Load Testing with Locust (Part 1)

Beau Lyddon Follow
Sep 27, 2017 · 12 min read

We believe it's critical to profile our systems under varied loads to understand their behavior. It's important to assess the system using both "bursty" traffic, sharp changes over a few minutes or even seconds, and slow, steadily changing traffic. We profile not only with changing traffic, but also steady traffic over longer periods of time — which is often neglected. The results are used to validate our assumptions about the service(s) behavior, and to configure the systems for expected production traffic.

We will explain how we use profiling data in practice using an example. For this example, we will use Google App Engine to build our service. App Engine provides a set of managed, scalable services and will automatically manage your stateless application. App Engine allows us to configure how many idle instances (think "containers") to provision beyond currently active instances. Idle instances act like a buffer to handle spikes in traffic, and the scheduler will attempt to maintain the buffer at the specified size. We can also configure how long a request will wait for an instance before a new instance will be allocated — pending latency. There are several other settings we can configure as well. App Engine is a good learning tool because services should be stateless and the advanced, automated scheduler is comparatively easy to reason about.

Let's consider a latency sensitive service. We might start by configuring the pending latency to be quite low and adding a buffer of idle instances. When a new request comes in, the scheduler routes it to an idle instance that has already been spun up — if one is available. Of course, a negative of maintaining the buffer of idle instances is that we are over provisioning — increasing compute costs. The application *should* be more responsive to sharp increases in load. However, if we do not profile our service carefully, there is the potential for us to *increase* latency. If we reduce the allowed pending time

below the median request latency plus our service's loading time, our "spin up time", we will be spinning up new instances too frequently. Those spin up times will ultimately take longer than just waiting for an existing instance to finish servicing its current request. For decreasing traffic this won't matter — we will be reducing instances anyways. For steady traffic, depending on the specific latency ratios, it could reach a over-provisioned steady state *or* it might result in constant instance churn (instances being continually spun up/down). For increasing traffic, it will spin up a tremendous number of instances — far more than necessary to maintain good performance. These nuances are why load profiling is so critical to optimizing the behavior and costs of your system. We need to conduct realistic profiling in order to measure the numbers (loading-up time, expected request latency) and combine them with our *expectations* (for idle instance count) in order to configure the system optimally.

Load testing allows us to create artificial usage of our system that mimics real usage. To do this correctly, we want to use our existing APIs in the same manner that our clients do. This means using production logs to build realistic usage patterns. If we have not yet released our system we could analyze traffic to our test instances, demos or betas. Existing tools like Locust allow us to more easily create these test scenarios.

The rest of this post walks us through installing Locust, and then creating a simple test scenario we will run against a real server. In part two we take our Locust setup and combine it with Google Container Engine (Google-hosted Kubernetes) to build a system uses multiple machines to generate significant amounts of traffic.

Locust

What is Locust

I've quoted Locust's high level description below but you can visit their documentation site to learn more.

> Locust is an easy-to-use, distributed, user load testing tool. It is intended for load-testing web sites (or other systems) and figuring out how many concurrent users a system can handle.
>
> The idea is that during a test, a swarm of locusts will attack your website. The behavior of each locust (or test user if you will) is defined by you and the swarming process is

> monitored from a web UI in real-time. This will help you battle test and identify bottlenecks in your code before letting real users in.
>
> Locust is completely event-based, and therefore it's possible to support thousands of concurrent users on a single machine. In contrast to many other event-based apps it doesn't use callbacks. Instead it uses light-weight processes, through gevent. Each locust swarming your site is actually running inside its own process (or greenlet, to be correct). This allows you to write very expressive scenarios in Python without complicating your code with callbacks.

Now that we have a rough idea of what Locust is, let's get it installed.

NOTE: If prefer Docker vs installing Locust locally, skip down to the Docker section towards the bottom of this guide.

Install Locust

Locust runs in a Python environment so you need to setup Python and its package install tools, if you don't already have them. Thankfully OS X and most Linux distros come with Python installed.

To verify that you have Python installed, open a terminal window and run the following command:

```
$ python --version
```

Hopefully you see a result like this:

```
Python 2.7.13
```

To run Locust you will need either Python 2.7.x or any version of Python 3 above 3.3. If you do not have a Python runtime installed please visit the Python site.

Once you have Python installed, we will install several packages from pypi.To do this, Python leverages Pip to install packages locally. To check if we have Pip installed open a terminal window and type

```
$ pip --version
```

You will see something like this if you have Pip installed:

```
pip 9.0.1 from /usr/lib/python2.7/site-packages (python 2.7)
```

If you do not have Pip please visit the Pip installation instructions.

Now that we have Python and Pip we can install Locust. If you prefer a virtual environment such as virtualenv you are welcome to use one. I personally install Locust into the virtual environment for the projects I am load testing, but it is not required.

