- The **allow_http_error** argument of the HTTP client's methods has been removed. Instead one can use the **catch_response** argument to get a context manager, which can be used together with a with statement.

  The following code in the previous Locust version:

  ```
  client.get("/does/not/exist", allow_http_error=True)
  ```

  Can instead now be written like:

  ```
  with client.get("/does/not/exist", catch_response=True) as response:
      response.success()
  ```

Other improvements and bug fixes

- Scheduled task callables can now take keyword arguments and not only normal function arguments.
- SubLocust classes that are scheduled using `locust.core.Locust.schedule_task()` can now take arguments and keyword arguments (available in *self.args* and *self.kwargs*).
- Fixed bug where the average content size would be zero when doing requests against a server that didn't set the content-length header (i.e. server that uses *Transfer-Encoding: chunked*)

Smaller API Changes

- The *require_once* decorator has been removed. It was an old legacy function that no longer fit into the current way of writing Locust tests, where tasks are either methods under a Locust class or SubLocust classes containing task methods.
- Changed signature of `locust.core.Locust.schedule_task()`. Previously all extra arguments that was given to the method was passed on to the task when it was called. It no longer accepts extra arguments. Instead, it takes an *args* argument (list) and a *kwargs* argument (dict) which are be passed to the task when it's called.
- Arguments for *request_success* event hook has been changed. Previously it took an HTTP Response instance as argument, but this has been changed to take the content-length of the response instead. This makes it easier to write custom clients for Locust.

0.5.1

- Fixed bug which caused –logfile and –loglevel command line parameters to not be respected when running locust without zeromq.

0.5

API changes

- Web inteface is now turned on by default. The **–web** command line option has been replaced by –no-web.
- *locust.events.request_success()* and *locust.events.request_failure()* now gets the HTTP method as the first argument.

Improvements and bug fixes

- Removed **–show-task-ratio-confluence** and added a **–show-task-ratio-json** option instead. The **–show-task-ratio-json** will output JSON data containing the task execution ratio for the locust "brain".
- The HTTP method used when a client requests a URL is now displayed in the web UI
- Some fixes and improvements in the stats exporting:
- A file name is now set (using content-disposition header) when downloading stats.
- The order of the column headers for request stats was wrong.
- Thanks Benjamin W. Smith, Jussi Kuosa and Samuele Pedroni!

0.4

API changes

- WebLocust class has been deprecated and is now called just Locust. The class that was previously called Locust is now called LocustBase.
- The *catch_http_error* argument to HttpClient.get() and HttpClient.post() has been renamed to *allow_http_error*.

Improvements and bug fixes

- Locust now uses python's logging module for all logging
- Added the ability to change the number of spawned users when a test is running, without having to restart the test.
- Experimental support for automatically ramping up and down the number of locust to find a maximum number of concurrent users (based on some parameters like response times and acceptable failure rate).
- Added support for failing requests based on the response data, even if the HTTP response was OK.
- Improved master node performance in order to not get bottlenecked when using enough slaves (>100)
- Minor improvements in web interface.
- Fixed missing template dir in MANIFEST file causing locust installed with "setup.py install" not to work.

Python Module Index

l

Index