Getting Started with Docker Swarm

Romin Irani
romin.k.irani@gmail.com
@iRomin

Part 1 : Getting Started with Docker Swarm

Part 2 : Docker Swarm on Google Compute Engine

Part 1

Getting Started with Docker Swarm

Container Orchestration systems is where the next action is likely to be in the movement towards Building →Shipping → Running containers at scale. The list of software that currently provides a solution for this are Kubernetes, Docker Swarm, Apache Mesos and other.

This part is is going to be about exploring the new Docker Swarm mode, where the Container Orchestration support got baked into the Docker toolset itself. In the next part, we will look at running Docker Swarm on Google Compute Engine.

Why do we want a Container Orchestration System?

To keep this simple, imagine that you had to run hundreds of containers. You can easily see that if they are running in a distributed mode, there are multiple features that you will need from a management angle to make sure that the cluster is up and running, is healthy and more.

Some of these necessary features include:

- Health Checks on the Containers
- Launching a fixed set of Containers for a particular Docker image
- Scaling the number of Containers up and down depending on the load
- Performing rolling update of software across containers
- and more…

Let us look at how we can do some of that using Docker Swarm. The Docker Documentation and tutorial for trying out Swarm mode has been excellent.

Pre-requisites

- You are familiar with basic Docker commands
- You have Docker Toolbox installed on your system
- You have the Docker version 1.12 atleast

Create Docker Machines

The first step is to create a set of Docker machines that will act as nodes in our Docker Swarm. I am going to create 6 Docker Machines, where one of them will act as the Manager (Leader) and the other will be worker nodes. You can create less number of machines as needed.

I use the standard command to create a Docker Machine named manager1 as shown below:

```
docker-machine create --driver hyperv manager1
```

Keep in mind that I am doing this on Windows 10, which uses the native Hyper-V manager so that's why I am using that driver. If you are using the Docker Toolbox with Virtual Box, it would be something like this:

```
docker-machine create --driver virtualbox manager1
```

Similarly, create the other worder nodes. In my case, as mentioned, I have created 5 other worker nodes.

After creating, it is advised that you fire the docker-machine ls command to check on the status of all the Docker machines (I have omitted the DRIVER).

```
NAME     DRIVER URL                      STATE
manager1 hyperv tcp://192.168.1.8:2376  Running
worker1  hyperv tcp://192.168.1.9:2376  Running
worker2  hyperv tcp://192.168.1.10:2376 Running
worker3  hyperv tcp://192.168.1.11:2376 Running
worker4  hyperv tcp://192.168.1.12:2376 Running
worker5  hyperv tcp://192.168.1.13:2376 Running
```

Note down the IP Address of the manager1, since you will be needing that. I will call that MANAGER_IP in the text later.

One way to get the IP address of the manager1 machine is as follows:

```
$ docker-machine ip manager1
192.168.1.8
```

You should be comfortable with doing a SSH into any of the Docker Machines. You will need that since we will primarily be executing the docker commands from within the SSH session to that machine.

Keep in mind that using docker-machine utility, you can SSH into any of the machines as follows:

```
docker-machine ssh <machine-name>
```

As an example, here is my SSH into manager1 docker machine.

```
$ docker-machine ssh manager1
                        ##         .
                  ## ## ##        ==
               ## ## ## ## ##    ===
           /"""""""""""""""""\___/ ===
      ~~~ {~~ ~~~~ ~~~ ~~~~ ~~~ ~ /  ===- ~~~
           \______ o           __/
             \    \         __/
              \____\_______/
_                 _   ____     _            _
| |__   ___   ___ | |_|___ \ __| | ___   ___| | _____ _ __
| '_ \ / _ \ / _ \| __| __) / _` |/ _ \ / __| |/ / _ \ '__|
| |_) | (_) | (_) | |_ / __/ (_| | (_) | (__|   <  __/ |
|_.__/ \___/ \___/ \__|_____\__,_|\___/ \___|_|\_\___|_|
Boot2Docker version 1.12.1, build HEAD : ef7d0b4 - Thu Aug 18
21:18:06 UTC 2016
Docker version 1.12.1, build 23cf638
docker@manager1:~$
```

Our Swarm Cluster

Now that our machines are setup, we can proceed with setting up the Swarm.

The first thing to do is initialize the Swarm. We will SSH into the manager1 machine and initialize the swarm in there.

```
$ docker-machine ssh manager1
```

This will initialize the SSH session and you should be at prompt as shown below:

```
$ docker-machine ssh manager1
                        ##         .
                  ## ## ##        ==
               ## ## ## ## ##    ===
           /"""""""""""""""""\___/ ===
      ~~~ {~~ ~~~~ ~~~ ~~~~ ~~~ ~ /  ===- ~~~
           \______ o           __/
             \    \         __/
              \____\_______/
```

```
 _                 _   ____     _            _
| |__   ___   ___ | |_|___ \ __| | ___   ___| | _____ _ __
| '_ \ / _ \ / _ \| __| __) / _` |/ _ \ / __| |/ / _ \ '__|
| |_) | (_) | (_) | |_ / __/ (_| | (_) | (__|   <  __/ |
|_.__/ \___/ \___/ \__|_____\__,_|\___/ \___|_|\_\___|_|
Boot2Docker version 1.12.1, build HEAD : ef7d0b4 - Thu Aug 18
21:18:06 UTC 2016
Docker version 1.12.1, build 23cf638
docker@manager1:~$
```

Perform the following steps:

```
$ docker swarm init --advertise-addr MANAGER_IP
```

On my machine, it looks like this:

```
docker@manager1:~$ docker swarm init — advertise-addr 192.168.1.8
Swarm initialized: current node (5oof62fetd4gry7o09jd9e0kf) is now
a manager.

To add a worker to this swarm, run the following command:
docker swarm join \
— token
SWMTKN-1–5mgyf6ehuc5pfbmar00njd3oxv8nmjhteejaald3yzbef7osl1-ad7b1k
8k3bl3aa3k3q13zivqd \
192.168.1.8:2377

To add a manager to this swarm, run ‘docker swarm join-token
manager’ and follow the instructions.

docker@manager1:~$
```

Great!

You will also notice that the output mentions the docker swarm join command to use in case you want another node to join as a worker. Keep in mind that you can have a node join as a worker or as a manager. At any point in time, there is only one LEADER and the other manager nodes will be as backup in case the current LEADER opts out.

At this point you can see your Swarm status by firing the following command as shown below:

```
docker@manager1:~$ docker node ls
ID               HOSTNAME STATUS AVAILABILITY MANAGER STATUS
5oof62fetd..*    manager1 Ready  Active       Leader
```

This shows that there is a single node so far i.e. manager1 and it has the value of Leader for the MANAGER column.

Stay in the SSH session itself for manager1.

Joining as Worker Node

To find out what docker swarm command to use to join as a node, you will need to use the join-token <role> command.

To find out the join command for a worker, fire the following command:

```
docker@manager1:~$ docker swarm join-token worker
To add a worker to this swarm, run the following command:
docker swarm join \
— token
SWMTKN-1–5mgyf6ehuc5pfbmar00njd3oxv8nmjhteejaald3yzbef7osl1-ad7b1k
8k3bl3aa3k3q13zivqd \
192.168.1.8:2377

docker@manager1:~$
```

Joining as Manager Node

To find out the the join command for a manager, fire the following command:

```
docker@manager1:~$ docker swarm join-token manager
To add a manager to this swarm, run the following command:
docker swarm join \
— token
SWMTKN-1–5mgyf6ehuc5pfbmar00njd3oxv8nmjhteejaald3yzbef7osl1–8xo0cm
d6bryjrsh6w7op4enos \
192.168.1.8:2377

docker@manager1:~$
```

Notice in both the above cases, that you are provided a token and it is joining the Manager node (you will be able to identify that the IP address is the same the MANAGER_IP address).

Keep the SSH to manager1 open. And fire up other command terminals for working with other worker docker machines.

Adding Worker Nodes to our Swarm

Now that we know how to check the command to join as a worker, we can use that to do a SSH into each of the worker Docker machines and then fire the respective join command in them.

In my case, I have 5 worker machines (worker1/2/3/4/5). For the first worker1 Docker machine, I do the following:

- SSH into the worker1 machine i.e. docker-machine ssh worker1
- Then fire the respective command that I got for joining as a worker. In my case the output is shown below:

  ```
  docker@worker1:~$ docker swarm join \
  — token
  SWMTKN-1–5mgyf6ehuc5pfbmar00njd3oxv8nmjhteejaald3yzbef7osl1-a
  d7b1k8k3bl3aa3k3q13zivqd \
  192.168.1.8:2377
  This node joined a swarm as a worker.
  docker@worker1:~$
  ```

I do the same thing by launching SSH sessions for worker2/3/4/5 and then pasting the same command since I want all of them to be worker nodes.

After making all my worker nodes join the Swarm, I go back to my manager1 SSH session and fire the following command to check on the status of my Swarm i.e. see the nodes participating in it:

```
docker@manager1:~$ docker node ls
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
1ndqsslh7fpquc7fi35leig54    worker4   Ready   Active
1qh4aat24nts5izo3cgsboy77    worker5   Ready   Active
25nwmw5eg7a5ms4ch93aw0k03    worker3   Ready   Active
5oof62fetd4gry7o09jd9e0kf *  manager1  Ready   Active        Leader
5pm9f2pzr8ndijqkkblkgqbsf    worker2   Ready   Active
9yq4lcmfg0382p39euk8lj9p4    worker1   Ready   Active

docker@manager1:~$
```

As expected, you can see that I have 6 nodes, one as the manager (manager1) and the other 5 as workers.

We can also do execute the standard docker info command here and zoom into the Swarm section to check out the details for our Swarm.

```
Swarm: active
NodeID: 5oof62fetd4gry7o09jd9e0kf
Is Manager: true
ClusterID: 6z3sqr1aqank2uimyzijzapz3
Managers: 1
Nodes: 6
Orchestration:
 Task History Retention Limit: 5
Raft:
 Snapshot Interval: 10000
 Heartbeat Tick: 1
 Election Tick: 3
Dispatcher:
 Heartbeat Period: 5 seconds
CA Configuration:
 Expiry Duration: 3 months
Node Address: 192.168.1.8
```

Notice a few of the properties:

- The Swarm is marked as active. It has 6 Nodes in total and 1 manager among them.
- Since I am running the docker info command on the manager1 itself, it shows the Is Manager as true.
- The Raft section is the Raft consensus algorithm that is used. Check out the details here.

Create a Service

Now that we have our swarm up and running, it is time to schedule our containers on it. This is the whole beauty of the orchestration layer. We are going to focus on the app and not worry about where the application is going to run.

All we are going to do is tell the manager to run the containers for us and it will take care of scheduling out the containers, sending the commands to the nodes and distributing it.

To start a service, you would need to have the following:

- What is the Docker image that you want to run. In our case, we will run the standard nginx image that is officially available from the Docker hub.
- We will expose our service on port 80.