Run this command to install Locust. (Alternative methods here):

```
$ pip install locustio
```

Now that we have Locust installed we can create and run a Locust script. But first, we need a server to hit.

Test Server

I created a repo we will use to build out the server and test scripts. Within that repo you will find an example_server program written in Go. If you are on OSX, and you trust binaries from the internet, you can grab the binary here; otherwise clone the repo with the following command:

```
$ git clone git@github.com:RealKinetic/locust_k8s.git
$ cd locust_k8s
```

To build the server, run:

```
$ go build examples/golang/example_server.go
```

And run the following command to start the server:

```
$ ./example_server
```

This server will listen on port `8080`. You can test it by visiting http://localhost:8080. You should see a page with:

> Our example home page.

There are two other endpoints exposed by our example server.

- `/login` accepts `POST` requests and returns a plain text response "*Login.*". The server will output "`Login Request`" to stdout when this endpoint is hit.
- `/profile` accepts `GET` requests and returns "*Profile.*". The server will output "`Profile Request`" to stdout when this endpoint is hit.

Now that we have an example server, we can create the Locust test file.

Running Locust

For this example we can use the example provided by Locust in their quick start documentation.

You can use the `locustfile.py` in our example repo, or create the file yourself.

Your `locustfile.py` should contain the following:

```
from locust import HttpLocust, TaskSet, task

class UserBehavior(TaskSet):
    def on_start(self):
```

```
        """ on_start is called when a Locust start before
            any task is scheduled
        """
        self.login()

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

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

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

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

The Locust documentation and quick start documentation provides an explanation of the contents. We highly recommend working through their docs to learn more.

Now that we have our locustfile we can do a test.

First ensure your example server is running:

```
$ ./example_server
```

Now, in a new terminal we will run Locust. We will pass it the name of our test file, `locustfile.py`, and tell it to run against our example server on port `8080` of `localhost`.

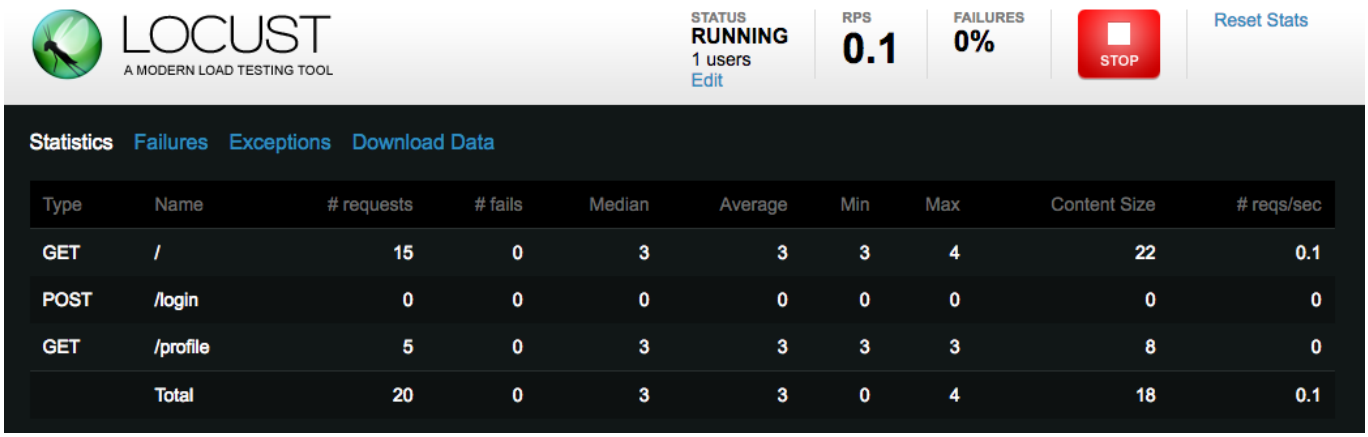
```
$ locust -f locustfile.py --host=http://localhost:8080
```

With Locust running we can open the web user interface at: http://localhost:8089.

For this test, we will simulate 1 user and specify 1 for the hatch rate. Click the `Start swarming` button. You should now see something similar to the following in the terminal running example server:

