The Docker Network Command

The docker network command is the main command for configuring and managing container networks. Run the docker network command from the first terminal.

docker network

Usage: docker network COMMAND

Manage networks

Options:

--help Print usage

Commands:

connect Connect a container to a network

create Create a network

disconnect Disconnect a container from a network

inspect Display detailed information on one or more networks

ls List networks

prune Remove all unused networks

rm Remove one or more networks

Run 'docker network COMMAND --help' for more information on a command.

The command output shows how to use the command as well as all of the docker network sub-commands. As you can see from the output, the docker network command allows you to create new networks, list existing networks, inspect networks, and remove networks. It also allows you to connect and disconnect containers from networks.

Step 2: List networks

Run a docker network ls command to view existing container networks on the current Docker host.

docker network ls

NETWORK ID NAME DRIVER SCOPE

3430ad6f20bf bridge bridge local

a7449465c379 host host local

06c349b9cc77 none null local

The output above shows the container networks that are created as part of a standard installation of Docker.

New networks that you create will also show up in the output of the docker network ls command.

You can see that each network gets a unique ID and NAME. Each network is also associated with a single driver. Notice that the “bridge” network and the “host” network have the same name as their respective drivers.

Step 3: Inspect a network

The docker network inspect command is used to view network configuration details. These details include; name, ID, driver, IPAM driver, subnet info, connected containers, and more.

Use docker network inspect <network> to view configuration details of the container networks on your Docker host. The command below shows the details of the network called bridge.

docker network inspect bridge

[

{

"Name": "bridge",

"Id": "3430ad6f20bf1486df2e5f64ddc93cc4ff95d81f59b6baea8a510ad500df2e57",

"Created": "2017-04-03T16:49:58.6536278Z",

"Scope": "local",

"Driver": "bridge",

"EnableIPv6": false,

"IPAM": {

"Driver": "default",

"Options": null,

"Config": [

{

"Subnet": "172.17.0.0/16",

"Gateway": "172.17.0.1"

}

]

},

"Internal": false,

"Attachable": false,

"Containers": {},

"Options": {

"com.docker.network.bridge.default\_bridge": "true",

"com.docker.network.bridge.enable\_icc": "true",

"com.docker.network.bridge.enable\_ip\_masquerade": "true",

"com.docker.network.bridge.host\_binding\_ipv4": "0.0.0.0",

"com.docker.network.bridge.name": "docker0",

"com.docker.network.driver.mtu": "1500"

},

"Labels": {}

}

]

**NOTE:** The syntax of the docker network inspect command is docker network inspect <network>, where <network> can be either network name or network ID. In the example above we are showing the configuration details for the network called “bridge”. Do not confuse this with the “bridge” driver.

Step 4: List network driver plugins

The docker info command shows a lot of interesting information about a Docker installation.

Run the docker info command and locate the list of network plugins.

docker info

Containers: 0

Running: 0

Paused: 0

Stopped: 0

Images: 0

Server Version: 17.03.1-ee-3

Storage Driver: aufs

<Snip>

Plugins:

Volume: local

Network: bridge host macvlan null overlay

Swarm: inactive

Runtimes: runc

<Snip>

The output above shows the **bridge**, **host**,**macvlan**, **null**, and **overlay** drivers.

Section #2 - Bridge Networking

Step 1: The Basics

Every clean installation of Docker comes with a pre-built network called **bridge**. Verify this with the docker network ls.

docker network ls

NETWORK ID NAME DRIVER SCOPE

3430ad6f20bf bridge bridge local

a7449465c379 host host local

06c349b9cc77 none null local

The output above shows that the **bridge** network is associated with the *bridge* driver. It’s important to note that the network and the driver are connected, but they are not the same. In this example the network and the driver have the same name - but they are not the same thing!

The output above also shows that the **bridge** network is scoped locally. This means that the network only exists on this Docker host. This is true of all networks using the *bridge* driver - the *bridge* driver provides single-host networking.

All networks created with the *bridge* driver are based on a Linux bridge (a.k.a. a virtual switch).

Install the brctl command and use it to list the Linux bridges on your Docker host. You can do this by running sudo apt-get install bridge-utils.

apk update

apk add bridge

Then, list the bridges on your Docker host, by running brctl show.

brctl show

bridge name bridge id STP enabled interfaces

docker0 8000.024252ed52f7 no

The output above shows a single Linux bridge called **docker0**. This is the bridge that was automatically created for the **bridge** network. You can see that it has no interfaces currently connected to it.