- We can specify the number of containers (or instances) to launch. This is specified via the replicas parameter.
- We will decide on the name for our service. And keep that handy.

What I am going to do then is to launch 5 replicas of the nginx container. To do that, I am again in the SSH session for my manager1 node. And I give the following docker service create command:

```
docker service create --replicas 5 -p 80:80 --name web nginx
ctolq1t4h2o859t69j9pptyye
```

What has happened is that the Orchestration layer has now got to work.

You can find out the status of the service, by giving the following command:

```
docker@manager1:~$ docker service ls
ID            NAME  REPLICAS  IMAGE  COMMAND
ctolq1t4h2o8  web   0/5       nginx
```

This shows that the replicas are not yet ready. You will need to give that command a few times.

In the meanwhile, you can also see the status of the service and how it is getting orchestrated to the different nodes by using the following command:

```
docker@manager1:~$ docker service ps web
ID   NAME   IMAGE  NODE      DESIRED STATE CURRENT STATE      ERROR
7i*  web.1  nginx  worker3   Running       Preparing 2 minutes ago
17*  web.2  nginx  manager1  Running       Running 22 seconds ago
ey*  web.3  nginx  worker2   Running       Running 2 minutes ago
bd*  web.4  nginx  worker5   Running       Running 45 seconds ago
dw*  web.5  nginx  worker4   Running       Running 2 minutes ago
```

This shows that the nodes are getting setup. It could take a while.

But notice a few things. In the list of nodes above, you can see that the 5 containers are being scheduled by the orchestration layer on manager1, worker2, worker3, worker4 and worker5.
There is no container scheduled for worker1 node and that is fine.

A few executions of docker service ls shows the following responses:

```
docker@manager1:~$ docker service ls
ID            NAME  REPLICAS  IMAGE  COMMAND
ctolq1t4h2o8  web   3/5       nginx
```

and then finally:

```
docker@manager1:~$ docker service ls
ID              NAME REPLICAS IMAGE COMMAND
ctolq1t4h2o8    web  5/5      nginx
```

If we look at the service processes at this point, we can see the following:

```
docker@manager1:~$ docker service ps web
ID  NAME   IMAGE  NODE      DESIRED STATE  CURRENT STATE
ERROR
7i*  web.1  nginx  worker3   Running        Running 4 minutes ago
17*  web.2  nginx  manager1  Running        Running 7 minutes ago
ey*  web.3  nginx  worker2   Running        Running 9 minutes ago
bd*  web.4  nginx  worker5   Running        Running 8 minutes ago
dw*  web.5  nginx  worker4   Running        Running 9 minutes ago
```

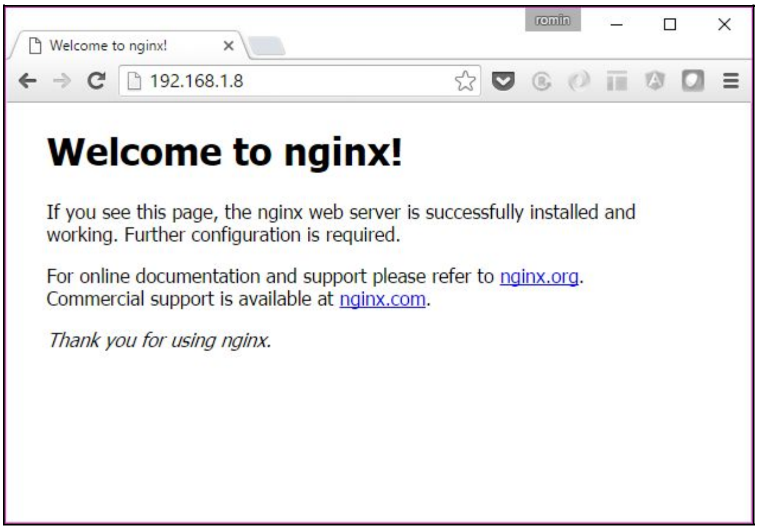
If you do a docker ps on the manager1 node right now, you will find that the nginx daemon has been launched.

```
docker@manager1:~$ docker ps
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS              PORTS               NAMES
933309b04630        nginx:latest        "nginx -g 'daemon off"   2
minutes ago       Up 2 minutes       80/tcp, 443/tcp
web.2.17d502y6qjhd1wqjle13nmjvc
docker@manager1:~$
```

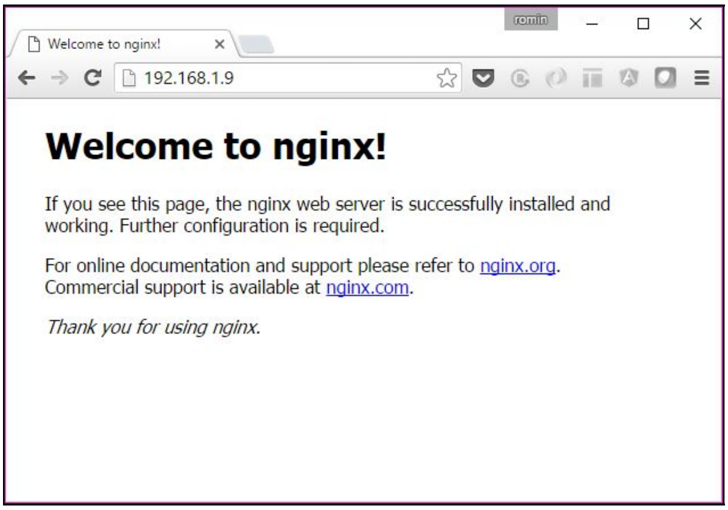
Accessing the Service

You can access the service by hitting any of the manager or worker nodes. It does not matter if the particular node does not have a container scheduled on it. That is the whole idea of the swarm.

Try out a curl to any of the Docker Machine IPs (manager1 or worker1/2/3/4/5) or hit the URL (http://<machine-ip>) in the browser. You should be able to get the standard NGINX Home page.



or if we hit the worker IP:



Nice, isn't it?

Ideally you would put the Docker Swarm service behind a Load Balancer. We will see that in the next part when we provision a Load Balancer on Google Compute Engine.

Scaling up and Scaling down

This is done via the docker service scale command. We currently have 5 containers running.

Let us bump it up to 8 as shown below by executing the command on the manager1 node.

```
$ docker service scale web=8
web scaled to 8
```

Now, we can check the status of the service and the process tasks via the same commands as shown below:

```
docker@manager1:~$ docker service ls
ID           NAME REPLICAS IMAGE COMMAND
ctolq1t4h2o8 web  5/8      nginx
```

In the ps web command below, you will find that it has decided to schedule the new containers on worker1 (2 of them) and manager1(one of them):

```
docker@manager1:~$ docker service ps web
ID   NAME   IMAGE  NODE      DESIRED STATE  CURRENT STATE                ERROR
7i*  web.1  nginx  worker3   Running        Running 14 minutes ago
17*  web.2  nginx  manager1  Running        Running 17 minutes ago
ey*  web.3  nginx  worker2   Running        Running 19 minutes ago
bd*  web.4  nginx  worker5   Running        Running 17 minutes ago
dw*  web.5  nginx  worker4   Running        Running 19 minutes ago
8t*  web.6  nginx  worker1   Running        Starting about a minute ago
b8*  web.7  nginx  manager1  Running        Ready less than a second ago
0k*  web.8  nginx  worker1   Running        Starting about a minute ago
```

We wait for a while and then everything looks good as shown below:

```
docker@manager1:~$ docker service ls
ID           NAME REPLICAS IMAGE COMMAND
ctolq1t4h2o8 web  8/8      nginx

docker@manager1:~$ docker service ps web
ID  NAME  IMAGE NODE     DESIRED STATE CURRENT STATE ERROR
7i* web.1 nginx worker3  Running       Running 16 minutes ago
17* web.2 nginx manager1 Running       Running 19 minutes ago
ey* web.3 nginx worker2  Running       Running 21 minutes ago
bd* web.4 nginx worker5  Running       Running 20 minutes ago
dw* web.5 nginx worker4  Running       Running 21 minutes ago
8t* web.6 nginx worker1  Running       Running 4 minutes ago
b8* web.7 nginx manager1 Running       Running 2 minutes ago
0k* web.8 nginx worker1  Running       Running 3 minutes ago
```

Inspecting nodes

You can inspect the nodes anytime via the docker node inspect command.

For example if you are already on the node (for example manager1) that you want to check, you can use the name self for the node.

```
$ docker node inspect self
```

Or if you want to check up on the other nodes, give the node name. For e.g.

```
$ docker node inspect worker1
```

Draining a node

If the node is ACTIVE, it is ready to accept tasks from the Master i.e. Manager. For e.g. we can see the list of nodes and their status by firing the following command on the manager1 node.

```
docker@manager1:~$ docker node ls
ID                           HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
1ndqsslh7fpquc7fi35leig54    worker4   Ready   Active
1qh4aat24nts5izo3cgsboy77    worker5   Ready   Active
25nwmw5eg7a5ms4ch93aw0k03    worker3   Ready   Active
5oof62fetd4gry7o09jd9e0kf *  manager1  Ready   Active        Leader
5pm9f2pzr8ndijqkkblkgqbsf    worker2   Ready   Active
9yq4lcmfg0382p39euk8lj9p4    worker1   Ready   Active
```

You can see that their AVAILABILITY is set to READY.

As per the documentation, When the node is active, it can receive new tasks:

- during a service update to scale up
- during a rolling update
- when you set another node to Drain availability
- when a task fails on another active node

But sometimes, we have to bring the Node down for some maintenance reason. This meant by setting the Availability to Drain mode. Let us try that with one of our nodes.

But first, let us check the status of our processes for the web services and on which nodes they are running:

```
docker@manager1:~$ docker service ps web
ID   NAME   IMAGE  NODE      DESIRED STATE  CURRENT STATE          ERROR
7i*  web.1  nginx  worker3   Running        Running 54 minutes ago
17*  web.2  nginx  manager1  Running        Running 57 minutes ago
ey*  web.3  nginx  worker2   Running        Running 59 minutes ago
bd*  web.4  nginx  worker5   Running        Running 57 minutes ago
dw*  web.5  nginx  worker4   Running        Running 59 minutes ago
8t*  web.6  nginx  worker1   Running        Running 41 minutes ago
b8*  web.7  nginx  manager1  Running        Running 39 minutes ago
0k*  web.8  nginx  worker1   Running        Running 41 minutes ago
```

You find that we have 8 replicas of our service:

- 2 on manager1
- 2 on worker1
- 1 each on worker2, worker3, worker4 and worker5

Now, let us use another command to check what is going on in node worker1.

```
docker@manager1:~$ docker node ps worker1
ID   NAME   IMAGE  NODE     DESIRED STATE  CURRENT STATE
8t*  web.6  nginx  worker1  Running        Running 44 minutes ago
0k*  web.8  nginx  worker1  Running        Running 44 minutes ago
```

We can also use the docker node inspect command to check the availability of the node and as expected, you will find a section in the output as follows:

```
$ docker node inspect worker1
…..
"Spec": {
"Role": "worker",
"Availability": "active"
},
…
```

or

```
docker@manager1:~$ docker node inspect — pretty worker1
ID: 9yq4lcmfg0382p39euk8lj9p4
Hostname: worker1
Joined at: 2016-09-16 08:32:24.5448505 +0000 utc
Status:
State: Ready
```

```
Availability: Active
Platform:
Operating System: linux
Architecture: x86_64
Resources:
CPUs: 1
Memory: 987.2 MiB
Plugins:
Network: bridge, host, null, overlay
Volume: local
Engine Version: 1.12.1
Engine Labels:
— provider = hyperv
```

We can see that it is “Active” for its Availability attribute.

