UPPSALA UNIVERSITET

IT 13 090

Examensarbete 15 hp
December 2013

Implementing dynamic allocation of user load in a distributed load testing framework

Hugo Heyman

Institutionen för informationsteknologi
Department of Information Technology

Abstract

Implementing dynamic allocation of user load in a distributed load testing framework

Hugo Heyman

Load testing has become an increasingly important matter in today's web applications. Upon releases of popular applications its users can create massive load potentially making an application unreachable. The scope of this thesis was to investigate as well as further develop an open source load testing framework: Locust. Locust aims to help creating as authentic simulated user behavior as possible. This allows an application's user-load threshold to be measured by the number of users and not just "requests per second" as most other load testing tools do. A challenge in implementing allocation of user-load during runtime of a test was to make code fit to the distributed architecture that Locust supports. By having a distributed architecture, a Locust test can be scaled up to simulate millions of simultaneous users. Two libraries essential to Locust: Greenlet and Gevent are studied in this thesis. These libraries make in-process concurrency possible by providing an API for running code in coroutines also known as "micro-threads". Another ambition during this thesis was to implement a feature allowing an automatic algorithm to find the application threshold by ramping up the number of simulated users. The implementation and results are presented in this thesis.

Handledare: Ronnie Kolehmainen
Ämnesgranskare: Justin Pearson
Examinator: Olle Gällmo
IT 13 090
Tryckt av: Reprocentralen ITC

Contents

1 Introduction

Locust[1] is a scalable, distributed user load testing framework for web services [1]. Its purpose is to help figure out the maximum number of concurrent users a system can handle.

Locust is a MIT-licensed open source project created and maintained by individual developers and the company ESN in Uppsala.

Locust test scenarios as well as Locust itself are written in Python. Writing test cases in python allows the tests to be diverged, complex and specific for a certain web service. A test cycle for a simulated user could include logging in, roaming the service for a while and then logging out.

As stated in the documentation of the framework, Locust is very “hackable” [2]. How Locust works will be discussed in more detail in the background section, but to summarize it simulates users that send HTTP requests to the server and then collects data and statistics on these requests.

To be capable of testing large scale services that run on multiple servers Locust itself needs to be scalable to generate enough workload on the servers. The distribution and scaling are dealt with by having a master/slave architecture.

During the period of this project Locust has been used in production by ESN. A project that ESN has been involved in is Battlelog [14], which is the social web platform for the PC game Battlefield 3 [15]. To load test the Battlelog service Locust has been used in order to simulate millions of users.

[1] Locust - http://locust.io/

2 Background

Locust was initially started at ESN when the need to performance test Battlelog arose. Existing performance testing tools such as JMeter [12] and Tsung [13] did not fully satisfy the needs of this project. The drawback of JMeter was that it was thread-bound, making it hard to simulate thousands of users, when every user needs its own thread. Tsung written in Erlang and using lightweight processes did not have this problem, however writing test scenarios and user behavior with XML could limit the test. Locust was written with the philosophy that it is the concurrent user count that is of most interest for the tester, not the number of requests per seconds (RPS) or any other traffic specific data that follows. If the simulated users can mimic real users in a realistic way, then the RPS is only a consequence of the users. [2]

2.1 Architecture overview

To fully understand the problems explained in this thesis, a rough overview of how Locust operates will be given. To do this in a less confusing way, smaller parts and modules of the framework will be described one at a time. To get a more direct understanding also see the Locust source code [1].

2.1.1 Starting and loading Locust

The execution flow starts in the main function of main.py. The first thing that happens is that console parameter options that locust was started with are parsed.

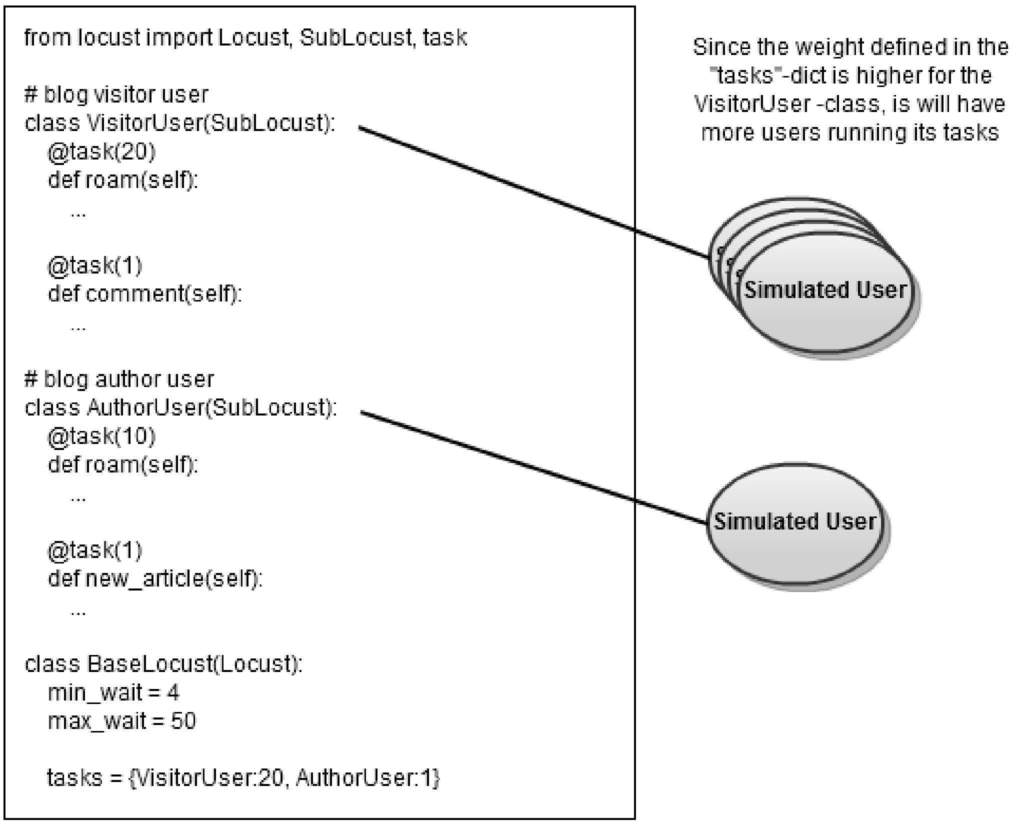
The locustfile that defines the test is loaded and Locust classes with associated "tasks" are parsed and stored. Tasks are simply decorated functions that perform HTTP requests such as GET and POST.

Each task can have an associated weight. The weight of a task determines how often the task will be executed compared to the rest of the tasks (of the same class).