```
2017/09/13 15:24:33 Login Request
2017/09/13 15:24:33 Profile Request
2017/09/13 15:24:40 Profile Request
2017/09/13 15:24:46 Index Request
2017/09/13 15:24:55 Index Request
2017/09/13 15:25:00 Index Request
2017/09/13 15:25:07 Profile Request
2017/09/13 15:25:12 Index Request
```

In the Locust UI you will see a list of the endpoints being hit. You will see the request counts incrementing for `/` and `/profile`. There should be no failures unless Locust is having issues connecting to your server.



Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
GET	/	15	0	3	3	3	4	22	0.1
POST	/login	0	0	0	0	0	0	0	0
GET	/profile	5	0	3	3	3	3	8	0
	Total	20	0	3	3	0	4	18	0.1

Docker

We're going to start by building and running our containers locally using Docker. Install Docker for your environment using the directions on the Docker website.

NOTE: Before continuing, you should stop the `example_server` from the previous sections using `ctrl-c` in its terminal window.

Docker Environment

We will run two containers: our service (our golang example server) and our Locust instance. To support our locust container's communication with our example server we need to configure a custom Docker network. Thankfully this is a simple process.

The following command will create a custom Docker network named `locustnw`:

```
$ docker network create --driver bridge locustnw
```

You can inspect this network with the following command:

```
$ docker network inspect locustnw
```

Now that we have our network setup let's create our example server container.

Example Server Container

You can either build an image of our example server or you can pull an existing image down from Dockerhub. Dockerhub is a repository of docker images that you can push and pull images to and from.

To build our example server run the following:

```
$ docker build examples/golang -t goexample
```

This will use the `Dockerfile` we've create in the `examples/golang` directory which consists of the following:

```
# Start with a base Golang image
FROM golang

MAINTAINER Beau Lyddon <beau.lyddon@realkinetic.com>

# Add the external tasks directory into /example
RUN mkdir example
```

```
ADD example_server.go example
WORKDIR example

# Build the example executable
RUN go build example_server.go

# Set the server to be executable
RUN chmod 755 example_server

# Expose the required port (8080)
EXPOSE 8080

# Start our example service
ENTRYPOINT ["./example_server"]
```

The `-t` argument tags our container with a name, `goexample`, in this case.

If you would like to pull the image down instead run:

```
$ docker pull lyddonb/goexample
```

Now that we've created our container we can run it with the following:

```
$ docker run -it --rm -p=8080:8080 \
  --name=exampleserver \
  --network=locustnw \
  goexample
```

- The `-p` flag exposes the container's port 8080 on localhost as port 8080. This is the port our example server is listening on.
- The `--name` flag allows us to give a named identifier to the container. This allows us to reference this container by name as a host instead of by IP address. This will be critical when we run the Locust container.
- The `--network` argument tells Docker to use our custom network for this container.

NOTE: If you get the following error, stop the `example_server` we started previously (or any other services on port 8080).

```
docker: Error response from daemon: driver failed programming
external connectivity on endpoint exampleserver (...): Error starting
userland proxy: Bind for 0.0.0.0:8080 failed: port is already
allocated.
```

Since we exposed and mapped port 8080, you can test that our server is working by visiting http://localhost:8080.

Once you've verified your example server container is running, you can now build and run your Locust container.

Locust Container

Running our locust container is similar to the process we used for our example server. Once again we can either pull or build the container image. To build run the following:

```
$ docker build docker -t locust-tasks
```

This builds the `Dockerfile` located in our `docker` directory. That file consists of:

```
# Start with a base Python 2.7.13 image
FROM python:2.7.13

MAINTAINER Beau Lyddon <beau.lyddon@realkinetic.com>

# Add the external tasks directory into /locust-tasks
RUN mkdir locust-tasks
ADD locust-tasks /locust-tasks
WORKDIR /locust-tasks

# Install the required dependencies via pip
RUN pip install -r /locust-tasks/requirements.txt

# Set script to be executable
RUN chmod 755 run.sh

# Expose the required Locust ports
EXPOSE 5557 5558 8089

# Start Locust using LOCUS_OPTS environment variable
ENTRYPOINT ["./run.sh"]
```

A note: this container doesn't run Locust directly but instead uses a `run.sh` file which lives in `docker/locust-tasks`. This file is important for part 2 of our tutorial where we will run locust in a distributed mode.

To pull the image:

```
$ docker pull lyddonb/locust-tasks
```

We will briefly discuss one import part of the `run.sh` file:

```
LOCUST="/usr/local/bin/locust"
LOCUS_OPTS="-f /locust-tasks/locustfile.py --host=$TARGET_HOST"
LOCUST_MODE=${LOCUST_MODE:-standalone}

if [[ "$LOCUST_MODE" = "master" ]]; then
    LOCUS_OPTS="$LOCUS_OPTS --master"
elif [[ "$LOCUST_MODE" = "worker" ]]; then
    LOCUS_OPTS="$LOCUS_OPTS --slave --master-host=$LOCUST_MASTER"
fi

echo "$LOCUST $LOCUS_OPTS"

$LOCUST $LOCUS_OPTS
```

We rely on an environment variable named `$TARGET_HOST` being passed into our locustfile. This is key for us to communicate across containers within our Docker network.

With our container built, we can run it with a similar command as our example service.