Now, let us set the Availability to DRAIN. When we give that command, the Manager will stop tasks running on that node and launches the replicas on other nodes with ACTIVE availability.
So what we are expecting is that the Manager will bring the 2 containers running on worker1 and schedule them on the other nodes (manager1 or worker2 or worker3 or worker4 or worker5).

This is done by updating the node by setting its availability to “drain”.

```
docker@manager1:~$ docker node update --availability drain worker1
worker1
```

Now, if we do a process status for the service, we see an interesting output (I have trimmed the output for proper formatting):

```
docker@manager1:~$ docker service ps web
ID   NAME       IMAGE  NODE      DESIRED STATE  CURRENT STATE
7i*  web.1      nginx  worker3   Running   Running about an hour ago
17*  web.2      nginx  manager1  Running   Running about an hour ago
ey*  web.3      nginx  worker2   Running   Running about an hour ago
bd*  web.4      nginx  worker5   Running   Running about an hour ago
dw*  web.5      nginx  worker4   Running   Running about an hour ago
2u*  web.6      nginx  worker4   Running   Preparing about a min ago
8t*   \_ web.6  nginx  worker1   Shutdown  Shutdown about a min ago
b8*  web.7      nginx  manager1  Running   Running 49 minutes ago
7a*  web.8      nginx  worker3   Running   Preparing about a min ago
0k*   \_ web.8  nginx  worker1   Shutdown  Shutdown about a min ago
docker@manager1:~$
```

You can see that the containers on worker1 (which we have asked to be drained) are being rescheduled on other workers. In our scenario above, they got scheduled to worker2 and worker3 respectively. This is required because we have asked for 8 replicas to be running in an earlier scaling exercise.

You can see that the two containers are still in “Preparing” state and after a while if you run the command, they are all running as shown below:

```
docker@manager1:~$ docker service ps web
ID  NAME       IMAGE  NODE       DESIRED STATE  CURRENT STATE
7i*  web.1      nginx  worker3   Running   Running about an hour ago
17*  web.2      nginx  manager1  Running   Running about an hour ago
ey*  web.3      nginx  worker2   Running   Running about an hour ago
bd*  web.4      nginx  worker5   Running   Running about an hour ago
dw*  web.5      nginx  worker4   Running   Running about an hour ago
2u*  web.6      nginx  worker4   Running   Running 8 minutes ago
8t*   \_ web.6  nginx  worker1   Shutdown  Shutdown 8 minutes ago
b8*  web.7      nginx  manager1  Running   Running 56 minutes ago
7a*  web.8      nginx  worker3   Running   Running 8 minutes ago
0k*   \_ web.8  nginx  worker1   Shutdown  Shutdown 8 minutes ago
```

This makes for cool demo, isn’t it?

Remove the Service

You can simply use the service rm command as shown below:

```
docker@manager1:~$ docker service rm web
web

docker@manager1:~$ docker service ls
ID NAME REPLICAS IMAGE COMMAND

docker@manager1:~$ docker service inspect web
[]
Error: no such service: web
```

Applying Rolling Updates

This is straightforward. In case you have an updated Docker image to roll out to the nodes, all you need to do is fire an service update command.

For e.g.

```
$ docker service update --image <imagename>:<version> web
```

Conclusion

I am definitely impressed with the simplicity of Docker Swarm. Just like the basic commands that come with the standard Docker toolset, it has been a good move to introduce the Swarm commands within the same toolset.

There is a lot more to Docker Swarm and I suggest that you dig further into the documentation.

I am not in a position to know whether Kubernetes or Swarm will emerge a winner but there is no doubt that we will have to understand both to see what their capabilities are and then take a decision.

Part 2

Docker Swarm on Google Compute Engine

This part builds on the previous one by going through the following:

- Setup a Docker Swarm on Google Compute Engine
- Experiment multiple scenarios like setting up multiple Swarm Manager, bringing a Swarm Manager down, setting up an Overlay network and more.

The first step is to create the Docker Swarm cluster on Google Compute Engine.

Prerequisites

This tutorial assumes that you have setup the following on your machine:

- Latest version of Docker Toolbox. Ensure that it is 1.12 or higher.
- Latest version of Google Cloud SDK Tools. Please download it from the link.

Google Cloud Platform Project

I suggest that you create a new Google Cloud Platform Project for this tutorial. But if you are familiar with the platform and wish to use an existing project, that is fine too. Note down the Project Id for the Google Cloud Platform project.

On your local machine, where you have already setup Docker Toolbox + Google Cloud Platform tools and assuming that you have the Google Cloud Platform project id handy, initialize the project using the gcloud utility that you have setup.

```
$ gcloud init
```

and go ahead with the rest of the steps, ensure that you select the zone/region of your choice and most importantly the project id.

To ensure that you are all set, just fire the following command and note if the properties are setup correctly. You should see similar values.

```
$ gcloud config list
Your active configuration is: [<your-config-name>]
[compute]
region = <your-selected-region> e.g. us-central1
zone = <your-selected-zone> e.g. us-central1-a
```

```
[core]
account = <your-email-id>
disable_usage_reporting = True
project = <YOUR_GCP_PROJECT_ID>
```

If the project is not set, I suggest that you do so with the following command:

```
$ gcloud config set project <YOUR_GCP_PROJECT_ID>
```

Creating the Docker Machines

The first step to creating the swarm is to provision the Docker machines. By that, we mean that we will be provisioning Compute Engine instances. We are going to have the following setup:

- 5 Compute Engine instances, all setup with docker and provisioned using the docker-machine utility that is part of the Docker Toolbox.
- We are going to have 3 Managers in the Swarm and 2 Workers in the Swarm. We need to give the names to our Compute Engine instances, so we will name them mgr-1, mgr-2, mgr-3 and w-1 & w-2.

To provision a Docker machine (Host) on compute engine, we use the following command (for mgr-1). Note that we are using the Google Cloud driver, specifying the machine type (n1-standard-1), giving a tag to all our machines and specifying the google-project-id. This is standard docker-machine create stuff.

```
$ docker-machine create mgr-1 \
                       -d google \
                       --google-machine-type n1-standard-1 \
                       --google-tags myswarm \
                       --google-project <YOUR_GCP_PROJECT_ID>
Running pre-create checks…
(mgr-1) Check that the project exists
(mgr-1) Check if the instance already exists
Creating machine…
(mgr-1) Generating SSH Key
(mgr-1) Creating host…
(mgr-1) Opening firewall ports
(mgr-1) Creating instance
(mgr-1) Waiting for Instance
(mgr-1) Uploading SSH Key
Waiting for machine to be running, this may take a few
minutes…
Detecting operating system of created instance…
Waiting for SSH to be available…
```

```
Detecting the provisioner…
Provisioning with ubuntu(systemd)…
Installing Docker…
Copying certs to the local machine directory…
Copying certs to the remote machine…
Setting Docker configuration on the remote daemon…
Checking connection to Docker…
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine
running on this virtual machine, run: docker-machine env mgr-1
```

We do the same for mgr-2, mgr-3, w-1 and w-2. On successful creation, we can use the docker-machine ls command to check on our Docker machines. The output should be similar to the one that I got below (Note that I have removed the SWARM and the ERRORS column from the output):

```
$ docker-machine ls
NAME   ACTIVE DRIVER STATE   URL                        DOCKER
mgr-1  —      google Running tcp://130.211.199.228:2376
v1.12.1
mgr-2  —      google Running tcp://104.154.244.185:2376
v1.12.1
mgr-3  —      google Running tcp://104.154.56.35:2376
v1.12.1
w-1    —      google Running tcp://107.178.213.86:2376
v1.12.1
w-2    —      google Running tcp://8.34.214.144:2376
v1.12.1
```

At this point, you could also use the gcloud compute instances list command to see the list of VMs that have been provisioned. In the listing you should see Compute Engine VMs as listed below:

```
$ gcloud compute instances list


NAME        ZONE           MACHINE_TYPE   PREEMPTIBLE
INTERNAL_IP  EXTERNAL_IP     STATUS
mgr-1       us-central1-a  n1-standard-1
10.240.0.2   130.211.199.228  RUNNING
mgr-2       us-central1-a  n1-standard-1
10.240.0.3   104.154.244.185  RUNNING
mgr-3       us-central1-a  n1-standard-1
10.240.0.4   104.154.56.35    RUNNING
w-1         us-central1-a  n1-standard-1
10.240.0.5   107.178.213.86   RUNNING
```

```
w-2          us-central1-a  n1-standard-1
10.240.0.6    8.34.214.144      RUNNING
```

Each of the VMs has been assigned an internal and external IP. The status of all the machines is also in RUNNING state.

Creating the Swarm

In Part 1, we had seen how to create the Swarm. To reiterate, we are going to create a Swarm with:

- 3 Manager nodes (This will make 1 as as the LEADER and the other 2 as Available)
- 2 Worker nodes

SSH into Google Compute Engine VMs

One of the nice features of Google Compute Engine VMs is that you have a SSH button right next to your list of Compute Engine VMs, which you can click and get into a SSH session with that VM.

If you go to the Google Cloud Console and then select Compute Engine, you will see a list that looks something like this:

Name	Zone	Machine type	Recommendation	In use by	Internal IP	External IP	Connect
mgr-1	us-central1-a	1 vCPU, 3.75 GB			10.240.0.2	130.211.199.228	SSH
mgr-2	us-central1-a	1 vCPU, 3.75 GB			10.240.0.3	104.154.244.185	SSH
mgr-3	us-central1-a	1 vCPU, 3.75 GB			10.240.0.4	104.154.56.35	SSH
w-1	us-central1-a	1 vCPU, 3.75 GB			10.240.0.5	107.178.213.86	SSH
w-2	us-central1-a	1 vCPU, 3.75 GB			10.240.0.6	8.34.214.144	SSH

Notice the SSH button to the extreme right for each machine that we created. Click on that to launch the SSH session for any of the machines. I will use the title SSH to mgr-1 session and so on to indicate which machine I am on.