The locust classes can be sub classed, in order to build a tree structure of tasks. Each locust class can have the associated variables min_wait and max_wait assigned.

These variables define an interval that simulated users will pick a random time from to wait before executing the next task.

Since the locustfile is just simply python, the freedom in defining a test in specific ways offers great potential.



```
from locust import Locust, SubLocust, task

# blog visitor user
class VisitorUser(SubLocust):
   @task(20)
   def roam(self):
      ...

   @task(1)
   def comment(self):
      ...

# blog author user
class AuthorUser(SubLocust):
   @task(10)
   def roam(self):
      ...

   @task(1)
   def new_article(self):
      ...

class BaseLocust(Locust):
   min_wait = 4
   max_wait = 50

   tasks = {VisitorUser:20, AuthorUser:1}
```

Since the weight defined in the "tasks"-dict is higher for the VisitorUser -class, is will have more users running its tasks
Simulated User
Simulated User

This is an example of a simple locust test for a made up blog web service (More details on how to write a locust test can be found in the Locust documentation [2])

2.1.2 How the Locust browser client works:

To do the actual HTTP requests the python standard library urllib2 is used.

The “browser” that is used by the simulated users is defined in clients.py. When a request is executed the time until a response has been received is recorded.

Locust is implemented using events. When a request is finished, an event is fired to allow the stats module to store a response time or possibly a failure.

2.1.3 The architecture of managing the simulated users:

Each simulated users run in Greenlet threads. A greenlet is actually not a real thread but it makes sense to think of it as such. Read further about the Greenlet and Gevent library in section 2.2.

The greenlets are scheduled (spawned and killed) by a runner in the runner locust module. The runner itself is a greenlet spawned from the main module. The runner can order its simulated users to start (spawning greenlets) or to stop (killing greenlets). When a simulated user has been started its local greenlet handles the actual requests and waiting.

2.1.4 The architecture of running locust distributed

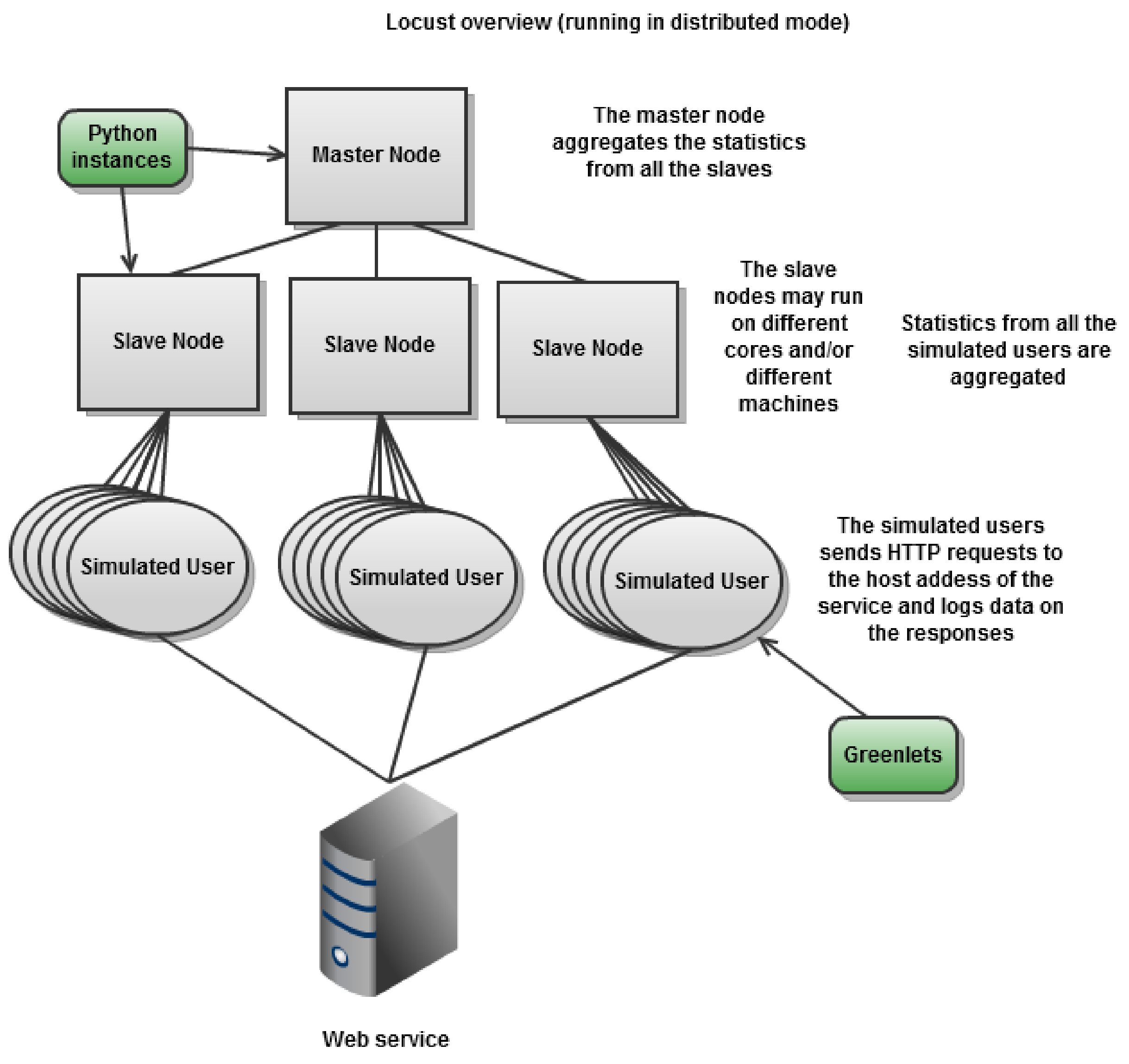
To be able to make use of more than one core running python, because of the python GIL, separate python instances have to be run [7]. So to utilize the full potential of a CPU with 4 cores, 4 python instances running locust would be used.

To communicate between any locust instances the socket API ZeroMQ [6] is used. This allows instances to cooperate both locally on the same machine and/or remotely.

When running Locust in distributed mode, it uses master/slave architecture. This is achieved by having the runner class managing the simulated users at each Locust instance be either a `MasterLocustRunner` or `SlaveLocustRunner`.

The master class handles no simulated users of its own but instead manages slave Locust instances.

A master instance can send messages to its slaves ordering them to start or stop tests. It also listens to messages from its slaves. These messages can be statistics that the master can aggregate to get the total results but also other messages such as “I got an exception” or “I’m shutting down”.



