Load Testing with Locust (Part 2)

Beau Lyddon Follow
Oct 5, 2017 · 13 min read

Docker, Kubernetes and Google Container Engine

In Part 1 we walked through setting up Locust. We ran a single instance locally and then we deployed it as a single node to Google Container Engine (GKE). In this post, we're going leverage GKE (and Kubernetes) to deploy and run Locust in distributed mode.

Distributed Locust

Locust distributed mode allows you to concurrently run your locust files on multiple machines. You can take a look at the documentation to learn more if you'd like, but it is pretty straightforward to setup. From the Locust docs:

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

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

This design fits well with Kubernetes and Google Container Engine. We can create a single master node and then as many worker nodes as we deem necessary. The nice thing about using Kubernetes is that we change the number of nodes at run-time. To run on GKE, we only need to create a few configuration files, walk through a couple of setup steps, and then we'll be running with as many machines as we'd like.

Distributed Locust on Google Container Engine

First, a few notes on GKE and Kubernetes. Between GKE, Kubernetes and Docker we are integrating three distinct ecosystems. Additionally, Docker is platform agnostic *and* is building their own runtime eco-system outside of Kubernetes. Kubernetes, while started within Google, is a true open source project that is attempting to be platform agnostic (so it can run in any environment) and is also *container* agnostic. In other words, Kubernetes supports container types beyond Docker. These designs allow us to plugin the pieces that make the most sense for our use case. However, the negative is that understanding the APIs and documentation can be difficult. Some commands that appear to be Kubernetes specific will trigger side effects on GKE. The end result is that some bash commands are nesting commands to be passed through to docker via Kubernetes.

Also, all of these projects are in relative infancy and constantly evolving. This makes a guide like this difficult to keep up to date. We highly recommend you review the Google Cloud Documentation, specifically the Google Compute Engine, Container Engine, and Cloud Networking documentation, in addition to the Kubernetes and Docker documentation to get an intuition and *especially* if you hit issues as you go through the tutorial.

We also recommend hitting up the Slack communities (GCP, Kubernetes, Docker) for help. We have found the communities around each of these projects extremely helpful.

Now, let's get started. In this example we'll use 7 worker nodes with a single master node.

Configuration Files

The configurations for our master, service definition, and workers are in kubernetes-config. You should see the following files in that directory:

- locust-master-controller.yaml
- locust-master-service.yaml
- locust-worker-controller.yaml

Master Controller (Replication Controller)

The master controller file (locust-master-controller.yaml) configures a Kubernetes Replication Controller. From the docs:

```
A ReplicationController ensures that a specified number of pod
replicas are running at any one time. In other words, a
ReplicationController makes sure that a pod or a homogeneous set of
pods is always up and available.
```

The replication controller ensures we always have the correct number of pods running for our cluster.

In our controller we've defined 1 replica that has 8 containers. They will all use the same Docker image with the same environment variables:

```
- LOCUST_MODE
  - The mode for this locust instance to run in. In this case
`master`
- TARGET_HOST
  - This is mentioned in Part 1 and is the target host of the
endpoint we'll be testing against.
```

Note: Below we will walk through updating these parameters for the specific scenario we'll be running.

Note: Kubernetes no longer recommends using Replication Controllers and instead now recommends Deployments for managing replication. To use Deployments, you need to ensure that your Kubernetes setup supports apiVersion `apps/v1beta2`. If Deployments are not yet supported by your Kubernetes setup you can continue to use Replication Controllers.

Master Service

The next file is the master service file (locust-master-service.yaml). It defines a Kubernetes Service. What are services? Once again from the Kubernetes documentation:

```
Kubernetes Pods are mortal. They are born and when they die, they are
not resurrected. ReplicationControllers in particular create and
destroy Pods dynamically (e.g. when scaling up or down or when doing
rolling updates). While each Pod gets its own IP address, even those
IP addresses cannot be relied upon to be stable over time. This leads
to a problem: if some set of Pods (let’s call them backends) provides
functionality to other Pods (let’s call them frontends) inside the
Kubernetes cluster, how do those frontends find out and keep track of
which backends are in that set?

Enter Services.

A Kubernetes Service is an abstraction which defines a logical set of
Pods and a policy by which to access them - sometimes called a micro-
service. The set of Pods targeted by a Service is (usually)
determined by a Label Selector (see below for why you might want a
Service without a selector).
```

In our case our service is `Master Load Balancer` that is exposing ports on the 8 pods it's managing.

Worker Controller (Replication Controller)

The last file is the worker controller (locust-worker-controller.yaml). Like our master controller the worker controller is also a `ReplicationController`. In this case though it's role is a worker. We have configured 7 replicas for this configuration. That number can be set to whatever you'd like but can also be modified on demand. We will show an example of that below. Also note this controller uses the same image again.