Note: You can also use the docker-machine env command on your local machine to set the environment variables that will allow the docker client to connect to a specific machine. If you are comfortable with it, use it by all means.

Note: You will notice the sudo prefix before the docker commands. If you want to avoid that, you should consider adding the user with root privileges to the docker user group. E.g. sudo usermod -aG docker <user_name>

Initialize the Swarm

First up, note down the Internal IP address of the mgr-1 instance. You will find that in the Compute Engine VM listing that we saw about in my case, it is 10.240.0.2.

- SSH to mgr-1 docker machine
- Give the following command:

```
romin_irani@mgr-1:~$ sudo docker swarm init \
                     --advertise-addr 10.240.0.2
Swarm initialized: current node (6l6qh3d1b6hps9ic095wsor27) is
now a manager.

To add a worker to this swarm, run the following command:
docker swarm join \
--token
SWMTKN-1-4lon4th27l53xvrpruohbld5lciux02rxs9go9fdt2672cdkhu-69
j9grxvmri7wni2k134m4dmw \
10.240.0.2:2377

To add a manager to this swarm, run 'docker swarm join-token
manager' and follow the instructions.

romin_irani@mgr-1:~$
```

At this point, we have only one node in our Swarm as shown below:

```
romin_irani@mgr-1:~$ sudo docker node ls

ID       HOSTNAME STATUS AVAILABILITY MANAGER STATUS
6l6.. * mgr-1    Ready  Active       Leader
```

To join the other nodes as workers or managers, we just have to know what is the token and ip to use as part of the docker swarm join command. This is made easy to simply executing the join-token <role> on the current master i.e. mgr-1 and noting down the commands:

```
romin_irani@mgr-1:~$ sudo docker swarm join-token manager
To add a manager to this swarm, run the following command:
docker swarm join \
— token
SWMTKN-1-4lon4th27l53xvrpruohbld5lciux02rxs9go9fdt2672cdkhu-43
c7vexnensp8mkvwl09preu7 \
```

```
10.240.0.2:2377
```

```
romin_irani@mgr-1:~$ sudo docker swarm join-token worker
To add a worker to this swarm, run the following command:

docker swarm join \
— token
SWMTKN-1-4lon4th27l53xvrpruohbld5lciux02rxs9go9fdt2672cdkhu-69
j9grxvmri7wni2k134m4dmw \
10.240.0.2:2377
```

You can now open up SSH sessions on each of the nodes. Remember that on mgr-2 and mgr-3 , we want to execute the command to join as a manager. And on w-1 and w-2 nodes, we want to execute the command to join as a worker.

Once you complete the commands on all the nodes, your Docker Swarm is now ready.

If we run the command to list down the Swarm nodes on mgr-1, we should get the following output:

```
romin_irani@mgr-1:~$ sudo docker node ls
```

```
ID         HOSTNAME   STATUS   AVAILABILITY   MANAGER STATUS
01l..      mgr-2      Ready    Active         Reachable
4e3..      mgr-3      Ready    Active         Reachable
6l6.. *    mgr-1      Ready    Active         Leader
7xo..      w-1        Ready    Active
8ie..      w-2        Ready    Active
```

You can see from the output above that we have 5 nodes running. Our mgr-1 (Manager) from where we launched the Swarm cluster is now a Leader. The other managers are in a Reachable state.

Great. Everything looks good for now.

Creating the Overlay Network

Let us create an overlay network now. An overlay network supports multi-host networking. We are going to be using the overlay network for our swarm services. When you specify an overlay network for your services, Swarm automatically assigns addresses to the containers.

Stay in the SSH session for mgr-1. We can look at our current list of networks as follows:

```
romin_irani@mgr-1:~$ sudo docker network list
```

```
NETWORK ID          NAME                DRIVER              SCOPE
d4b360ee71b4        bridge              bridge              local
9a55643d34e8        docker_gwbridge     bridge              local
b6348ecb0afa        host                host                local
dzzo90eqcmt2        ingress             overlay             swarm
46f4630544d0        none                null                local
```

You will notice that at the local scope, you have the default bridge and the host network. You will also notice that Docker Swarm created a default overlay network called ingress. As per the documentation, “the swarm manager uses ingress load balancing to expose the services you want to make available externally to the swarm.”

Let us go ahead now and create our own overlay network named nw1. So on the manager node (mgr-1) do the following:

```
$ sudo docker network create --driver overlay
nw17ffh8lexsm9fhiukslkyiml02
```

We can now inspect the list of network services as follows:

```
romin_irani@mgr-1:~$ sudo docker network list
```

```
NETWORK ID          NAME                DRIVER              SCOPE
d4b360ee71b4        bridge              bridge              local
9a55643d34e8        docker_gwbridge     bridge              local
b6348ecb0afa        host                host                local
dzzo90eqcmt2        ingress             overlay             swarm
46f4630544d0        none                null                local
7ffh8lexsm9f        nw1                 overlay             swarm
```

You can see that the overlay network (named nw1) is created.

Creating the Service

We learnt in the first part of this tutorial on how to create a Docker Swarm service. We are going to do just that by creating a standard NGINX service with 6 replicas. This time however, we are going to use the overlay network (nw1) that we created so that we can later on see how it all comes together when working with multiple services.

On the mgr-1 node, execute the following command:

```
romin_irani@mgr-1:~$ sudo docker service create --replicas 6
--network nw1 -p 80:80/tcp --name nginx nginx
0omlto8a98zahgbsqs0ajz159
```

We can see the services as follows:

```
romin_irani@mgr-1:~$ sudo docker service ls
ID            NAME   REPLICAS  IMAGE  COMMAND
0omlto8a98za  nginx  6/6       nginx
```

We can see that 6 containers have been launched for NGINX image. To understand the distribution of these 6 containers on the 5 nodes that we have, we can use the following command:

```
romin_irani@mgr-1:~$ sudo docker service ps nginx
ID   NAME     IMAGE  NODE   DESIRED STATE  CURRENT STATE
9z*  nginx.1  nginx  mgr-2  Running        Running 32 seconds ago
6e*  nginx.2  nginx  w-1    Running        Running 32 seconds ago
1o*  nginx.3  nginx  w-1    Running        Running 32 seconds ago
6n*  nginx.4  nginx  mgr-1  Running        Running 32 seconds ago
8l*  nginx.5  nginx  w-2    Running        Running 32 seconds ago
8p*  nginx.6  nginx  mgr-3  Running        Running 32 seconds ago
```

We can see that the Swarm Manager distributed the containers across all the 5 nodes : running 2 containers on worker node w-1 and distributing the other containers equally across all the remaining nodes.

So at this point, we have the standard NGINX container running on our 5 nodes. These 5 nodes are nothing but our VMs i.e. Google Compute Engine instances. And if you recollect, each of these Compute Engine instances were provided both an internal IP Address and an external IP Address.

You could list out the output of the gcloud compute instances list command again. You could either use that from your laptop or just go to the Compute Engine instances list in the Google Cloud console.

Name	Zone	Machine type	Recommendation	In use by	Internal IP	External IP	Connect
mgr-1	us-central1-a	1 vCPU, 3.75 GB			10.240.0.2	130.211.199.228	SSH
mgr-2	us-central1-a	1 vCPU, 3.75 GB			10.240.0.3	104.154.244.185	SSH
mgr-3	us-central1-a	1 vCPU, 3.75 GB			10.240.0.4	104.154.56.35	SSH
w-1	us-central1-a	1 vCPU, 3.75 GB			10.240.0.5	107.178.213.86	SSH
w-2	us-central1-a	1 vCPU, 3.75 GB			10.240.0.6	8.34.214.144	SSH

Internal Connectivity

You can SSH into any of the Compute Engine instances. Notice that the Internal IPs from the list of instances above. Simply use ‘curl <InternalIPAddress>’ for any of the instances and you should get back the HTML content of the default NGINX home page. So in short, the instances can communicate to each other internally via the Internal IP Addresses.

External Connectivity

To do this, we will need to create a Firewall rule to allow traffic from outside targetted towards port 80 and we should target the Compute Engine instancces that we had tagged earlier with the myswarm tag, which we used as value for the

```
--google-tags myswarm
```

while creating the Docker machine.

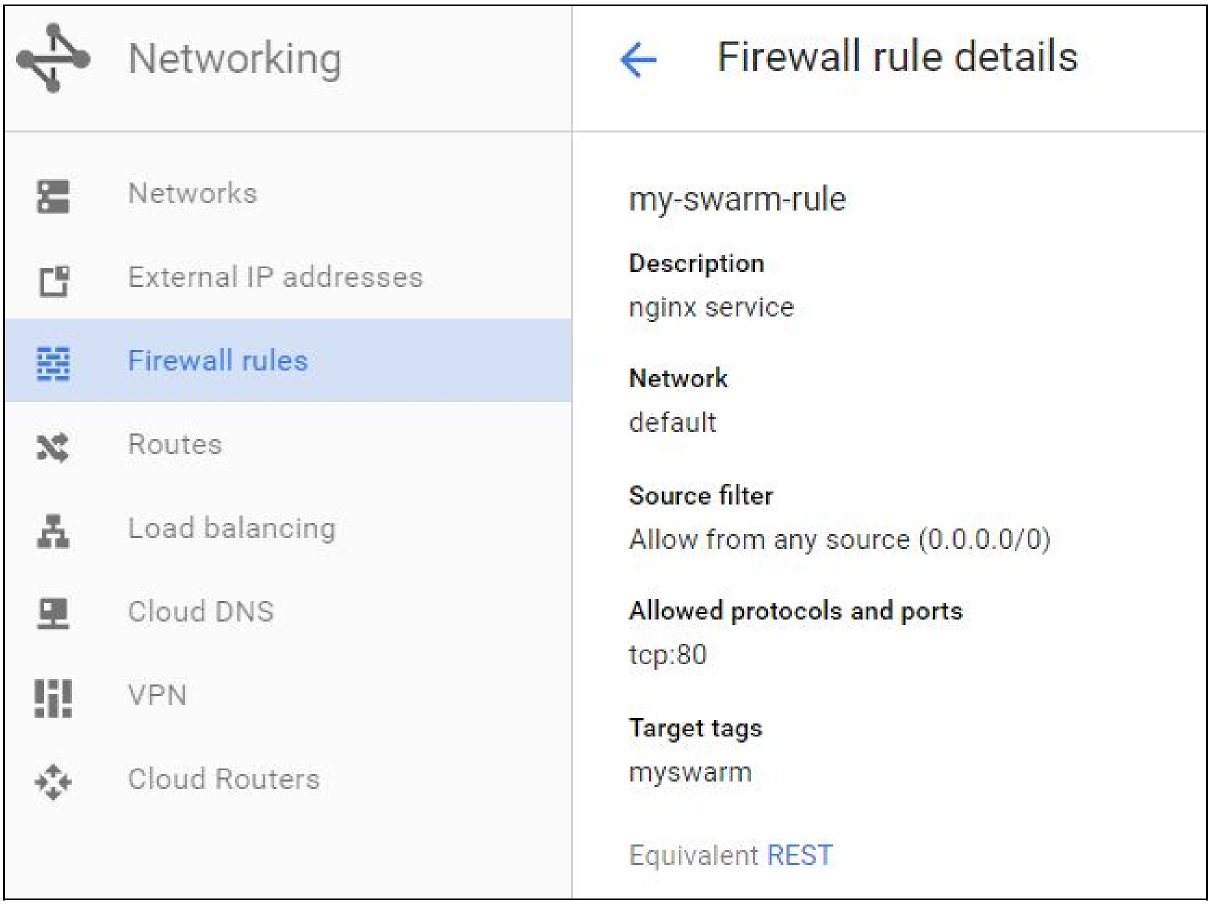
From the gcloud utility on your local machine, fire the following command:

```
$ gcloud compute firewall-rules create my-swarm-rule --allow
tcp:80 --description "nginx service" --target-tags myswarm
```