Locust overview (running in distributed mode)
Python instances
Master Node
The master node aggregates the statistics from all the slaves
Slave Node
Slave Node
Slave Node
The slave nodes may run on different cores and/or different machines
Statistics from all the simulated users are aggregated
Simulated User
Simulated User
Simulated User
The simulated users sends HTTP requests to the host addess of the service and logs data on the responses
Greenlets
Web service

2.1.5 The user interface

The user interface for Locust is an interactive web interface hosted with the micro python web framework Flask [10].

Statistics for each URL requested by the simulated locust users are rendered in a table constantly refreshed by JavaScript code that does HTTP request to the Locust web server.

A couple of variables and interesting data are displayed in a panel on the interface, namely: the Locust status, the current RPS (requests per seconds), the failure rate, and also if you are running in distributed mode, the number of slaves connected.

The user can do things like starting a new test, stopping a running test, or download percentile stats for all requests' response times.

The status of Locust is set at the HTML body tag's class attribute; this way the interface can change upon events in Locust.

2.2 Gevent, greenlets and monkey patching

Since these libraries and techniques are extensively used in Locust some deeper research on how they work was made. Greenlets are sometimes referred to as lightweight pseudo-threads; this is because they may act like threads but in reality they run on the same OS thread and are scheduled cooperatively [5]. Gevent is a Python networking library that uses greenlets to provide a synchronous API on top of libevent event loop [4]. The piece of code below shows how asynchronous non-blocking code can be implemented using gevent and greenlets:

```
import gevent
import random

def task(id):
    gevent.sleep(random.randint(0, 5))
    print "task %d done" % (id)
```

The function `task` will sleep 0 to 5 seconds and then print a message that it's done

```
for i in xrange(10):
    task(i)
```

These calls will have a total wait time average of 25 seconds and result in the output:

```
task 0 done
task 1 done
task 2 done
…
```

and so forth

If instead the tasks are run in separate greenlets they will execute asynchronous:

```
l = []
for i in xrange(10):
    l.append(gevent.spawn(task, i))
gevent.joinall(l)
```

The output lines may come in any order and the total wait time will be max 5 seconds.

Monkey patching is a technique for patching standard library functions at run time to be compatible with gevent and greenlets. For example using `time.sleep` instead of `gevent.sleep` as done in the code above, but patching the `time` library gives the same result:

```
import gevent
import time
from gevent import monkey
import random

monkey.patch_time()

def task(id):
    time.sleep(random.randint(0, 5))
    print "task %d done" % (id)
```

2.3 Tutorial - How to get started with Locust

Here is a short tutorial on how to set up Locust, all its prerequisite packages and how to run a basic test run towards locust own web server (the user interface).

On a Mac or Linux the setup should in most cases be easy:

Use a PyPI script (Python Package Index) to fetch and install Locust like so:

$ sudo pip install locustio

or

$ sudo easy_install locustio

(make sure your ../python/script directory is in your path)

On a Windows machine, you have to install the pre-compiled binaries for the packages Gevent and Greenlet. When this is done you can use the script “pip install” or “easy_install” just like previously shown on a Mac/Linux machine. Although you will need to install the Distribute package: http://python-distribute.org/distribute_setup.py to make use of easy_install and pip.

Now locust is ready to use, type:

$ locust --help

to see all available options

Below is code for a very basic locust-file (a python-file) to load test towards locust's own web server. It just sends GET requests towards the index page and a URL for fetching updates of the statistics.

```
from locust import Locust
def index(l):
    l.client.get("/")

def stats(l):
    l.client.get("/stats/requests")

class WebsiteUser(Locust):
    tasks = [(index,1),(stats,2)]
    min_wait=2000
    max_wait=5000
```

Now start the Locust service with our locustfile like this:

$ locust -f <locustfile> --web --host http://127.0.0.1:8089

Now visit http://127.0.0.1:8089 in a browser on the same machine to see the web interface; set the number of users to simulate and the hatch rate; click swarm and see the statistics generated in real time.

3 Problem Description

These are the target implementation challenges I will examine, discuss and also try to implement.

3.1 The possibility to change the number of simulated users during run time

The first problem, which is a sub problem of the second one, is to implement a feature allowing the tester to change the number of simulated users during an ongoing test. As done prior to this project, the number of users to be spawned would be set at the start of the simulation. In order to change this number at run time one would have to restart the whole simulation meaning that all users would have to be spawned up again. Considering that the time it takes to spawn large amounts of users is quite significant shows the need of this feature. A challenge implementing this feature will be refactoring some parts of the present code.

3.2 Implementing auto-adjustable allocation of locust users

The second problem is to implement auto-adjustable allocation of locusts. This is a mechanism that in an automated way can find the maximum number of concurrent users that a web service can handle without getting request failures or response times higher than a set limit. An algorithm to find a stable state below a defined limit in a fast way should be researched and implemented.

4 Implementations

The Locust code base and associated libraries make up the major reference material during the thesis work. The methods used include studying code, identifying possible pitfalls and finally provide code implementations for the target problems of this thesis.

4.1 Getting to know the code

To get a good overlook of how Locust works, the execution path was followed and `main.py` was the inevitable starting point. Quite early on it was apparent that a number of new libraries were extensively used. An interruption in the code reading was made to research on how greenlets [5], gevent [4] and monkey patching were used in order to write non-blocking I/O asynchronous code. After getting a brief overlook and understanding of Locust, Jonatan Heyman and Ronnie Kolehmainen at ESN had a walk-through explaining the different parts and architecture of Locust, as well as showing how to write a simple Locust file and perform tests with it. Most of the Locust code was documented and commented but some complex parts such as the `stats.py` module lacked in documentation, which made it quite hard to follow.

4.2 Implementing dynamic allocation of users

After getting to know the code, the first problem was to figure out how to extend Locust making it capable of spawning additional users during test runs. To start with, focus was set on how the spawning of users was done when running on a single node, and how this could be extended. The `start_hatching` function of the `LocustRunner` class in `core.py` was where the user greenlets [5] were started, and this was the first function that had to be adapted to the future changes. The class variables `num_clients` and `hatch_rate` of `LocustRunner` were set when a test was started and were used inside `start_hatching` to determine the number of users to spawn and at what rate/second to spawn them. The first change was to add arguments to the `start_hatching` function that would replace the use of the class variables that represent the hatch rate and the total amount of users; this so that if 10 users are first spawned but then 15 are desired, 15 would be the `locust_count` argument and `start_hatching` would subtract the current amount of users (`num_clients`) from the desired ones and then spawn that many (5 in this case).