You can also use the ip a command to view details of the **docker0** bridge.

ip a

<Snip>

3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default

link/ether 02:42:52:ed:52:f7 brd ff:ff:ff:ff:ff:ff

inet 172.17.0.1/16 scope global docker0

valid\_lft forever preferred\_lft forever

Step 2: Connect a container

The **bridge** network is the default network for new containers. This means that unless you specify a different network, all new containers will be connected to the **bridge** network.

Create a new container by running docker run -dt ubuntu sleep infinity.

docker run -dt ubuntu sleep infinity

Unable to find image 'ubuntu:latest' locally

latest: Pulling from library/ubuntu

d54efb8db41d: Pull complete

f8b845f45a87: Pull complete

e8db7bf7c39f: Pull complete

9654c40e9079: Pull complete

6d9ef359eaaa: Pull complete

Digest: sha256:dd7808d8792c9841d0b460122f1acf0a2dd1f56404f8d1e56298048885e45535

Status: Downloaded newer image for ubuntu:latest

846af8479944d406843c90a39cba68373c619d1feaa932719260a5f5afddbf71

This command will create a new container based on the ubuntu:latest image and will run the sleep command to keep the container running in the background. You can verify our example container is up by running docker ps.

docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

846af8479944 ubuntu "sleep infinity" 55 seconds ago Up 54 seconds heuristic\_boyd

As no network was specified on the docker run command, the container will be added to the **bridge** network.

Run the brctl show command again.

brctl show

bridge name bridge id STP enabled interfaces

docker0 8000.024252ed52f7 no vethd630437

Notice how the **docker0** bridge now has an interface connected. This interface connects the **docker0** bridge to the new container just created.

You can inspect the **bridge** network again, by running docker network inspect bridge, to see the new container attached to it.

docker network inspect bridge

<Snip>

"Containers": {

"846af8479944d406843c90a39cba68373c619d1feaa932719260a5f5afddbf71": {

"Name": "heuristic\_boyd",

"EndpointID": "1265c418f0b812004d80336bafdc4437eda976f166c11dbcc97d365b2bfa91e5",

"MacAddress": "02:42:ac:11:00:02",

"IPv4Address": "172.17.0.2/16",

"IPv6Address": ""

}

},

<Snip>

Step 3: Test network connectivity

The output to the previous docker network inspect command shows the IP address of the new container. In the previous example it is “172.17.0.2” but yours might be different.

Ping the IP address of the container from the shell prompt of your Docker host by running ping -c5 <IPv4 Address>. Remember to use the IP of the container in **your** environment.

ping -c5 172.17.0.2

PING 172.17.0.2 (172.17.0.2) 56(84) bytes of data.

64 bytes from 172.17.0.2: icmp\_seq=1 ttl=64 time=0.055 ms

64 bytes from 172.17.0.2: icmp\_seq=2 ttl=64 time=0.031 ms

64 bytes from 172.17.0.2: icmp\_seq=3 ttl=64 time=0.034 ms

64 bytes from 172.17.0.2: icmp\_seq=4 ttl=64 time=0.041 ms

64 bytes from 172.17.0.2: icmp\_seq=5 ttl=64 time=0.047 ms

--- 172.17.0.2 ping statistics ---

5 packets transmitted, 5 received, 0% packet loss, time 4075ms

rtt min/avg/max/mdev = 0.031/0.041/0.055/0.011 ms

The replies above show that the Docker host can ping the container over the **bridge** network. But, we can also verify the container can connect to the outside world too. Lets log into the container, install the ping program, and then ping www.github.com.

First, we need to get the ID of the container started in the previous step. You can run docker ps to get that.

docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

846af8479944 ubuntu "sleep infinity" 7 minutes ago Up 7 minutes heuristic\_boyd

Next, lets run a shell inside that ubuntu container, by running docker exec -it <CONTAINER ID> /bin/bash.

docker exec -it yourcontainerid /bin/bash

root@846af8479944:/#

Next, we need to install the ping program. So, lets run apt-get update && apt-get install -y iputils-ping.

apt-get update && apt-get install -y iputils-ping

Lets ping www.github.com by running ping -c5 www.github.com

ping -c5 www.github.com

PING www.docker.com (104.239.220.248) 56(84) bytes of data.