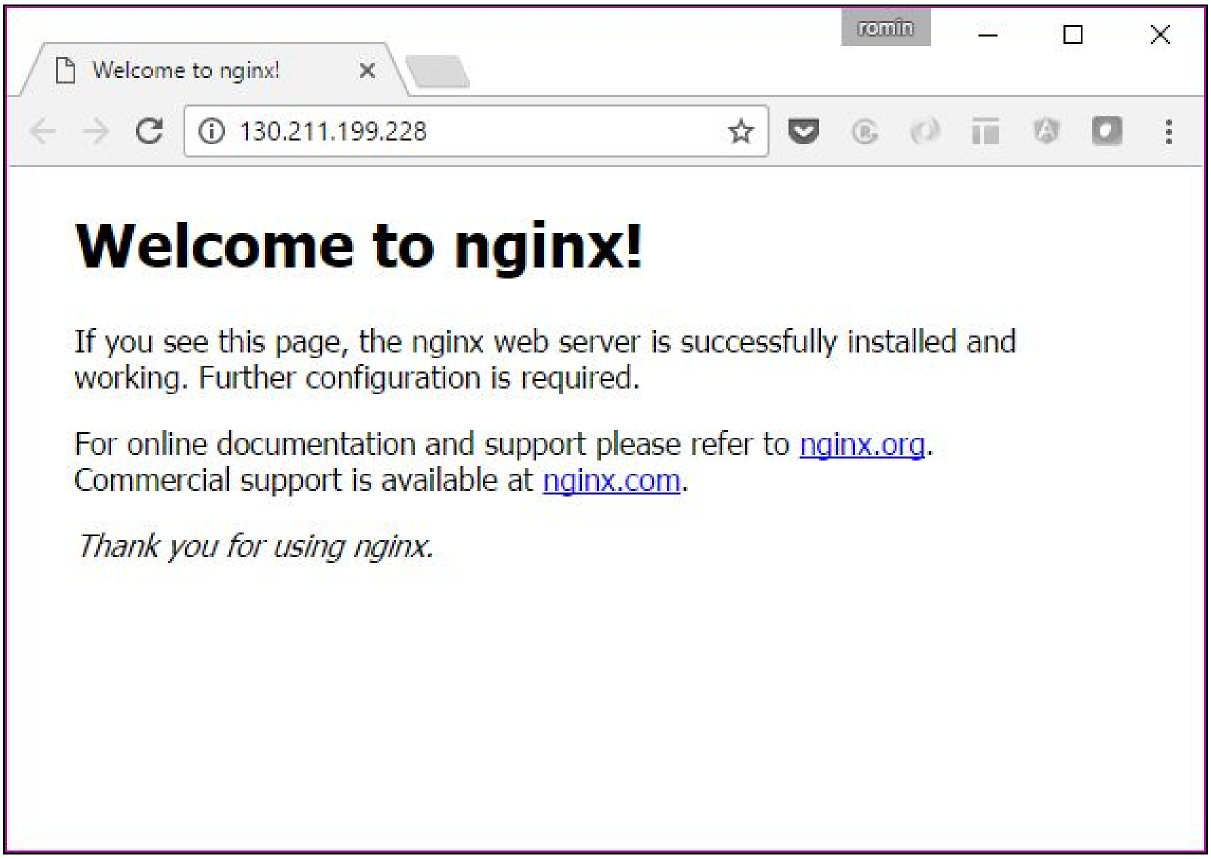
On successful creation, you will notice from your web Google Cloud console, that in the Networking → Firewalls list, you have an entry as shown below:

Name	Source tag / IP range / Subnetworks	Allowed protocols / ports	Target tags	Network
my-swarm-rule	0.0.0.0/0	tcp:80	myswarm	default

The details of which are shown below:



Now, if you hit any of the external IP Addresses from the Compute Engine instances list, you will get the NGINX home page as shown below, when I access one of the External IPs:



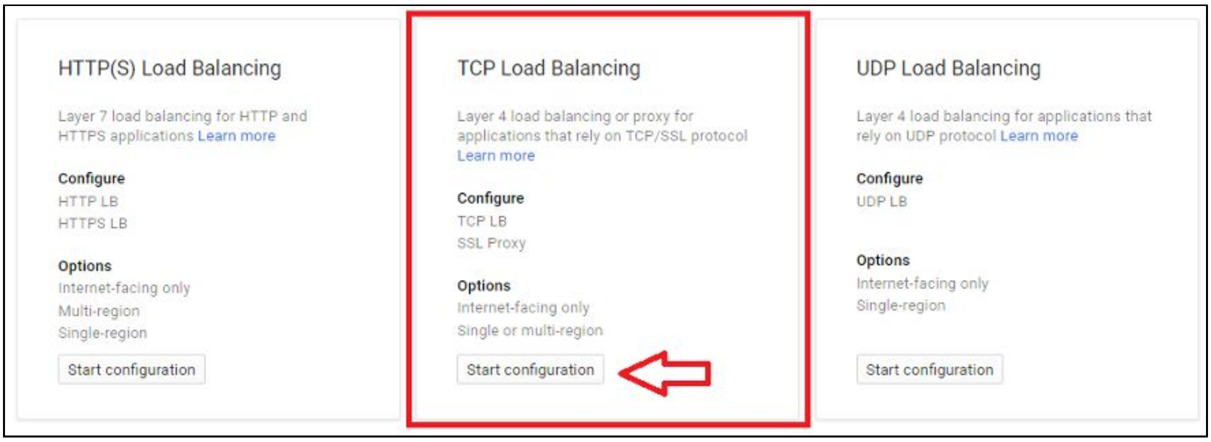
So, we are now able to access our service from any of the External IP addresses. But this is not what we want to do. We want to put these machines behind a Load Balancer. And simply by hitting the Load Balancer public IP, the traffic should get routed to any of the instances i.e. the nodes in our Docker Swarm Cluster.

For this, we need to create the Load Balancer, that Google Compute Engine provides.

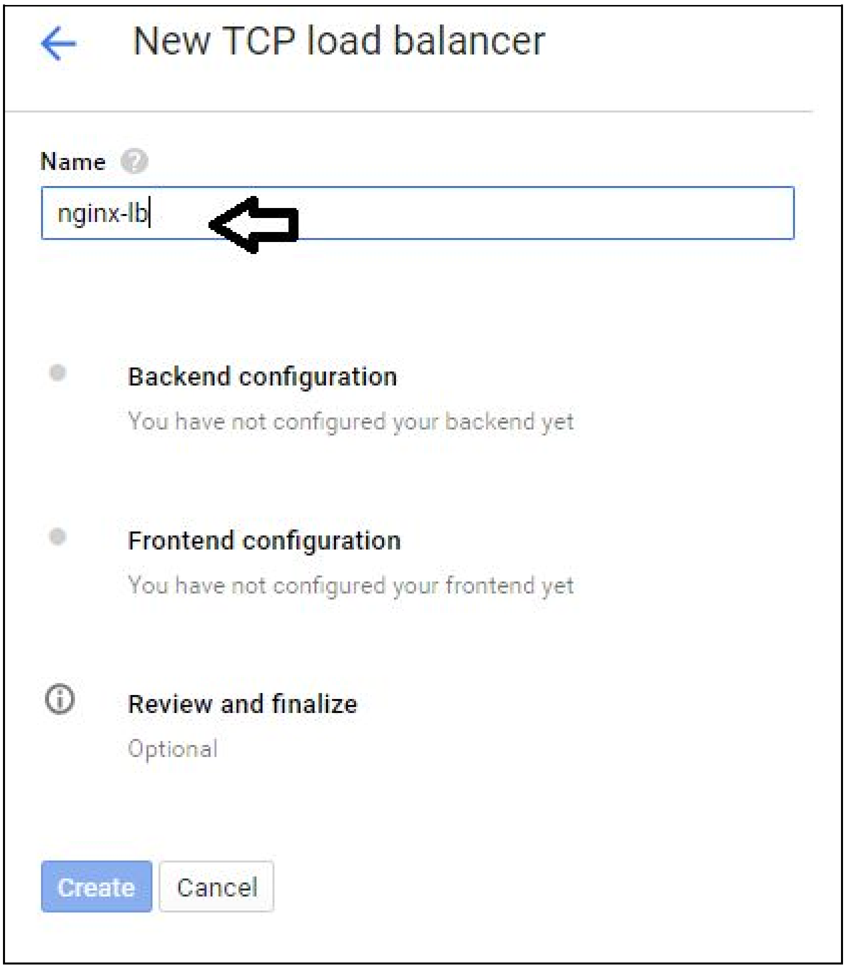
Creating the Load Balancer

To create the Load Balancer, do the following:

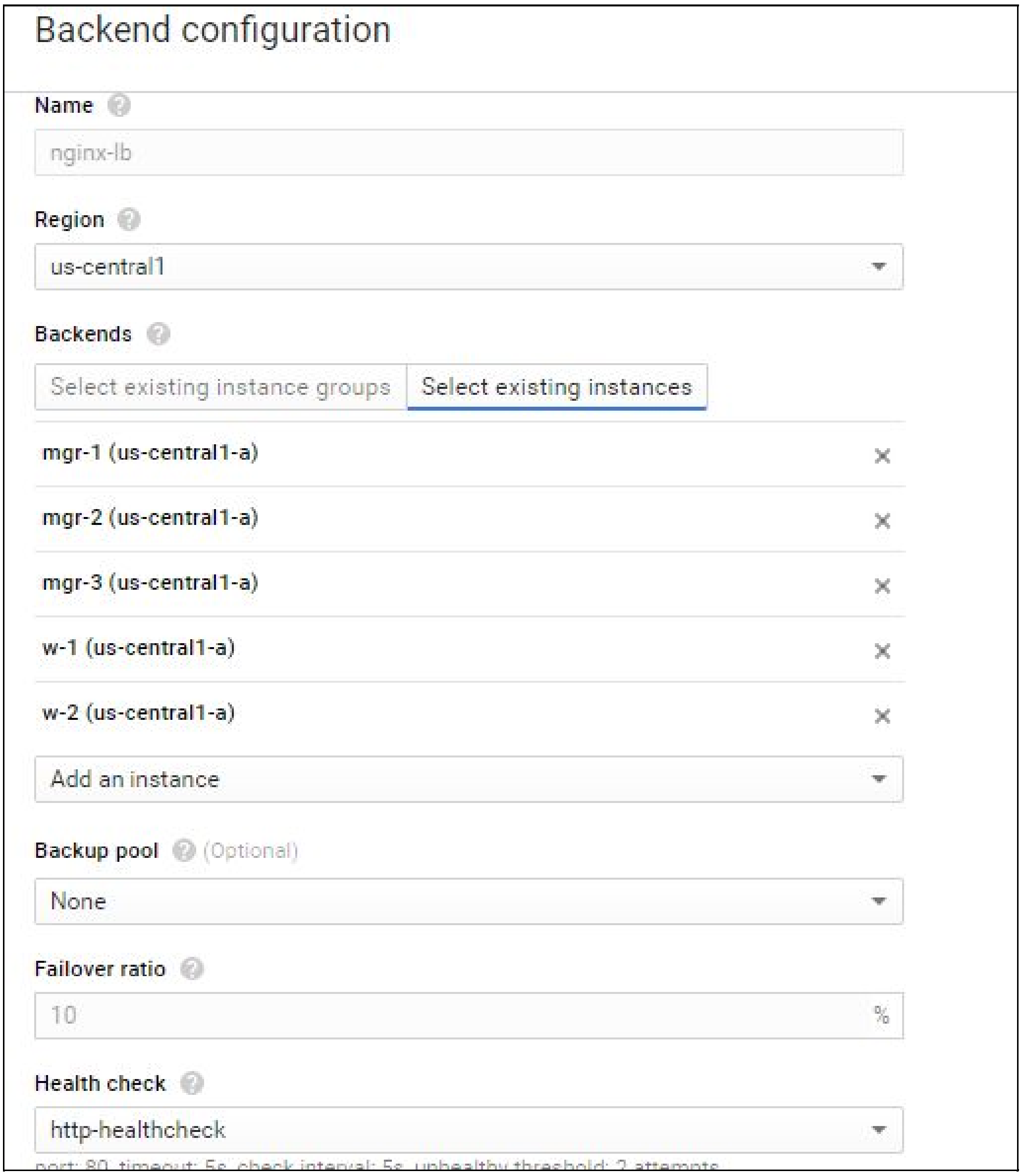
- Go to Google Cloud console.
- Go to Networking → Load Balancing. Click on Create Load Balancer button.
- You will see 3 options, click the Start configuration button in the TCP Load Balancing option as shown below:



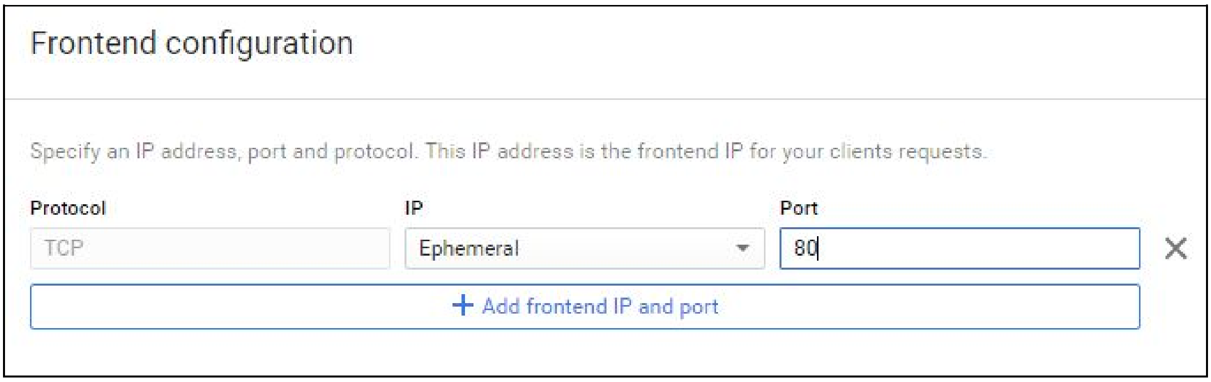
- Click on Continue in the next section and then you will reach the configuration for Backend and Frontend as shown below. Give your Load Balancer a name as shown below:



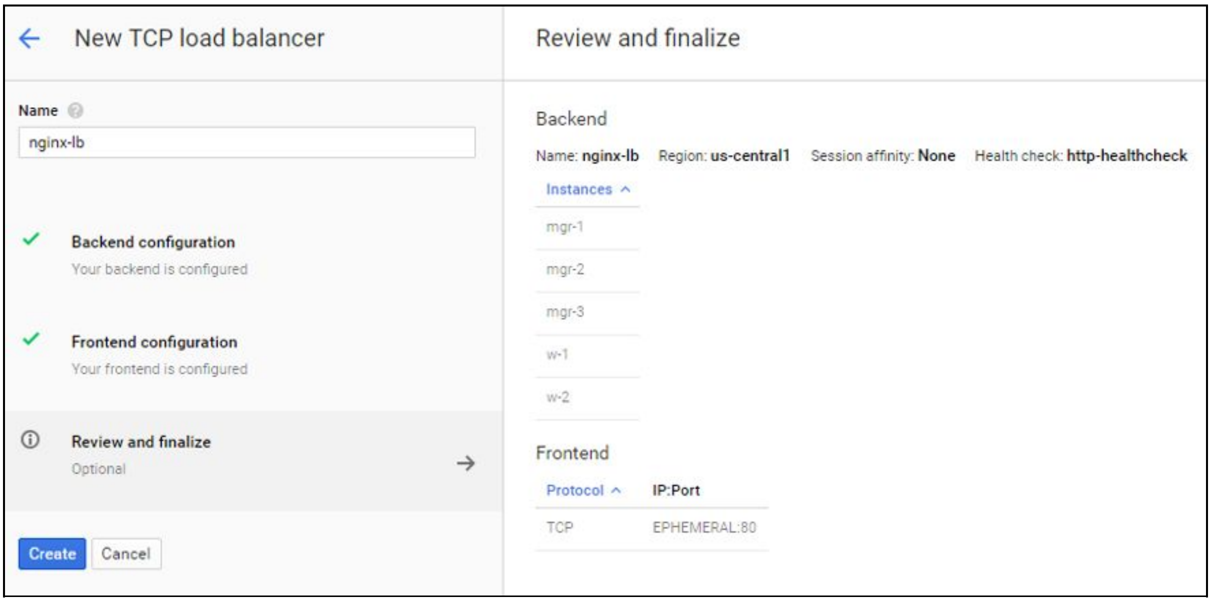
- Click on Backend configuration next. Select the region for your load balancer — I went with the ones in which I had my instances. Then select existing instances, and pick all the nodes (compute engine instances) that we created. Finally select a standard http port 80 healthcheck. This will enable the load balancer to check the health of each nodes. And since our nginx service is running on port 80, a simple healthcheck on port 80 is good enough for now.



- Now go to Front end configuration and enter the port as 80. We are going with an ephemeral ip for now, but you could also get yourself a Static IP Address.

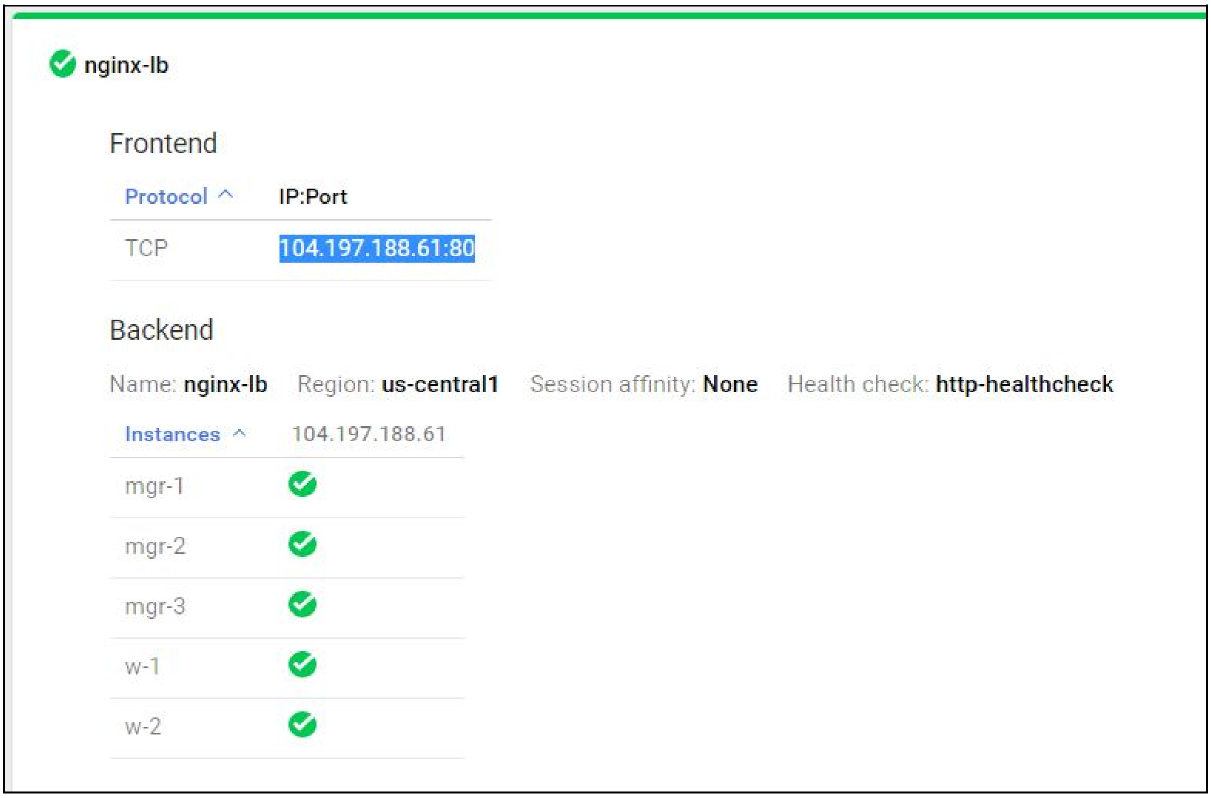


- Your review and finalize should look this:



Click on Create button to provision your Load Balancer. Give it some time.