To do the actual spawning `start_hatching` called `spawn_locusts`. But before the local function `hatch` spawned the greenlets it first did some weighting of the `SubLocust` –classes found in the Locust file. Since it was foreseen that the same routine would be needed when killing off greenlets a decision was made to refactor `spawn_locusts` and make the weighting routine a separate function.

4.2.1 core.py LocustRunner.weight_locusts()

```
def weight_locusts(self, amount, stop_timeout = None):
    """
    Distributes the amount of locusts for each WebLocust-class
    according to its weight
    returns a list "bucket" with the weighted locusts
    """
    bucket = []
    weight_sum = sum((locust.weight for locust in
        self.locust_classes if locust.tasks))
    for locust in self.locust_classes:
        if not locust.tasks:
            warnings.warn("Notice: Found locust (%s) got no tasks.
              Skipping..." % locust.__name__)
            continue

        if self.host is not None:
            locust.host = self.host
        if stop_timeout is not None:
            locust.stop_timeout = stop_timeout

        # create locusts depending on weight
        percent = locust.weight / float(weight_sum)
        num_locusts = int(round(amount * percent))
        bucket.extend([locust for x in xrange(0, num_locusts)])
    return bucket
```

The next step was to implement reduction of users. To do this, the function `kill_locusts` was implemented which is also called from `start_hatching`. A check was added in `start_hatching` to see whether the `locust_count` argument was higher or lower than the current locust count (`num_clients`). A problem that arose when writing this function was how to identify specific `Locust` class objects contained by greenlets in the `self.locusts` `Group` object to be able to kill them off. The problem was solved by adding the `Locust` class object in the local function `hatch` as an argument when spawning the greenlet for each `Locust` class:

```
self.locusts.spawn(start_locust, locust)
```
instead of just

```
self.locusts.spawn(start_locust)
```
This made it possible to identify the class object for each greenlet in the group so that the corresponding greenlet could be killed.

4.2.2 core.py LocustRunner.kill_locusts()

```
def kill_locusts(self, kill_count):
    """
    Kill kill_count of weighted locusts from the Group() object in
    self.locusts
    """
    bucket = self.weight_locusts(kill_count)
    kill_count = len(bucket)
    self.num_clients -= kill_count
    print "killing locusts:", kill_count
    dying = []
    for g in self.locusts:
        for l in bucket:
            if l == g.args[0]:
                dying.append(g)
                bucket.remove(l)
                break
    for g in dying:
        self.locusts.killone(g)
    events.hatch_complete.fire(self.num_clients)
```

`kill_locusts` does the following:

- Calculates a weighted list of locusts to kill
- Subtracts its argument `kill_count` from `num_clients`
- Goes through every weighted `Locust` class object and puts greenlets from the `Group` object `self.locusts` to be killed in a list called `dying`
- The list is then iterated over and the greenlets [5] in the group are simply killed with `self.locusts.killone(g)`

Extending the changes made so far to also work when running in distributed mode did not require a lot of work since the `SlaveLocustRunner` class inherits the functions for spawning/killing users. Although here are some things that were necessary to adapt:

- `hatch_rate` and `num_clients` had to be added to the message that the master sends to its slaves when requesting spawning/killing of users. On the slave side `hatch_rate` and `num_clients` had to be added as arguments to the `start_hatching` request.
- When distributing the spawning/killing of users evenly among a master's slaves in `start_hatching` of the `MasterLocustRunner` the class variables are used just as they were in the `LocustRunner` class. This was resolved in the same way as it was done in `LocustRunner`; by adding the arguments `start_hatching` and `locust_count` to the `start_hatch` function.

4.3 User interface for dynamic allocation

When the back-end features had been implemented the next step was to get the new features accessible from the user interface.

To make it easier and to keep the theme of the interface, the HTML div and form for starting a new test was simply copied and one for editing the current number of users was made. To make the button and form for editing users appear at the right times, the CSS of the form had to be changed so that the “display” property for the different states that was rendered in the body tag would hide or show the form. A problem encountered was that when submitting the form, the class of the body tag would not change and things would get erroneously rendered. This was due to the fact that submitting the form would not reload the whole page since the submit event was captured by JavaScript, using jQuery [8], and the form was submitted using an AJAX request while the normal form submit handling was prevented. This was solved using JavaScript to manually change the class of the body tag to “hatching” when submitting.

4.4 Implementing auto-adjustable allocation of locust users

The main idea behind the “auto adjustable allocation” -feature was that by analyzing the data from the responses of all users, a stable maximum number of users could be found. The routine for managing this would be performed in an iterative fashion where the number of users are increased or decreased until some thresholds of the data are met.

The feature was named “ramping”, since the intention was that the number of users would ramp up and down until a high stable point was found. A considerable amount of time was spent figuring out how this feature would be most useful, but it's really hard to conclude anything without testing it. A couple of problematic scenarios that could arise were foreseen: What if a web server under heavy load would simply crash or reach a broken state that it is unable to recover from? How does the curve: percentile response time versus number of users look like? Is the response time linearly bound to the number of users or is it not affected at all until all requests fail? All these scenarios could potentially differ depending on the service, the server software and the hardware that is tested.

After considering this it seemed necessary for the ramping feature to be diverse and not too static. The intention was to use response failure ratio and response time percentile values as main factors controlling the ramping. Response time percentile being a response time that a certain percent of all requests performed is below.

4.5 Ramping statistics

Before working on the actual ramping algorithm, statistics gathering to be used by the ramping function would have to be implemented, namely the response failure ratio and the momentary response time percentile.

- **failure ratio**

  The percentage of HTTP requests giving responses with status codes in the range of 400-599 (where 4xx codes are client errors and 5xx server errors)

- **momentary response time percentile**

  The response time value at which a certain percent of all response time values from successful HTTP requests are bellow.

Implementing the failure ratio proved to be a simple task since the number of successful requests and requests that had failed were data already gathered in `stats.py`. The function (`fail_ratio`) was implemented as a class property of `RequestStats` in `stats.py`.

4.5.1 `stats.py RequestStats.failratio`

```
@property
def fail_ratio(self):
    try:
        return float(self.num_failures) / (self.num_reqs +
self.num_failures)
    except ZeroDivisionError:
        if self.num_failures > 0:
            return 1.0
        else:
            return 0.0
```

Getting the momentary percentile stats would prove to be harder and more time consuming. `stats.py` was lacking in documentation and it was hard to see the architecture and execution path.

In a first attempt to implement a momentary response time percentile, a mix up of what was happening in the master node and what the slave nodes did was made. After some consultancy with Jonatan Heyman at ESN this could be identified and it was suggested that the stats gathering for the momentary response time percentile was put in a separate module and also that the ramp feature would be an optional feature in Locust to begin with, this so that any overhead added by the additional stats gathering would not affect the Locust tests ESN was currently running. The new module was named `rampstats.py` and can be found in appendix A of this paper.

