Locust

Release 0.7.0

January 20, 2014
# Contents

1 What is Locust? ........................................ 3
   1.1 Features .......................................... 3
   1.2 Background ....................................... 4
   1.3 Authors .......................................... 4
   1.4 License .......................................... 4

2 Installation ......................................... 5
   2.1 Installing ZeroMQ .................................. 5
   2.2 Installing Locust on Windows ...................... 5
   2.3 Installing Locust on OS X ......................... 5

3 Quick start ......................................... 7

4 Writing a locustfile ................................. 9
   4.1 The Locust class .................................. 9
   4.2 TaskSet class .................................... 10
   4.3 Making HTTP requests ............................ 12
   4.4 Using the HTTP client ........................... 13

5 Testing other systems using custom clients ......... 15
   5.1 Sample XML-RPC Locust client ................... 15

6 API .................................................. 17
   6.1 Locust class ..................................... 17
   6.2 HttpLocust class ................................ 17
   6.3 TaskSet class ................................... 17
   6.4 task decorator .................................. 19
   6.5 HttpSession class ............................... 19
   6.6 Response class .................................. 21
   6.7 ResponseContextManager class .................. 22
   6.8 InterruptTaskSet Exception ....................... 23
   6.9 Event hooks ..................................... 23

7 Extending Locust .................................... 25
   7.1 Adding Web Routes ................................ 25

8 Changelog ......................................... 27
   8.1 0.7 ............................................. 27
   8.2 0.6.2 ........................................... 29
Python Module Index
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
LOCUST IS AN EASY-TO-USE, DISTRIBUTED, USER LOAD TESTING TOOL. 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 it’s own process (or greenlet, to be correct). This allows you to write very expressive scenarios in Python without complicating your code with callbacks.

1.1 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 (aka boomerang code) allows code to look and behave 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. Request per second != number of users online.

- **Web-based UI**
  
  Locust has a neat HTML+JS user interface that shows relevent 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.
1.2 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 UI which you might think for 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 lightweight 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 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 painpoints 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.

1.3 Authors

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

1.4 License

Open source licensed under the MIT license (see LICENSE file for details).
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
```

### 2.1 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
```

### 2.2 Installing Locust on Windows

The easiest way to get Locust running on Windows is to first install pre built binary packages for gevent (0.13) 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/

### 2.3 Installing Locust on OS X

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

1. Install [Homebrew](http://mxcl.github.com/homebrew/).
2. Install libevent (dependency for gevent):

```
brew install libevent
```

3. Then follow the above instructions.
Below is a quick little example of a simple `locustfile.py`:

```python
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 `task` 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”. TaskSets can be nested.

The `HttpLocust` class inherits from the `Locust` class, and it adds a client attribute which is an instance of `HttpSession <locust.clients.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:

```python
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):
```
```python
def index(self):
    self.client.get('/

def profile(self):
    self.client.get('/profile"
```

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 (\textit{min\_wait} and \textit{max\_wait}) as well as other user behaviours.

To run Locust with the above locust file, if it was named \textit{locustfile.py}, we could run (in the same directory as locust-file.py):

```
locust
```

or if the locust file is located elsewhere we could run:

```
locust -f ..\locust_files\my_locust_file.py
```

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
```

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

```
locust -f ..\locust_files\my_locust_file.py --slave
```

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
```

\textbf{Note:} To see all available options type

```
locust --help
```
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.

## 4.1 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.

### 4.1.1 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 details below.

### 4.1.2 The `min_wait` and `max_wait` attributes

Additionally to the `task_set` attribute, one usually want to declare the `min_wait` and `max_wait` attributes. These are the minimum and maximum time, in milliseconds, that a simulated user will wait between executing each task. `min_wait` and `max_wait` defaults to 1000, and therefore a locust will always wait 1 second between each task if `min_wait` and `max_wait` is not declared.

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

```python
from locust import Locust, TaskSet, task

class MyTaskSet(TaskSet):
    @task
    def my_task(l):
        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 overrides in a `TaskSet` class.
4.1.3 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.

4.2 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 it’s name suggests, a collection of tasks. These tasks are normal python callables and - if we were loadtesting 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 it’s tasks and call it. It will then wait a number of milliseconds, choosed at random between the Locust class’ **min_wait** and **max_wait** attributes (unless min_wait/max_wait has been defined directly under the TaskSet, in which case it will use it’s own values instead). Then it will again pick a new task which will be called, then wait again, and so on.

4.2.1 Declaring tasks

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

Here is an example:

```python
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 tasks’ execution ratio. In the following example **task2** will be executed twice as much as **task1**:

```python
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
```
4.2.2 task attribute

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

The `tasks` attribute which is either a list of python callables, or a `<callable : int>` dict. The tasks are python callables, that recieves 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):

```python
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:

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

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

4.2.3 TaskSets can be nested

A very important property of TaskSets are 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 refering to a python function, you point it 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 above example, if the ForumPage would get selected for execution when the UserBehaviour TaskSet is executing, is. that the ForumPage TaskSet would start executing. The ForumPage TaskSet would then pick one of it’s own task, execute it, then 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 that it will stop executing the ForumPage task set and the execution will continue in the UserBehaviour instance. If we wouldn’t have had 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

4.2.4 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.

4.2.5 Referencing the Locust instance, or the parent TaskSet instance

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

4.3 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.

**HttpLocust.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 it’s 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/:

```python
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.

4.4 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 reference 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):

```python
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"})

4.4.1 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 it’s status_code will be 0.