```
$ docker run -it --rm -p=8089:8089 \
  -e "TARGET_HOST=http://exampleserver:8080" \
  --network=locustnw locust-tasks:latest
```

Once again we're exposing a port but this time it's port `8089`, the default locust port. We pass the network name to ensure this container also runs on our custom `locustnw` network.

We also pass an additional argument: `-e`. This argument sets environment variables inside the Docker container. In this case we're passing in `http://exampleserver:8080` as the variable `TARGET_HOST`. This is how the `$TARGET_HOST` environment variable needed by our `run.sh` script is set. We also see how the custom Docker network and named containers allow us to use `exampleserver` as the host name versus attempting to find the containers IP address and passing that in. This simplifies things a great deal.

Now that we have our locust server running we can visit http://localhost:8089 in a browser on our local machine. This connects to our container which will test against our example server, also running within a container.

```
$ open http://localhost:8089
```

Deployment

We can install Locust directly on any machine we'd like. Bare metal, a VM, or, in our case, we're going to install into a Docker container and deploy the container to Google Container Engine (GKE).

Google Container Engine

Prerequisites

- Google Cloud Platform account
- Install and setup Google Cloud SDK

** Be sure to run `$ gcloud init` after installing `gcloud`.

Note: when installing the Google Cloud SDK you will need to enable the following additional components:

- `Compute Engine Command Line Interface`
- `kubectl` (`$ gcloud components install kubectl`)

Set an environment variable to the cloud project you will be using:

```
$ export PROJECTID=<YOUR_PROJECT_ID>
```

Before continuing, if you did not run `gcloud init` and set your defaults, you can also set your preferred zone and project:

```
$ gcloud config set compute/zone ZONE
$ gcloud config set project $PROJECTID
```

Deploying our example container

We have summarized the key steps needed to deploy a container from the Google Container Engine Quickstart below. If you would like a more thorough walkthrough, view the quickstart guide.

First up we're going to create a cluster on GKE:

```
$ gcloud container clusters create example-cluster
```

Next, we will tag our example container that we will be pushing it in the Google Container Registry. Google Container Registry is Google's hosted Docker Registry. You can use any Docker registry that you'd prefer.

```
$ docker tag goexample gcr.io/$PROJECTID/goexample
```

Now that we've tagged our image, we can push it to the registry with the following:

```
$ gcloud docker -- push gcr.io/$PROJECTID/goexample
```

With that pushed we can now run the image:

```
$ kubectl run example-node \
  --image=gcr.io/$PROJECTID/goexample:latest --port=8080
```

This is a similar command to our local Docker command, we assigned the container a name and exposed a port. In this case however we're using `kubectl` which is the Kubernetes Command Line Inteface.

Next we expose the container. Note that the `--type="LoadBalancer"` option in kubectl requests that Google Cloud Platform provision a load balancer for your container, which is billed per the regular Load Balancer pricing.

```
$ kubectl expose deployment example-node --type="LoadBalancer"
```

Once you've exposed your container, get the IP address by running the following:

```
$ kubectl get service example-node
```

With that External IP address, open a browser to view your service running on GKE:

```
$ open http://EXTERNAL-IP:8080
```

Test with local locust

Test your newly deployed server with the following:

```
$ docker run -it --rm -p=8089:8089 \
  -e "TARGET_HOST=http://EXTERNAL-IP:8080" \
  --network=locustnw locust-tasks:latest
```

NOTE: Use the `External IP` from the earlier command (`$ kubectl get service example-node`).

We're once again taking advantage of the `$TARGET_HOST` environment variable.

Open locust in a browser http://localhost:8089:

```
$ open http://localhost:8089
```

To run locust on GKE, follow the same process as for the example server. You will need to replace `goexample` with the locust container image `locus-tasks`. You can run Locust within the same cluster or create a new cluster. In a real environment, to get a more realistic representation, separate your Load tester from the system under test.

Cleanup

Having these containers running will cost you money, remove them when you are done.

First delete the example-node from the cluster:

```
$ kubectl delete service example-node
```

Once that's been removed we can then delete the cluster itself:

```
$ gcloud container clusters delete example-cluster
```

Part 1 Complete

We now have a working example_server and a Locust file we can use to load test that server. Locust is multi-threaded, and can create a decent amount of traffic, but it is limited by the single machine's resources. The true power of Locust comes in its ability to distribute out over multiple machines. However, creating a clustered environment is more involved. In part two we'll walk through leveraging Google Container Engine and Locust's distributed mode to build a maintainable, distributed environment to run larger load tests from.

Continue on to Part 2

Docker Load Testing Kubernetes Python Locust