This is what `rampstats.py` does when running in local mode:

- To store the response time for each request a queue data type is used (`collections.deque`), this so that old response times can be popped from one end and new response times can be appended to the other.
- A listener is listening to the event `request_success` and when fired the response time for the request is appended to the queue. Also, if the queue is longer than the current RPS (requests per second) * a time window given in seconds, the queue will pop response times until the length of the list goes below.
- To get the momentary percentile the function `current_percentile` can be called. It calls the function percentile from `stats.py` on the sorted queue of response times.

When `rampstats.py` is run in distributed mode there are a few additions and things done differently:

- The slaves append the response times to a list when the event `request_success` fires
- An event listener; `report_to_master`, is triggered in the slave when a slave is about to report its stats to the master, and the list that the slave is holding is added to the data that is sent to the master. Then the list is emptied.
- The master is listening to only one event in `rampstats.py` and that is `slave_report`. The master receives the lists of response times sent from its slaves, but how can the master know in what order the response times were actually measured? This is dealt with by not considering the order of the response times within the list. Instead the same `deque` object that in local mode contained response times now contains lists of response times. Also lists are now appended or popped instead of single response times. In order to calculate the response time percentile for a certain time back in history, the number of slaves times how far back in seconds gives length of the queue that represent this time window. This is almost true except for that the slaves as of now only report every third second so the result should be divided by 3.
- Having lists of response times in the queue should give much less CPU overhead than having a flat queue of the response times. The overhead could be huge if the test of a service had tenth or maybe hundreds of thousands of requests per second. Instead the overhead with this solution should be O(n) for the amount of slaves (appending and popping a `deque` has the time complexity ϴ(1)) .

- When the `current_percentile` function from `rampstats.py` is called while running in distributed mode the queue of lists containing response times is flattened and then the same way as it is done in local mode, the percentile can be calculated and returned.

4.6 The ramping feature

After the new statistics implementations were done and tested it was time to start implementing the ramping feature. To a start the algorithm would only spawn up users until a first boundary was met and then stop the ramping. Many different parameters were tested and used to be able to configure how the ramping would react. These are the parameters that are used in the final implementation:

- `start_count`

  This is the initial amount of users that will be spawned all in one step prior to any ramping.

- `hatch_rate`

  The speed users/second at which the users are to be spawned.

- `hatch_stride`

  This many users are spawned at each time before calibrating and checking if a boundary is reached.

- `precision`

  As the algorithm gets closer to “the sweet spot” `hatch_stride` will decrease and when `hatch_stride` is lower than the `precision` parameter and the boundaries are not exceeded the sweet spot has been found and the ramping can stop.

- `calibration_time`

  This is the waiting time in between spawning or killing `hatch_stride` amount of users. This needs to be long enough to let the amount of requests stabilize and to get enough statistics so accurate measurements can be made.

- `max_locusts`

  This is the boundary for the maximum number of users spawned.

- `percent`

  This is the percent value of the response time percentile, meaning that if percent is set to 95 the response time limit will represent 95% of the requests.

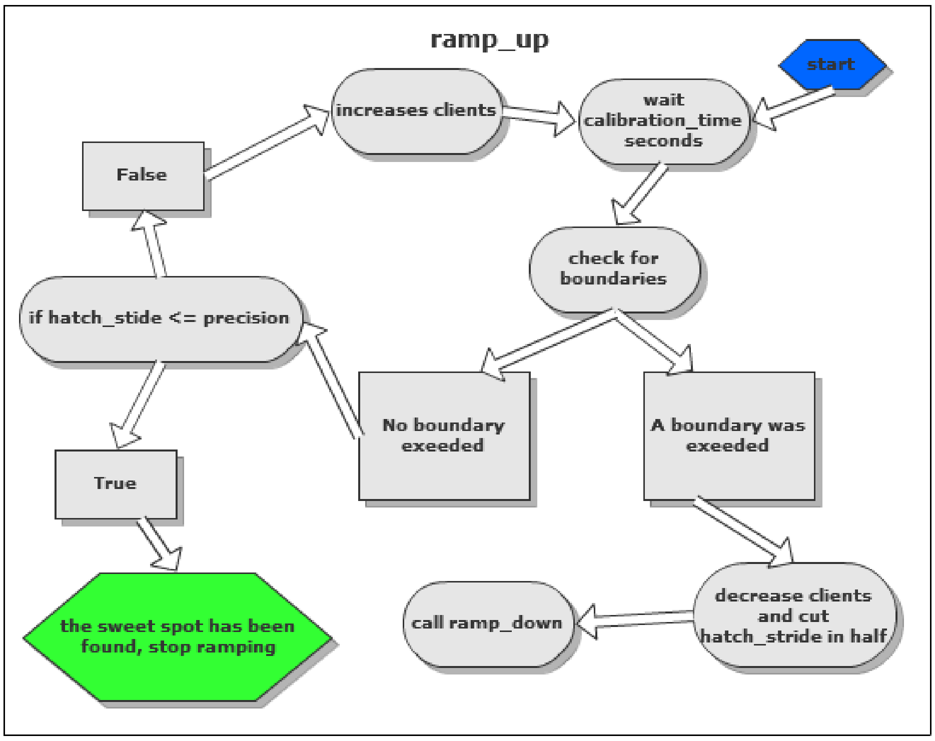
- `response_time_limit`

  This sets the boundary for how high the response time percentile can be in milliseconds.