4.4.2 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 okay, by using the catch_response argument and a with statement:

```python
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:

```python
with client.get("/does_not_exist/", catch_response=True) as response:
    if response.status_code == 404:
        response.success()
```
CHAPTER 5

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.

5.1 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:

```python
import time
import xmlrpclib
from locust import Locust, events, task, TaskSet

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
    get 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
            # reported correctly in the statistics, we would probably need to hook in at a lower

        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.

```python

    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 it’s 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:

```python

    import time
    import random
    from SimpleXMLRPCServer import SimpleXMLRPCServer
    import xmlrpc

    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()
```
6.1 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

6.2 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.

6.3 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.

TaskTests can be nested, which means that a TaskSet’s `tasks` attribute can contain another TaskSet. If the nested TaskSet is scheduled to be executed, it will be instantiated and called from the current executing TaskSet. Execution in the 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,list` list of two-tuples, or a `dict` of `callable: int`, 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}
```
6.4 task decorator

```python
@task(100)
def read_thread(self):
    pass
@task(7)
def create_thread(self):
    pass
```

6.5 HttpSession class

```python
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-requests'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).

```python
define
/delete(url, **kwargs)
define
get(url, **kwargs)
```

Sends a DELETE request. Returns `Response` object.

Parameters

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

Sends a GET request. Returns `Response` object.

Parameters

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

Sends a HEAD request. Returns `Response` object.

**Parameters**

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

**options** *(url, **kwargs)*

Sends a OPTIONS request. Returns `Response` object.

**Parameters**

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

**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.

**post** *(url, data=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`.
- **kwargs** – Optional arguments that `request` takes.

**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.

**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...
Locust, Release 0.7.0

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** – (optional) Float describing the timeout of the request.
- **allow_redirects** – (optional) Boolean. Set to True by default.
- **proxies** – (optional) Dictionary mapping protocol to the URL of the proxy.
- **return_response** – (optional) If False, an un-sent Request object will returned.
- **config** – (optional) A configuration dictionary. See request.defaults for allowed keys and their default values.
- **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.

### 6.6 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.

**Response.apparent_encoding**

The apparent encoding, provided by the lovely Charade library (Thanks, Ian!).

**Response.close()**

Closes the underlying file descriptor and releases the connection back to the pool.

*Note: Should not normally need to be called explicitly.*

**Response.content**

Content of the response, in bytes.

**Response.cookies = None**

A CookieJar of Cookies the server sent back.

**Response.elapsed = None**

The amount of time elapsed between sending the request and the arrival of the response (as a timedelta)

**Response.encoding = None**

Encoding to decode with when accessing r.text.
Response.headers = None
Case-insensitive Dictionary of Response Headers. For example, headers['content-encoding']
will return the value of a 'Content-Encoding' response header.

Response.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.

Response.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.

Response.iter_lines (chunk_size=512, decode_unicode=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.

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

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

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

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

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

Response.status_code = None
Integer Code of responded HTTP Status.

Response.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 soley 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.

Response.url = None
Final URL location of Response.

6.7 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:

```python
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()
```

### 6.8 InterruptTaskSet Exception

**exception InterruptTaskSet** (reschedule=True)

Exception that will interrupt a Locust when thrown inside a task

### 6.9 Event hooks

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

```python
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: $a, $b" % (a, b)
    my_event += on_my_event
    my_event.fire(a="foo", b="bar")
```

#### 6.9.1 Available hooks

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

- **request_success** = `<locust.events.EventHook object at 0x2552710>`
  `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 at 0x2552210>`
  `request_failure` is fired when a request fails

  Event is fired with the following arguments:
Locust, Release 0.7.0

• **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 at 0x25521d0>`

`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 at 0x25524d0>`

`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 at 0x2552510>`

`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 at 0x2552550>`

`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 at 0x2552590>`

`quitting` is fired when the locust process in exiting

24 Chapter 6. API
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:

```python
from locust import events

def my_success_handler(method, path, response_time, response, **kw):
    print "Successfully fetched: $s" % (path)

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.

### 7.1 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:

```python
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
8.1 0.7

8.1.1 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 functionality 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.

8.1.2 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.

8.1.3 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

8.1.4 gevent updated to version 1.0

gevent 1.0 has now been released and Locust has been updated accordingly.
8.1.5 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)

8.1.6 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.

8.1.7 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.

8.1.8 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:

```python
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.

8.1.9 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

8.2 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.

8.3 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.

8.4 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.

8.4.1 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:

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

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


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
    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.

8.4.2 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
Locust, Release 0.7.0

requests. However, it also comes with some small API changes which 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 wants to disable the Accept-Encoding that the HTTP client uses, or any other HTTP headers you can do:

```python
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:

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

  And should now be specified like:

  ```python
  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.

```python
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`.

    `responseContextManager.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:
```

8.4. 0.6 31
with self.client.get("/", catch_response=True) as response:
    if response.content == "":
        response.failure("No data")

ResponseContextManager.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:

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

Can instead now be written like:

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

---

### 8.4.3 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`)

### 8.4.4 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 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.

---

### 8.5 0.5.1

- Fixed bug which caused –logfile and –loglevel command line parameters to not be respected when running locust without zeromq.
8.6 0.5

8.6.1 API changes

- Web interface 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 get the HTTP method as the first argument.

8.6.2 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!

8.7 0.4

8.7.1 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`.

8.7.2 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.
locust.events, 23