Networking Between Container Hosts

## Destroy the existing Swarm instance

Networking in Docker 1.9 requires the use of Consul as the swarm dependency. So we are going to create a new Swarm instance for our hosts.

That means we will get rid of the old Swarm from lab3:

```

docker-machine kill swarm-agent-00

docker-machine rm swarm-agent-00

docker-machine kill swarm-agent-01

docker-machine rm swarm-agent-01

docker-machine kill swarm-master

docker-machine rm swarm-master

```

## Create A New Swarm Instance with Consul

Next, we need to add a few Docker Engine options which are necessary for networking to function. Perform these functions on:

```

$ sudo service docker stop

$ sudo sed -i '10 a --cluster-advertise=eth0:2376' /etc/default/docker

$ sudo sed -i '10 a --cluster-store=consul://<host-a-public-IP>:8500' /etc/default/docker

$ sudo service docker start

```

On host `a` start a consul server:

```

$ docker run -d -p 8500:8500 -h consul progrium/consul -server -bootstrap-expect 1

```

Now we can configure our new Swarm agents on each host `a` and `b`:

```

$ docker run -d --restart=always --name swarm-agent-consul swarm:1.0.0 join --advertise $(curl http://169.254.169.254/latest/meta-data/public-ipv4):2376 consul://<host-a-public-IP>:8500

```

Start swarm manager on `a`

```

$ docker run -d --restart=always --name swarm-agent-master-consul -p 3376:3376 -v /etc/docker:/etc/docker swarm:1.0.0 manage --tlsverify --tlscacert=/etc/docker/ca.pem --tlscert=/etc/docker/server.pem --tlskey=/etc/docker/server-key.pem -H tcp://0.0.0.0:3376 --strategy spread consul://<host-a-public-IP>:8500

```

On Host `a` point the Docker CLI to the new Swarm instance:

```

$ export DOCKER\_TLS\_VERIFY=1

$ export DOCKER\_HOST=tcp://$(curl http://169.254.169.254/latest/meta-data/public-ipv4):3376

```

Performing a `docker info` should give you a Swarm output:

```

student001a@student001a:~$ docker info

Containers: 4

Images: 4

Role: primary

Strategy: spread

Filters: affinity, health, constraint, port, dependency

Nodes: 2

student001a: 52.91.204.239:2376

└ Containers: 2

└ Reserved CPUs: 0 / 1

└ Reserved Memory: 0 B / 1.017 GiB

└ Labels: executiondriver=native-0.2, kernelversion=3.19.0-30-generic, operatingsystem=Ubuntu 14.04.3 LTS, provider=amazonec2, storagedriver=aufs

student001b: 54.86.251.223:2376

└ Containers: 2

└ Reserved CPUs: 0 / 1

└ Reserved Memory: 0 B / 1.017 GiB

└ Labels: executiondriver=native-0.2, kernelversion=3.19.0-30-generic, operatingsystem=Ubuntu 14.04.3 LTS, provider=amazonec2, storagedriver=aufs

CPUs: 2

Total Memory: 2.033 GiB

Name: 33b92fe60f6e

```

## Network the Containers

Create a new overlay network which containers can join. `multihost` is an arbitrary name and can be called whatever you want:

```

docker network create -d overlay multihost

```

Create our first container and using a constraint, pin it to host `a`. This will leave a host idle with `top` running.

```

docker run -d --name="long-running" --net="multihost" --env="constraint:node==\*a\*" busybox top

```

Pin the second container to host `b` and ping `long-running` container:

```

docker run -it --rm --net="multihost" --env="constraint:node==\*b\*" busybox ping long-running

```

## Understanding Networking Further

To get a better grasp on the networking, lets create another network and spin up an Ubuntu container shell:

```

$ docker network create -d overlay testnetwork

$ docker run -i -t --net="testnetwork" ubuntu /bin/bash

```

From here we can do an `ifconfig` and see that the `eth0` adapter is given a private IP address on the `10.0.1.0/24` network. This will prove that networking can be very extensive within a single Docker Swarm instance.

```

student001a@student001a:~/compose$ docker run -i -t --net="multi" ubuntu /bin/bash

root@3cdff88af090:/# ifconfig

eth0 Link encap:Ethernet HWaddr 02:42:0a:00:01:02

inet addr:10.0.1.2 Bcast:0.0.0.0 Mask:255.255.255.0

inet6 addr: fe80::42:aff:fe00:102/64 Scope:Link

UP BROADCAST RUNNING MULTICAST MTU:1450 Metric:1

RX packets:11 errors:0 dropped:0 overruns:0 frame:0

TX packets:6 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:0

RX bytes:926 (926.0 B) TX bytes:508 (508.0 B)

eth1 Link encap:Ethernet HWaddr 02:42:ac:12:00:02

inet addr:172.18.0.2 Bcast:0.0.0.0 Mask:255.255.0.0

inet6 addr: fe80::42:acff:fe12:2/64 Scope:Link

UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1

RX packets:6 errors:0 dropped:0 overruns:0 frame:0

TX packets:6 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:0

RX bytes:508 (508.0 B) TX bytes:508 (508.0 B)

lo Link encap:Local Loopback

inet addr:127.0.0.1 Mask:255.0.0.0

inet6 addr: ::1/128 Scope:Host

UP LOOPBACK RUNNING MTU:65536 Metric:1

RX packets:0 errors:0 dropped:0 overruns:0 frame:0

TX packets:0 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:0

RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

```

## Bringing it all together

The combination of persistent storage and overlay networks brings some exciting new ground to Docker use cases. In this section, we will quickly see how we can move containers and still persist data.

On host `a`:

```

$ docker network create -d overlay netstore

$ docker run -d --name="long-running-persist" --net="netstore" --volume-driver=rexray -v test3:/test --env="constraint:node==\*a\*" busybox top

$ docker run -it --rm --net="netstore" --env="constraint:node==\*b\*" busybox ping long-running-persist

```

As expected, we should see pings start hitting with containers on different hosts.

Now perform the following from 'a':

```

$ docker exec long-running-persist touch /test/persist

$ docker stop long-running-persist

$ docker rm long-running-persist

```

Let's start the container again with a few different flags, this time so we can see the file persistence. on host `a`:

```

$ docker run -tid --name="long-running-persist" --net="netstore" --volume-driver=rexray -v test3:/test --env="constraint:node==\*b\*" busybox

```

Now the persisitent container is running from `b`, next thing is to ensure the files are present and it is still able to ping:

```

$ docker exec long-running-persist ls /test

$ docker run -it --rm --net="netstore" --env="constraint:node==\*b\*" busybox ping long-running-persist

```

and we will see our files there. Awesome!

Clean-up:

```

$ docker stop long-running-persist

$ docker volume rm test3

```