- `acceptable_fail`

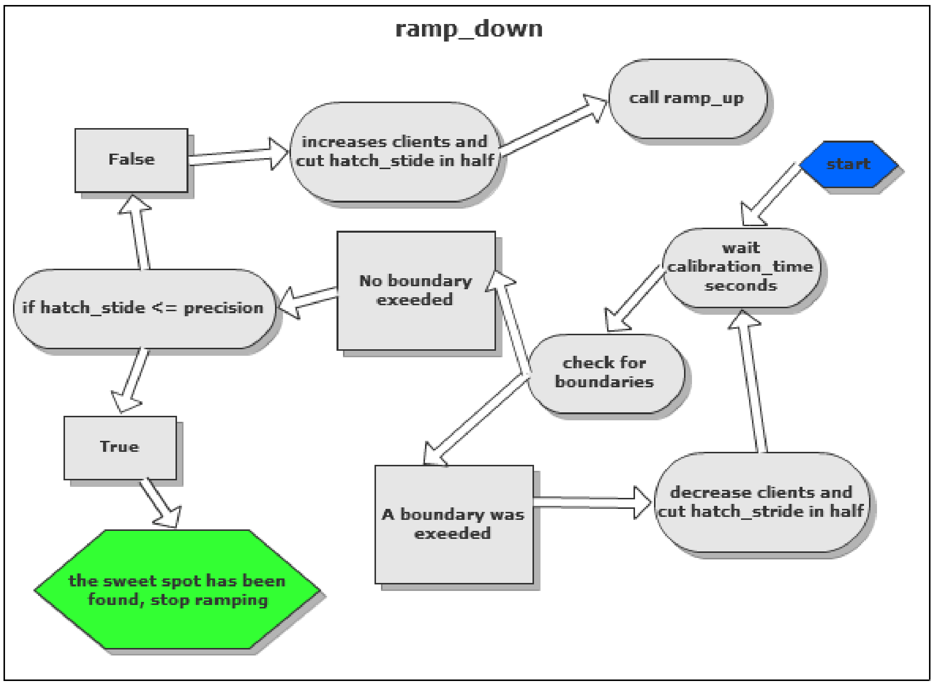
  This sets the boundary for how many of the requests are allowed to fail given in percent.

To start ramping the `start_ramping` function is called with all parameter arguments. `start_ramping` has two local functions: `ramp_up` and `ramp_down`; `ramp_up` is called within `start_ramping` but prior to this the number of initial users (`start_count`) are spawned. In between `start_hatching` -calls `ramp_up` and `ramp_down` spins in a while loop waiting for the Locust status to change from "HATCHING" to "RUNNING". To not block the IO a sleep of one second is performed each time. Here is a flow chart describing the `ramp_up` function:



ramp_up
start
wait calibration_time seconds
check for boundaries
No boundary exeeded
A boundary was exeeded
decrease clients and cut hatch_stride in half
call ramp_down
if hatch_stide <= precision
False
increases clients
True
the sweet spot has been found, stop ramping

This is a flowchart describing `ramp_down`:


ramp_down
call ramp_up
False
increases clients and cut hatch_stide in half
start
wait calibration_time seconds
No boundary exeeded
if hatch_stide <= precision
check for boundaries
True
A boundary was exeeded
decrease clients and cut hatch_stride in half
the sweet spot has been found, stop ramping

At first `hatch_stride` would grow in size every time it ramped up so that finding the boundary would go faster. After tests and some thinking it was evident that this really could be counterproductive for several reasons: With the initial (`start_count`) users one should be able to do a fair estimate of how much a server can handle making `hatch_stride` growing less useful. Not knowing how the servers react to excess workload makes it important the resources are exceeded with as little as possible. The `start_ramping` function can be seen in Appendix B of this paper.

4.7 User interface for the ramping feature

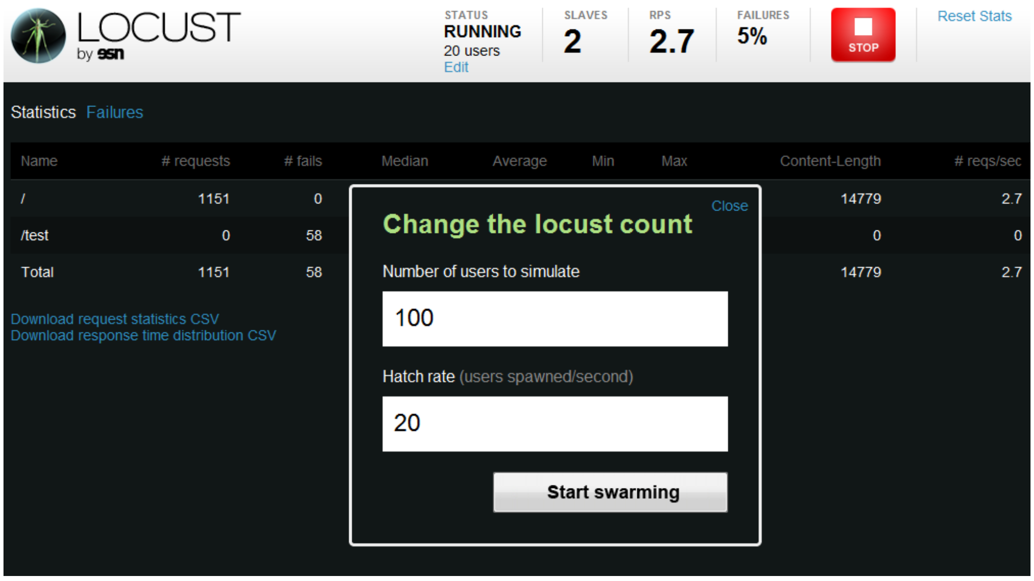
The user interface could be implemented much similar to how the form for editing user amount during run time was done. However the function handling the URL request had to be implemented. A problem that came up was that when submitting the form for ramping, the interface would “freeze” (stop updating) when submitting it. This was because the `start_ramping` function would take too long and not letting the function for the ramp URL return its JSON response. The solution was to let the `start_ramping` function run in its own greenlet. Since the whole ramping feature was intended to be optional, so was the feature in the interface. Therefore a variable was sent with the response of the index URL ('/') to tell whether it was running with the ramping feature or not.

5 Results

The results of this thesis are ultimately the code contributions to Locusts code base. All code quotations in this report except for the tutorials in the background section are contributions to Locusts code base.

5.1 Allocating users during ongoing tests

With the refactoring and changes in `core.py`, Locust now has the ability to increase or decrease the number of simulated users during ongoing tests. This feature should be of great use to anyone using Locust. Not having to restart an entire test each time a new number of users are to be tested should save people a lot of time. Killing off users is a lightweight operation, hence it can be done almost instantly and does not need any wait time in between. Spawning additional users on the other hand cannot, since it's more computationally heavy to spawn the greenlets and therefore the spawning of users is bound to a hatch rate. Changing the number of users in the user interface is done similarly to how a new test is started. A button "edit" below the current user count opens the form for changing the number of users.