There are 3 environment variables used to configure the worker nodes:

```
- LOCUST_MODE:
  - Similar to the master but in this case it's `worker`
- LOCUST_MASTER:
  - The worker needs to configure the master and we've called our
master `locust-master`
- TARGET_HOST:
  - The same target host again
```

Now that we’ve walked through the 3 configuration files, we can tweak them for our use case and start the deployment process.

Deploy Controllers and Services

For the next section we'll be interacting with Google Cloud Platform via their command line interface gcloud. Ensure you have the tools installed or follow their install documentation if not.

Also, ensure your tools are authenticated against the Google Cloud Project that you wish to deploy the cluster into (you can verify the project by running `kubectl get pods`). This does *NOT* have to be the same cloud project running your service-under-test. For simplicity, this guide we will assume they are running in the same cloud project. To login you can use `gcloud auth login`.

Configure the Controller Hosts

Before deploying the `locust-master` and `locust-worker` controllers, update each to point to the location of your deployed sample web application. Set the `TARGET_HOST` environment variable found in the `locust-master-controller` and `locust-worker-controller` spec entires to the web application URL. For this example it will be the same EXTERNAL-IP as we used above. Be sure to include the port.

```
- name: TARGET_HOST
  key: TARGET_HOST
  value: http://EXTERNAL-IP:8080
```

Configure the Controller Docker Image

Now we need to tag our existing Locust Docker Image so we can deploy it to Google Image Repository. If you do not already have an image created, go to Part 1 of this tutorial and follow those directions. If you did not follow the "Before you begin" instructions from the Google Container Engine Quick Start, ensure you have a Google Cloud Project available to use. Once you have a Google Cloud Project setup, replace PROJECT-ID throughout the rest of this guide with your Google Cloud Project Id.

To help with this, you may run the following:

```
$ export PROJECTID=<YOUR CLOUD PROJECT-ID>
```

Be sure to replace `<YOUR CLOUD PROJECT-ID>` with your cloud project's id.

```
$ docker tag locust-tasks gcr.io/$PROJECTID/locust-tasks
$ gcloud docker -- push gcr.io/$PROJECTID/locust-tasks
```

Note: you are not required to use the Google Container Registry. If you'd like to publish your images to the Docker Hub please refer to the steps in Working with Docker Hub.

Once the Docker image has been rebuilt and uploaded to the registry you will need to update the controllers with your new image location. The `spec.template.spec.containers.image` field in each controller controls which Docker image to use.

If you uploaded your Docker image to the Google Container Registry:

```
image: gcr.io/$PROJECTID/locust-tasks:latest
```

If you uploaded your Docker image to the Docker Hub:

```
image: USERNAME/locust-tasks:latest
```

Note: the image location includes the `latest` tag so that the image is pulled down every time a new Pod is launched. To use a Kubernetes-cached copy of the image, remove `:latest` from the image location.

Deploy Kubernetes Cluster

First create the Google Container Engine cluster using the `gcloud` command as shown below.

Note: This command defaults to creating a three node Kubernetes cluster (not counting the master) using the `n1-standard-1` machine type. Refer to the `gcloud container`

`clusters create` documentation information on specifying a different cluster configuration.

```
$ gcloud container clusters create CLUSTER-NAME
```

If you do not have a default Google Cloud Project Id set you can append the `--project=` argument to all of your gcloud commands like so:

```
$ gcloud container clusters create CLUSTER-NAME --project=$PROJECTID
```

If you do not know if you have a project set run the following command:

```
$ glcoud config list
```

And you should see an output like so:

```
[compute]
zone = us-central1-a
[core]
account = email.address@somedomain.com
project = your-google-cloud-project-id
```

I recommend setting both your project and compute zone. You can find the directions for both in the Google Cloud Tools Documentation.

Now let's take a look at the type of machines we can run our container nodes on. Run the following command to get a list:

```
$ gcloud compute machine-types list
```

You can also visit the Google Compute Machine Type Documenation to learn more.

In our case we're going to use some higher cpu machines as CPU is often the limiting factor when running locust. The command below uses the `n1-highcpu-8` machine type. This is a High-CPU machine type with 8 virtual CPUs and 7.20 GB of memory. Obviously the more powerful machines you use the more expensive they are - use whatever is comfortable for you. Note that if the locust system starts hitting failures, you may need to increase the CPU or Memory of your machines, or add more machines.