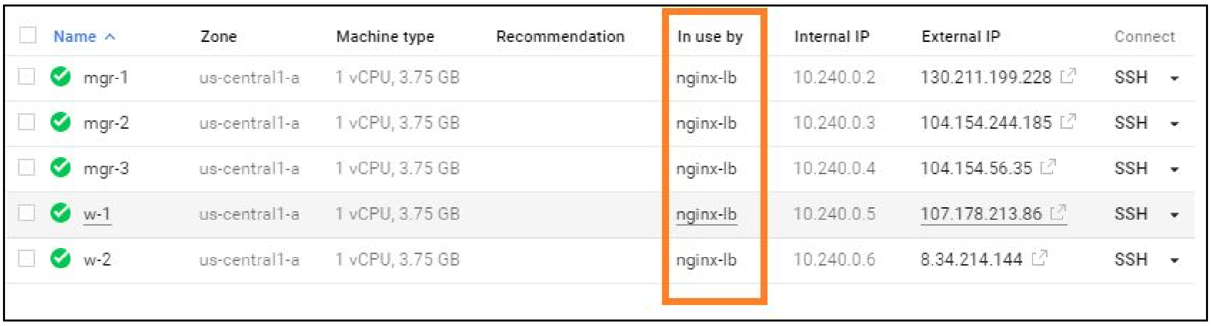
Once it is created, you can inspect it by clicking on the Load Balancer name. The details are shown below:



You can see that the Frontend part of it has been assigned an IP Address, which has been highlighted above. We can now hit this IP Address on port 80 and it will divert the traffic to be served by any of the healthy instances. All these instances are nothing but our nodes and since our NGINX service is running on these nodes on port 80, any of them will be able to serve it.

Go ahead, launch the browser and visit the Load Balancer IP in the browser. You should be fine:

Additionally, if you go to the list of Compute Engine instances, you will see that the nodes are now in use by our Load Balancer as shown below:



Name	Zone	Machine type	Recommendation	In use by	Internal IP	External IP	Connect
mgr-1	us-central1-a	1 vCPU, 3.75 GB		nginx-lb	10.240.0.2	130.211.199.228	SSH
mgr-2	us-central1-a	1 vCPU, 3.75 GB		nginx-lb	10.240.0.3	104.154.244.185	SSH
mgr-3	us-central1-a	1 vCPU, 3.75 GB		nginx-lb	10.240.0.4	104.154.56.35	SSH
w-1	us-central1-a	1 vCPU, 3.75 GB		nginx-lb	10.240.0.5	107.178.213.86	SSH
w-2	us-central1-a	1 vCPU, 3.75 GB		nginx-lb	10.240.0.6	8.34.214.144	SSH

We are looking good for now. Let us do a little deep dive into our overlay network and see what is going on.

Understand the overlay network

Let us go back to the list of networks that we have. We can do that from any node. I suggest that you SSH first to mgr-1 and do the following:

```
$ sudo docker network list
```

```
NETWORK ID          NAME                DRIVER              SCOPE
d4b360ee71b4        bridge              bridge              local
9a55643d34e8        docker_gwbridge     bridge              local
b6348ecb0afa        host                host                local
dzzo90eqcmt2        ingress             overlay             swarm
46f4630544d0        none                null                local
7ffh8lexsm9f        nw1                 overlay             swarm
```

Notice again that created an overlay network nw1 and its Network ID is7ffh8lexsm9f.

Let me also show you the output from the node listing in our cluster to understand where our 6 containers are currently running:

```
romin_irani@mgr-1:~$ sudo docker service ps nginx
ID    NAME     IMAGE  NODE   DESIRED STATE  CURRENT STATE
9z*   nginx.1  nginx  mgr-2  Running        Running 32 seconds ago
6e*   nginx.2  nginx  w-1    Running        Running 32 seconds ago
1o*   nginx.3  nginx  w-1    Running        Running 32 seconds ago
6n*   nginx.4  nginx  mgr-1  Running        Running 32 seconds ago
8l*   nginx.5  nginx  w-2    Running        Running 32 seconds ago
8p*   nginx.6  nginx  mgr-3  Running        Running 32 seconds ago
```

You can notice that there is only one container running on node mgr-1 and that it’s the nginx.4 container.

On the manager (mgr-1) node, you can now inspect the network by giving the following command:

```
romin_irani@mgr-1:~$ sudo docker network inspect nw1
[
   {
       "Name": "nw1",
       "Id": "7ffh8lexsm9fhiukslkyiml02",
       "Scope": "swarm",
       "Driver": "overlay",
       "EnableIPv6": false,
       "IPAM": {
           "Driver": "default",
           "Options": null,
           "Config": [
               {
                   "Subnet": "10.0.0.0/24",
                   "Gateway": "10.0.0.1"
               }
           ]
       },
       "Internal": false,
       "Containers": {
```

```
"3d32b7128776a8398e741ad542427f0edee0f165c3d8de66fd9b3777f6194a24": {
                "Name": "nginx.4.6nlj4cg74wxx76r6cumwrnfto",
                "EndpointID":
"2a99474d5e67433e7b0a371fb56225ec8d330a0af4fbe9d573b7c10e890292b3",
                "MacAddress": "02:42:0a:00:00:03",
                "IPv4Address": "10.0.0.3/24",
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.driver.overlay.vxlanid_list": "257"
        },
        "Labels": {}
    }
]
```

First notice the Network ID that is the same value as what we have seen in the network listing and that it is a swarm overlay network.

Now, notice the interesting part in the section for Containers. You will find that it is running one container, which is what we expect and you will see the same name i.e. nginx.4. What this means is that 1 container is bound to that overlay network.

Now, go to w-1 or any other worker node i.e. SSH into it. In our output, we have 2 containers (nginx.2 and nginx.3) running on that node, as highlighted below:

```
romin_irani@mgr-1:~$ sudo docker service ps nginx
ID    NAME     IMAGE  NODE   DESIRED STATE  CURRENT STATE
9z*   nginx.1  nginx  mgr-2  Running        Running 32 seconds ago
6e*   nginx.2  nginx  w-1    Running        Running 32 seconds ago
1o*   nginx.3  nginx  w-1    Running        Running 32 seconds ago
6n*   nginx.4  nginx  mgr-1  Running        Running 32 seconds ago
8l*   nginx.5  nginx  w-2    Running        Running 32 seconds ago
8p*   nginx.6  nginx  mgr-3  Running        Running 32 seconds ago
```

On w-1 node, if we inspect our overlay network, we will get the following output:

```
romin_irani@w-1:~$ sudo docker network inspect nw1
[
    {
        "Name": "nw1",
        "Id": "7ffh8lexsm9fhiukslkyiml02",
        "Scope": "swarm",
        "Driver": "overlay",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
```

```
            "Config": [
                {
                    "Subnet": "10.0.0.0/24",
                    "Gateway": "10.0.0.1"
                }
            ]
        },
        "Internal": false,
        "Containers": {

"89c37d465f8e3c064a796f24fd4fa82326e7b2fd0b364e64e1f8d2eddf23c84b": {
                "Name": "nginx.3.1oicz8rkfcggwh978b76t5ijg",
                "EndpointID":
"2433172095f9090b9502308be3069ffae2d1664a062ca85fe922f21b29dfe93a",
                "MacAddress": "02:42:0a:00:00:08",
                "IPv4Address": "10.0.0.8/24",
                "IPv6Address": ""
            },

"b3af0472d70e5c7210025ccbccb4432cb3c80e0ce2f907131ef4db6efb7c9d3e": {
                "Name": "nginx.2.6ezc0zmi6jbb3nmtjt0vpwpvt",
                "EndpointID":
"9b256f06c7f4d1b68f5592c7934a0f9770e7187dd49d5da6b033ea879b539cb0",
                "MacAddress": "02:42:0a:00:00:07",
                "IPv4Address": "10.0.0.7/24",
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.driver.overlay.vxlanid_list": "257"
        },
        "Labels": {}
    }
]
```

In the container list, you will find that 2 containers (nginx.2 and nginx.3) correctly bound to the network nw1.

Note: You should SSH on other nodes too and inspect the network from there too!

Now that we have inspected the network, let us understand that in an overlay network, we have a Virtual IP Address and a DNS name for each service by default. So in essence, when someone hits our service via the service name, it will resolve to a Virtual IP Address.

To understand that, we can inspect the service from the manager node. In the SSH session for mgr-1, inspect the nginx service as shown below:

```
romin_irani@mgr-1:~$ sudo docker service inspect nginx
[
    {
```

```
"ID": "1e68lm0x00zsqzdpje7nwql3j",
"Version": {
    "Index": 54
},
"CreatedAt": "2016-09-25T09:57:13.328195699Z",
"UpdatedAt": "2016-09-25T09:57:13.336401804Z",
"Spec": {
    "Name": "nginx",
    "TaskTemplate": {
        "ContainerSpec": {
            "Image": "nginx"
        },
        "Resources": {
            "Limits": {},
            "Reservations": {}
        },
        "RestartPolicy": {
            "Condition": "any",
            "MaxAttempts": 0
        },
        "Placement": {}
    },
    "Mode": {
        "Replicated": {
            "Replicas": 6
        }
    },
    "UpdateConfig": {
        "Parallelism": 1,
        "FailureAction": "pause"
    },
    "Networks": [
        {
            "Target": "7ffh8lexsm9fhiukslkyiml02"
        }
    ],
    "EndpointSpec": {
        "Mode": "vip",
        "Ports": [
            {
                "Protocol": "tcp",
                "TargetPort": 80,
                "PublishedPort": 80
            }
        ]
    }
},
"Endpoint": {
    "Spec": {
"UpdatedAt": "2016-09-25T09:57:13.336401804Z",
        "Mode": "vip",
```

```
                "Ports": [
                    {
                        "Protocol": "tcp",
                        "TargetPort": 80,
                        "PublishedPort": 80
                    }
                ]
            },
            "Ports": [
                {
                    "Protocol": "tcp",
                    "TargetPort": 80,
                    "PublishedPort": 80
                }
            ],
            "VirtualIPs": [
                {
                    "NetworkID": "dzzo90eqcmt2bvy1ygb59x0i3",
                    "Addr": "10.255.0.8/16"
                },
                {
                    "NetworkID": "7ffh8lexsm9fhiukslkyiml02",
                    "Addr": "10.0.0.2/24"
                }
            ]
        },
        "UpdateStatus": {
            "StartedAt": "0001-01-01T00:00:00Z",
            "CompletedAt": "0001-01-01T00:00:00Z"
        }
    }
]
```

Scroll down to the VirtualIPs section and you will notice that for the overlay network nw1, whose Network Id is “7fff….”, the associated address is 10.0.0.2 as shown above. So in short, the service name nginx resolves to that Virtual IP address, which in return will then hit any of the nodes servicing that request via the internal load balancing providing by Swarm.

We will see this at the end of the blog post, when we create another service, go into that container instance and then are able to lookup the nginx service by name. But before that, this section should suffice to tell you how it is constructed behind the scenes and in case you need to debug, you know how to go about it , one by one.

Let us first look at a few other features, just to test them out, so that we better understand what is going on in Docker Swarm.