STATUS RUNNING 20 users Edit | SLAVES 2 | RPS 2.7 | FAILURES 5%

Name	# requests	# fails	Median	Average	Min	Max	Content-Length	# reqs/sec
/	1151	0					14779	2.7
/test	0	58					0	0
Total	1151	58					14779	2.7

Change the locust count

Number of users to simulate: 100

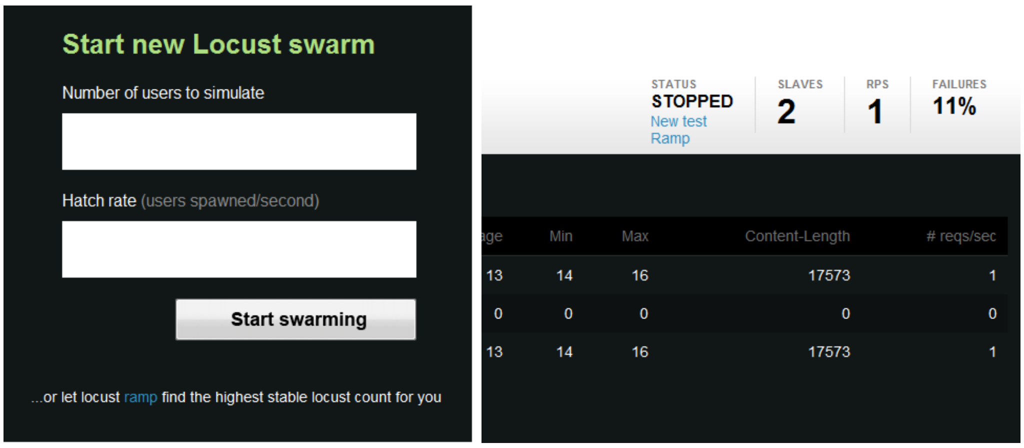
Hatch rate (users spawned/second): 20

LOCUST
by esn
STOP
Reset Stats
Statistics
Failures
Download request statistics CSV
Download response time distribution CSV
Close
Start swarming

Changing the number of locusts (users) during a test run

5.2 The ramping feature

The ramping feature adds a helping tool to Locust when measuring the capacity of a system. There are no rules for how this feature should be used; one might want to know when the response time for a certain percentile is reached, or when requests start to fail. Ramping on a simple blog running on a desktop computer or a big distributed service should work just as well since the ramping is configurable.


Start new Locust swarm

Number of users to simulate

Hatch rate (users spawned/second)

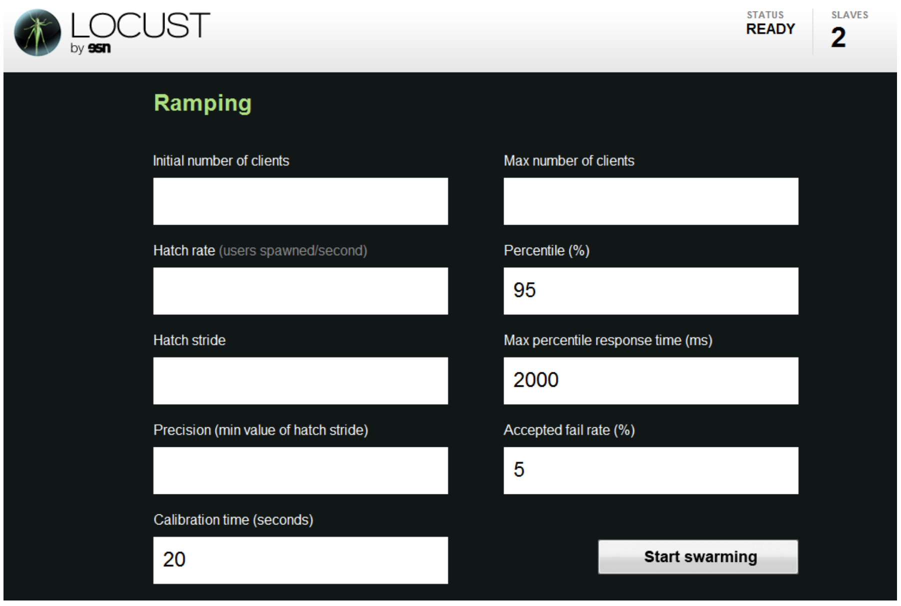
Start swarming

...or let locust ramp find the highest stable locust count for you

STATUS STOPPED New test Ramp | SLAVES 2 | RPS 1 | FAILURES 11%

age	Min	Max	Content-Length	# reqs/sec
13	14	16	17573	1
0	0	0	0	0
13	14	16	17573	1

When ramping is enabled, the ramping form can be accessed either at the start page of Locust or in the interface when Locust is in state stopped



STATUS READY | SLAVES 2

Ramping

Initial number of clients:	Max number of clients:
Hatch rate (users spawned/second):	Percentile (%): 95
Hatch stride:	Max percentile response time (ms): 2000
Precision (min value of hatch stride):	Accepted fail rate (%): 5
Calibration time (seconds): 20	Start swarming

This is the form for ramping in the user interface.

6 Discussion

While Locust can be a very useful tool while testing the setup of a service, no details on what bottlenecks that exist are given. Locust will not tell whether a bottleneck has its origin in a database driver, the CPU capacity, or poorly written code. The only measurement Locust focus on is how many end users a service can manage; given that the test written is representative for the end users.

6.1 Necessity of the ramp feature

One could argue that if time has been spent writing a proper test for a service; the time it takes to manually find the user limit is minute in comparison. This may be true when only testing once or twice; but in many real productions the necessity of frequent testing is often considered critical. These are quotes from Molyneaux: “*Performance awareness should be built into the application life cycle as early as possible*”, “*You really can't carry out effective performance testing without using automated test tools*” [3].

6.2 Future development

Concurrent to this project Locust was optimized in how it gathers statistics; response times for all requests gathered during a test are rounded off to two figure numbers and saved in a dict instead of having a list that could grow immensely large. Having this dict helps when calculating the median response time, which is something done frequently.

6.2.1 Tree architecture distribution

If it would show that the ramping feature or any other task adds significant overhead when gathering stats in the master, a solution could be to extend Locust further to have a `"NodeLocustRunner"` class. That is a `LocustRunner` acting as both master and slave; it would receive messages and data just like the `MasterLocustRunner` but forward any message giving the network a tree structure. This extension should not require too much work since most functions could be inherited from the existing classes.

6.2.2 CPU and server utilization monitoring

To make sure that slave locusts are able to create enough workload, it could be useful to see the CPU load for the slave machines. To implement this the library `psutil` [9] could be used. It could also be interesting to see the CPU load of the actual web server or other server components within the architecture of a service. A Locust instance with the only purpose to send reports on the CPU load might be a handy addition to Locust. This addition could be utilized to further extend the ramp feature, giving additional data to consider when configuring a ramp test.