Since we have 7 worker nodes, with 1 controller, we need to let container engine know by passing in the `--num-node` parameter with a value of 8 in our case. The command looks like so:

```
$ gcloud container clusters create CLUSTER-NAME \
  --machine-type=n1-highcpu-8 --num-nodes=8
```

Let's call our cluster `locust-cluster`. The exact command we will run is:

```
$ gcloud container clusters create locust-cluster \
  --machine-type=n1-highcpu-8 --num-nodes=8
```

Now, let's make the cluster our default cluster in this project by adding it to our gcloud config with the following command:

```
$ gcloud config set container/cluster locust-cluster
```

You can run `gcloud config list` again to confirm the following entry:

```
[container]
cluster = locust-cluster
```

Get the credentials for your cluster.

```
$ gcloud container clusters get-credentials locust-cluster
```

List your clusters by issuing the following command:

```
$ gcloud container clusters list
```

You should see your newly created locust-cluster. If you created the example-cluster from Part 1 it should also be listed.

After a few minutes, you'll have a working Kubernetes cluster with three nodes (not counting the Kubernetes master). This provisions our GCP VMs that will serve as the hosts for our Kubernetes pods that we'll deploy next.

Next we will get ready to deploy our nodes. We will now interact with Kubernetes more directly via kubectl. We will be using some simple kubectl commands in this guide, but I highly recommend reading the documentation to see what is available. There are many commands for updating and inspecting services. Here is a kubectl cheat sheet for some of the most common commands.

Now it is time to deploy our pods. First setup our credentials for working with the cluster.

```
$ gcloud container clusters get-credentials locust-cluster
```

Next, ensure the `kubectl` command is pointing at your cluster. If you run the following command you should see a list of contexts:

```
$ kubectl config get-clusters
```

Ideally you will see something like `gke_PROJECT-ID_us-central1-a_locust-cluster` with an asterisk `*` to signify that it is your default context. If it is not your default, run the following command.

```
$ kubectl config use-context gke_PROJECT-ID_ZONE_CLUSTER-NAME
```

Once again verify with:

```
$ kubectl config get-clusters
```

Deploy locust-master

Now that `kubectl` is setup, we're going to deploy the `locust-master-controller` by issuing a create command pointed at the master controller yaml file:

```
$ kubectl create -f kubernetes-config/locust-master-controller.yaml
```

To confirm the Replication Controller and Pod were created, run the following:

```
$ kubectl get rc
```

The output should look something like:

```
NAME            DESIRED   CURRENT   READY     AGE
locust-master   1         1         1         41s
```

Now run:

```
$ kubectl get pods -l name=locust,role=master
```

Which will output:

```
NAME                   READY     STATUS    RESTARTS   AGE
locust-master-ltg5k    1/1       Running   0          1m
```

Next, deploy the `locust-master-service`:

```
$ kubectl create -f kubernetes-config/locust-master-service.yaml
```

To check the service status run:

```
$ kubectl get svc
```

This step will expose the Pod with an internal DNS name (`locust-master`) and ports `8089`, `5557` - `5563`. As part of this step, the `type: LoadBalancer` directive in `locust-master-service.yaml` instructs Google Container Engine to create a Google Compute Engine forwarding-rule from a publicly available IP address to the `locust-master` Pod. To view the newly created forwarding-rule, execute the following:

```
$ gcloud compute forwarding-rules list
```

Deploy locust-worker

Next up is deploying our workers. We will deploy `locust-worker-controller` with the following:

```
$ kubectl create -f kubernetes-config/locust-worker-controller.yaml
```

The `locust-worker-controller` is set to deploy 7 `locust-worker` Pods, to confirm they were deployed run the following:

```
$ kubectl get pods -l name=locust,role=worker
```

You should see output similar to:

```
NAME                  READY     STATUS    RESTARTS   AGE
locust-worker-07m8v   1/1       Running   0          29s
locust-worker-21f8p   1/1       Running   0          29s
locust-worker-0j3ln   1/1       Running   0          29s
...
```

OPTIONAL: To scale the number of `locust-worker` Pods, issue a replication controller `scale` command. You can scale your pods up or down.

```
$ kubectl scale --replicas=20 replicationcontrollers locust-worker
```

To confirm that the Pods have launched and are ready, get the list of `locust-worker` Pods:

```
$ kubectl get pods -l name=locust,role=worker
```

Note: depending on the desired number of `locust-worker` Pods, the Kubernetes cluster may need to be launched with more than 3 compute engine nodes and may also need a machine type more powerful than n1-standard-1. Refer to the `gcloud container clusters create` documentation for more information.