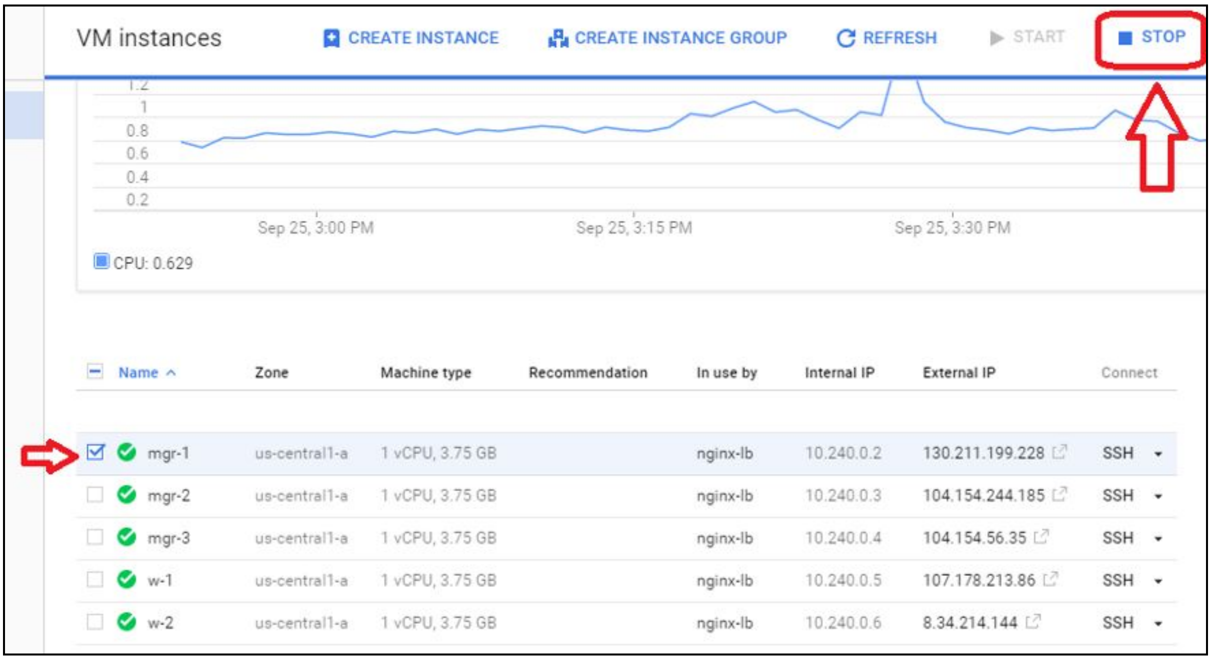
Bring the Leader down

The first test that we will try is to bring out Leader down. Let me list out the current set of nodes and their Status in our Docker Swarm cluster.

```
romin_irani@mgr-1:~$ sudo docker node ls
ID        HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
011..     mgr-2     Ready   Active        Reachable
4e3..     mgr-3     Ready   Active        Reachable
616.. *   mgr-1     Ready   Active        Leader
7xo..     w-1       Ready   Active
8ie..     w-2       Ready   Active
```

So, we have mgr-1 as the Leader and we have two other managers, who are reachable. What we expect is that if we bring mgr-1 down, then one of the other managers should take over as Leader. I suggest you also read up on Raft Consensus protocol to understand how the negotiation could take place and what would be some constraints on the number of managers and workers that you need have consensus from.

So, I am going to go and stop the current running instance on Google Compute Engine. You can do that from the Web console as shown below:



Wait till it has stopped. Now we can SSH into mgr-2 instance and see what is going on:

```
romin_irani@mgr-2:~$ sudo docker node ls
ID       HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
01.. *   mgr-2     Ready   Active        Leader
4e..     mgr-3     Ready   Active        Reachable
61..     mgr-1     Down    Active        Unreachable
```

```
7x..     w-1        Ready   Active
8i..     w-2        Ready   Active
```

It is interesting to see that mgr-2 became the Leader. If you hit the Load Balancer IP, everything is still working fine.

Let us look at what happened to our existing container instances. Remember that we had mentioned that we want 6 replicas of the nginx service. And if you recollect, one container was running on mgr-1.

```
romin_irani@mgr-2:~$ sudo docker service ps nginx
ID    NAME          IMAGE  NODE   DESIRED STATE  CURRENT STATE
9z..  nginx.1       nginx  mgr-2  Running        Running 29 min ago
6e..  nginx.2       nginx  w-1    Running        Running 29 min ago
1o..  nginx.3       nginx  w-1    Running        Running 29 min ago
1s..  nginx.4       nginx  mgr-2  Running        Running 3 min ago
6n..   \_ nginx.4   nginx  mgr-1  Shutdown       Running 29 min ago
8l..  nginx.5       nginx  w-2    Running        Running 29 min ago
8p..  nginx.6       nginx  mgr-3  Running        Running 29 min ago
```

You will notice that the container nginx.4 which was running on mgr-1 was taken down and relaunched on mgr-2. Looks good!

Bringing the original Leader back up again

What happens if we bring mgr-1 back up again. Will it take over as the Leader again, since it was the original leader or will be be a Manager node but cannot become a leader just by coming up again.

It is straightforward to try this. Simply go to the Cloud console and restart the instance. Wait till the instance is powered on and running:

If you are still in the SSH session on mgr-2, you can try:

```
romin_irani@mgr-2:~$ sudo docker node ls
ID       HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
01.. *   mgr-2     Ready   Active        Leader
4e..     mgr-3     Ready   Active        Reachable
6l..     mgr-1     Ready   Active        Reachable
7x..     w-1       Ready   Active
8i..     w-2       Ready   Active
```

You find that mgr-2 is still the Leader. mgr-1 is now Reachable but did not get instantly promoted to be a Leader.

What about our 6 containers, would some of them get relaunched on mgr-1, just because it came up. Let's see:

```
romin_irani@mgr-2:~$ sudo docker service ps nginx
ID     NAME          IMAGE  NODE   DESIRED STATE  CURRENT STATE
9z..   nginx.1       nginx  mgr-2  Running        Running 35 min ago
6e..   nginx.2       nginx  w-1    Running        Running 35 min ago
1o..   nginx.3       nginx  w-1    Running        Running 35 min ago
1s..   nginx.4       nginx  mgr-2  Running        Running 9 min ago
6n..    \_ nginx.4   nginx  mgr-1  Shutdown       Running 35 min ago
8l..   nginx.5       nginx  w-2    Running        Running 35 min ago
8p..   nginx.6       nginx  mgr-3  Running        Running 35 min ago
```

Well, it did not! Docker Swarm does not assign containers to newly joined nodes unless the service is scaled or some other nodes are drained and so on. On how to scale, you can follow the [Docker Swarm Tutorial] that I earlier wrote.

Note: You can try scaling the service up by a few more replicas and see what happens. Try it as an exercise. Hint --> $ docker service scale nginx=8

Bringing a Node down and backup

This should be straightforward to predict and try out. I will leave it as an exercise for the reader. Just stop w-1 node and then check on the status of the nodes in the swarm and also how it relaunches containers on the other remaining RUNNING nodes.

Do keep in mind that as we saw earlier, bringing up the node, does not mean that it will immediately get assigned some containers.

Creating another Service

It is time now to see how the overlay network is working. To reiterate, the overlay network allows containers across multiple hosts to communicate to each other. What this means is that you should be able to simply access any service by its name in any of the containers on the same overlay network.

By referring to the service by its name, it also allows us to scale the number of containers up and down, make them join the swarm and still keep accessing them via a uniform service name.

I am going to use the example from the Docker Swarm overlay network service documentation and use it over here.

First up, we will create a new Docker Swarm service. And then from the containers running this new service, we will see that we can access the service by name.
SSH in mgr-1 or mgr-2 instance. And create the new service as shown below. Note that we are going to use the same overlay network nw1.

```
romin_irani@mgr-1:~$ sudo docker service create --name my-busybox
--network nw1 busybox sleep 3000
azeevpytjcwsfoyvuv2pj4vdu

romin_irani@mgr-1:~$ sudo docker service ls
ID            NAME        REPLICAS  IMAGE    COMMAND
1e68lm0x00zs  nginx       6/6       nginx    azeevpytjcws
my-busybox  1/1       busybox  sleep 3000

romin_irani@mgr-1:~$ sudo docker service ps my-busybox
ID                         NAME          IMAGE    NODE   DESIRED
STATE  CURRENT STATE           ERROR
4elkoyujbyausx7zi5u5i7999  my-busybox.1  busybox  mgr-1  Running
Running 21 seconds ago
```

You will notice that in the first command, we are starting up a busybox service named my-busybox, we want only one replica of it, we are using the same overlay network nw1.

Notice that we gave a delay of 3000s so that the container is alive for a while before shutting down because on its own the busybox container will just exit otherwise.

Great! The next command that you see above is the standard service listing and you can see that it has 2 services now: nginx and my-busybox service.

Similarly, the last command is to find out where the my-busybox service containers are running. We find that it is running on mgr-1.

Now, let us get into the Bash shell for the running container for the my-busybox service.

```
romin_irani@mgr-1:~$ sudo docker service ps my-busybox
ID                         NAME          IMAGE    NODE   DESIRED
STATE  CURRENT STATE           ERROR
4elkoyujbyausx7zi5u5i7999  my-busybox.1  busybox  mgr-1  Running
Running 21 seconds ago
```

Notice that we have the following attributes:

- NAME is my-busybox.1
- ID is 4elkoyujbyausx7zi5u5i7999

In summary, you can form a unique name for the Container as NAME.ID

To go into the bash shell for this container, execute the following command:

```
romin_irani@mgr-1:~$ sudo docker exec -it
my-busybox.1.4elkoyujbyausx7zi5u5i7999 /bin/sh
/ #
/ #
/ #
```

Now inside this shell, we can do a lookup for our service nginx by name.

```
/ #
/ #
/ # nslookup nginx
Server: 127.0.0.11
Address 1: 127.0.0.11
Name: nginx
Address 1: 10.0.0.2
/ #
/ #
```

You can even do a wget inside over here to validate that we are able to hit the service and get the NGINX default home page:

```
/ # wget -O - nginx
Connecting to nginx (10.0.0.2:80)
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
   body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
   }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>
```

```
<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>
<p><em>Thank you for using nginx.</em></p>
</body>
</html>
-                    100%
|*******************************************************************|   612
0:00:00 ETA
/ #
```

This shows that we now have a network where each of the services are available to all containers on the network. This way you can link up multiple containers as needed.

Due to the fact that we have a service abstraction now, you can scale your nodes — add / remove them — and not affect the containers that are accessing it by service name. They will not be worried about where the containers are running i.e. on which nodes.

Conclusion

I like the simplicity of Docker Swarm and conducting these experiments gave me a good sense of understanding how it is working behind the scenes, what to expect and most importantly, to actually see it work.

Please let me know in the comments if you have any feedback. Reach out to me at romin.k.irani@gmail.com