7 References

[1] Jonatan Heyman, Carl Byström, Joakim Hamrén, *User load testing tool for web services*, ESN Social Software, 2011-11-05, Available at: http://locust.io and version 0.4 of Locust
https://github.com/locustio/locust

[2] Jonatan Heyman, Carl Byström, *Locust documentation*, 2011-11-05, http://docs.locust.io

[3] Ian Molyneaux, *The art of application performance testing*, O'Reilly Media, Inc, 2009

[4] Denis Bilenko, *Co-routine network library for Python*, 2011-11-05, http://www.gevent.org/

[5] Kyle Ambroff, *Lightweight in-process micro-threads library for Python*, 2011-11-05, http://pypi.python.org/pypi/greenlet

[6] iMatix Corporation, *High performance asynchronous messaging library with socket style API*, 2011-11-05,
http://www.zeromq.org/

[7] Python Software Foundation, *Python global interpreter lock*, 2011-11-05, http://docs.python.org/glossary.html#term-gil

[8] John Resig, *Cross-browser JavaScript library*, 2011-11-05, http://www.jquery.com/

[9] Giampaolo Rodola, *Monitoring of system utilization library*, 2011-11-05, http://code.google.com/p/psutil/

[10] Armin Ronacher, *A Python micro web framwork*, 2011-11-05, http://flask.poco.org

[11] Python Software Foundation, *Python Documentation*, 2011-11-05, http://docs.python.org

[12] Apache Software Foundation, *JMeter*, 2011-11-05, http://jmeter.apache.org/

[13] Nicolas Niclausse, *Tsung*, 2011-11-05, http://tsung.erlang-projects.org

[14] Electronic Arts / ESN, *Battlelog*, 2011-11-05 http://battlelog.battlefield.com

[15] Electronic Arts, *Battlefield 3*, 2011-11-05, http://www.battlefield.com/battlefield3

Appendix A

rampstats.py

```
from stats import percentile, RequestStats
from core import locust_runner, DistributedLocustRunner
from collections import deque
import events
import math

master_response_times = deque([])
slave_response_times = []

# Are we running in distributed mode or not?
is_distributed = isinstance(locust_runner, DistributedLocustRunner)

# The time window in seconds that current_percentile use data from
PERCENTILE_TIME_WINDOW = 15.0

def current_percentile(percent):
    if is_distributed:
    # Flatten out the deque of lists and calculate the percentile to be
    returned
        return percentile(sorted([item for sublist in
    master_response_times for item in sublist]), percent)
     else:
        return percentile(sorted(master_response_times), percent)

def on_request_success(_, response_time, _2):
    if is_distributed:
        slave_response_times.append(response_time)
    else:
        master_response_times.append(response_time)

        # remove from the queue
        rps = RequestStats.sum_stats().current_rps
        if len(master_response_times) > rps*PERCENTILE_TIME_WINDOW:
            for i in xrange(len(master_response_times) -
        int(math.ceil(rps*PERCENTILE_TIME_WINDOW))):
                master_response_times.popleft()

def on_report_to_master(_, data):
    global slave_response_times
    data["current_responses"] = slave_response_times
    slave_response_times = []

def on_slave_report(_, data):
    from core import locust_runner, SLAVE_REPORT_INTERVAL

    if "current_responses" in data:
        master_response_times.append(data["current_responses"])
```

```
# remove from the queue
slaves = locust_runner.slave_count
response_times_per_slave_count = PERCENTILE_TIME_WINDOW /
SLAVE_REPORT_INTERVAL
if len(master_response_times) > slaves *   response_times_per_slave_count:
   master_response_times.popleft()
```

Appendix B

core.py LocustRunner.start_ramping

```
def start_ramping(self, hatch_rate=None, max_locusts=1000,
              hatch_stride=100, percent=0.95,
          response_time_limit=2000, acceptable_fail=0.05,
          precision=200, start_count=0,
          calibration_time=15):

    from rampstats import current_percentile
    if hatch_rate:
        self.hatch_rate = hatch_rate

    def ramp_down_help(clients, hatch_stride):
        print "ramping down..."
        hatch_stride = max(hatch_stride/2, precision)
        clients -= hatch_stride
        self.start_hatching(clients, self.hatch_rate)
        return clients, hatch_stride

    def ramp_up(clients, hatch_stride, boundery_found=False):
        while True:
            if self.state != STATE_HATCHING:
                if self.num_clients >= max_locusts:
                    print "ramp up stopped due to max locusts limit
                        reached:", max_locusts
                    client, hatch_stride = ramp_down_help(clients,
                                                hatch_stride)
                    return ramp_down(clients, hatch_stride)
                gevent.sleep(calibration_time)
                fail_ratio = RequestStats.sum_stats().fail_ratio
                if fail_ratio > acceptable_fail:
                    print "ramp up stopped due to acceptable fail ratio %d
                        %% exceeded with fail ratio %d%%" %
                    (acceptable_fail*100, fail_ratio*100)
                    client, hatch_stride = ramp_down_help(clients,
                                                hatch_stride)
                    return ramp_down(clients, hatch_stride)
                p = current_percentile(percent)
                if p >= response_time_limit:
                    print "ramp up stopped due to percentile response
                        times getting high:", p
                    client, hatch_stride = ramp_down_help(clients,
                                                hatch_stride)
                    return ramp_down(clients, hatch_stride)
                if boundery_found and hatch_stride <= precision:
                    print "sweet spot found, ramping stopped!"
                    return
                print "ramping up..."
                if boundery_found:
                    hatch_stride = max((hatch_stride/2),precision)
            clients += hatch_stride
            self.start_hatching(clients, self.hatch_rate)
        gevent.sleep(1)

def ramp_down(clients, hatch_stride):
    while True:
        if self.state != STATE_HATCHING:
            if self.num_clients < max_locusts:
                gevent.sleep(calibration_time)
                fail_ratio = RequestStats.sum_stats().fail_ratio
                if fail_ratio <= acceptable_fail:
                    p = current_percentile(percent)
                    if p <= response_time_limit:
                        if hatch_stride <= precision:
                            print "sweet spot found, ramping stopped!"
                            return
                        print "ramping up..."
                        hatch_stride = max((hatch_stride/2),precision)
                        clients += hatch_stride
                        self.start_hatching(clients, self.hatch_rate)
                        return ramp_up(clients, hatch_stride, True)
            print "ramping down..."
            hatch_stride = max((hatch_stride/2),precision)
            clients -= hatch_stride
            if clients > 0:
                self.start_hatching(clients, self.hatch_rate)
            else:
                print "WARNING: no responses met the ramping
                    thresholds, check your ramp configuration,
                locustfile and \"--host\" address"
                print "ramping stopped!"
                return
        gevent.sleep(1)

if start_count > self.num_clients:
    self.start_hatching(start_count, hatch_rate)
ramp_up(start_count, hatch_stride)
```