Setup Firewall Rules

The final step in deploying these controllers and services is to allow traffic from your publicly accessible forwarding-rule IP address to the appropriate Container Engine instances. The firewall rules *should* be added by default. However, if they were not you can use this section as a setup guide. If they are setup you can skip to the execution stage. To verify run the following:

```
$ gcloud compute firewall-rules list
```

You should see some locust cluster rules listed:

```
NAME                                          NETWORK  SRC_RANGES
RULES
SRC_TAGS  TARGET_TAGS
gke-locust-cluster-19d08658-all               default  10.16.0.0/14
tcp,udp,icmp,esp,ah,sctp
gke-locust-cluster-19d08658-ssh               default  104.155.155.130/32
tcp:22
gke-locust-cluster-19d08658-node
gke-locust-cluster-19d08658-vms               default  10.128.0.0/9
tcp:1-65535,udp:1-65535,icmp
gke-locust-cluster-19d08658-node
k8s-fw-a022b849d81f311e79d7442010a8001e  default  0.0.0.0/0
tcp:8089,tcp:5557,tcp:5558,tcp:5559,tcp:5560,tcp:5561,tcp:5562,tcp:55
63            gke-locust-cluster-19d08658-node
```

The target tag is the node name prefix up to `-node` and is formatted as `gke-CLUSTER-NAME-[...]-node`. For example, if your node name is `gke-mycluster-12345678-node-abcd`, the target tag would be `gke-mycluster-12345678-node`.

If the locust items are listed then continue on. Otherwise to create the firewall rule, execute the following:

```
$ gcloud compute firewall-rules create FIREWALL-RULE-NAME \
  --allow=tcp:8089 --target-tags gke-CLUSTER-NAME-[...]-node
```

Execute Tests

To execute the Locust tests, open the IP address of your forwarding-rule (see above) on port `8089` in your browser. Next enter the number of clients to spawn, the client hatch rate, and finally start the simulation.

You can run `kubectl get services` to view your service which will have the EXTERNAL-IP Address listed with it:

```
NAME            CLUSTER-IP      EXTERNAL-IP     PORT(S)
AGE
locust-master   10.19.251.225   35.193.138.78
8089:30218/TCP,5557:30894/TCP,5558:32095/TCP,5559:32301/TCP,5560:3060
2/TCP,5561:30879/TCP,5562:30160/TCP,5563:31944/TCP   10m
```

And now you can run your tests. My external IP is `35.193.138.78`, so I would visit open `http://35.193.138.78:8089/` in my browser.

Managing

We recommend you start with a single user test to verify the system is working correctly before attempting to run larger or more complex scenarios. The most common commands for interacting with pods, nodes, and clusters are found on the cheatsheet. There are commands for inspecting your cluster if you encounter any issues. Tailing the logs (`kubectl logs -f my-pod` or `kubectl logs -f my-pod -c my-container`) is often the quickest way to see if any issues are arising.

For more information on managing Container Engine clusters, visit the Kubernetes Resource Management documentation.

Deployment Cleanup

Once you have run your tests be sure to cleanup your locust cluster to avoid accruing unnecessary costs. To tear down the workload simulation cluster, use the following steps. First, delete the Locust Kubernetes cluster:

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

Next, delete the forwarding rule that forwards traffic into the cluster. To find the forwarding rule, run:

```
$ gcloud compute forwarding-rules list
```

You should see a rule that has your IP_ADDRESS. Copy the name for that entry and use it with the following command:

```
$ gcloud compute forwarding-rules delete FORWARDING-RULE-NAME
```

Finally, delete the firewall rule that allows incoming traffic to the cluster. Similar to the forwarding rules, first list the firewall rules:

```
$ gcloud compute firewall-rules list
```

Then use the names for any firewall rules containing locust in the name with the following command:

```
$ gcloud compute firewall-rules delete FIREWALL-RULE-NAME
```

You can use these same steps to cleanup the example service as well. You may need to update your default context first. You can switch it by listing the clusters:

```
$ kubectl config get-clusters
```

Then take the example name and set it as default:

```
$ kubectl config use-context CONTEXT-NAME
```

Note: Please don't forget to clean your resources up! Nobody likes seeing a shocking bill at the end of the month.

The End

That concludes our walk-through of running Locust in distributed mode with Kubernetes and deployed to Google Container Engine. If you'd like to dig deeper into Kubernetes and really understand how it works, then I highly recommend Kelsey Hightower's Kubernetes the Hard Way.

If you or your company are looking for help using Google Cloud Platform, Kubernetes, or need help shipping a product, then contact us. You can also find us on Twitter: @real_kinetic and LinkedIn. You can also find me on twitter: @lyddonb.

The code for this tutorial can be found on our GitHub.

Docker Kubernetes Locust Load Testing Google Cloud Platform

Medium About Help Legal