64 bytes from 104.239.220.248: icmp\_seq=1 ttl=45 time=38.1 ms

64 bytes from 104.239.220.248: icmp\_seq=2 ttl=45 time=37.3 ms

64 bytes from 104.239.220.248: icmp\_seq=3 ttl=45 time=37.5 ms

64 bytes from 104.239.220.248: icmp\_seq=4 ttl=45 time=37.5 ms

64 bytes from 104.239.220.248: icmp\_seq=5 ttl=45 time=37.5 ms

--- www.docker.com ping statistics ---

5 packets transmitted, 5 received, 0% packet loss, time 4003ms

rtt min/avg/max/mdev = 37.372/37.641/38.143/0.314 ms

Finally, lets disconnect our shell from the container, by running exit.

exit

We should also stop this container so we clean things up from this test, by running docker stop <CONTAINER ID>.

docker stop yourcontainerid

This shows that the new container can ping the internet and therefore has a valid and working network configuration.

Step 4: Configure NAT for external connectivity

In this step we’ll start a new **NGINX** container and map port 8080 on the Docker host to port 80 inside of the container. This means that traffic that hits the Docker host on port 8080 will be passed on to port 80 inside the container.

**NOTE:** If you start a new container from the official NGINX image without specifying a command to run, the container will run a basic web server on port 80.

Start a new container based off the official NGINX image by running docker run --name web1 -d -p 8080:80 nginx.

docker run --name web1 -d -p 8080:80 nginx

Unable to find image 'nginx:latest' locally

latest: Pulling from library/nginx

6d827a3ef358: Pull complete

b556b18c7952: Pull complete

03558b976e24: Pull complete

9abee7e1ef9d: Pull complete

Digest: sha256:52f84ace6ea43f2f58937e5f9fc562e99ad6876e82b99d171916c1ece587c188

Status: Downloaded newer image for nginx:latest

4e0da45b0f169f18b0e1ee9bf779500cb0f756402c0a0821d55565f162741b3e

Review the container status and port mappings by running docker ps.

docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

4e0da45b0f16 nginx "nginx -g 'daemon ..." 2 minutes ago Up 2 minutes 443/tcp, 0.0.0.0:8080->80/tcp web1

The top line shows the new **web1** container running NGINX. Take note of the command the container is running as well as the port mapping - 0.0.0.0:8080->80/tcp maps port 8080 on all host interfaces to port 80 inside the **web1** container. This port mapping is what effectively makes the containers web service accessible from external sources (via the Docker hosts IP address on port 8080).

Now that the container is running and mapped to a port on a host interface you can test connectivity to the NGINX web server.

To complete the following task you will need the IP address of your Docker host. This will need to be an IP address that you can reach (e.g. your lab is hosted in Azure so this will be the instance’s Public IP - the one you SSH’d into). Just point your web browser to the IP and port 8080 of your Docker host. Also, if you try connecting to the same IP address on a different port number it will fail.

If for some reason you cannot open a session from a web broswer, you can connect from your Docker host using the curl 127.0.0.1:8080 command.

curl 127.0.0.1:8080

**<!DOCTYPE html>**

<html>

<Snip>

<head>

<title>Welcome to nginx!</title>

<Snip>

<p><em>Thank you for using nginx.</em></p>

</body>

</html>

If you try and curl the IP address on a different port number it will fail.

**NOTE:** The port mapping is actually port address translation (PAT).

Section #3 - Overlay Networking

Step 1: The Basics

In this step you’ll initialize a new Swarm, join a single worker node, and verify the operations worked.

Run docker swarm init --advertise-addr $(hostname -i).

docker swarm init --advertise-addr $(hostname -i)

Swarm initialized: current node (rzyy572arjko2w0j82zvjkc6u) is now a manager.

To add a worker to this swarm, run the following command:

docker swarm join \

--token SWMTKN-1-69b2x1u2wtjdmot0oqxjw1r2d27f0lbmhfxhvj83chln1l6es5-37ykdpul0vylenefe2439cqpf \

10.0.0.5:2377

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

In the first terminal copy the entire docker swarm join ... command that is displayed as part of the output from your terminal output. Then, paste the copied command into the second terminal.

docker swarm join \

> --token SWMTKN-1-69b2x1u2wtjdmot0oqxjw1r2d27f0lbmhfxhvj83chln1l6es5-37ykdpul0vylenefe2439cqpf \

> 10.0.0.5:2377

This node joined a swarm as a worker.

Run a docker node ls to verify that both nodes are part of the Swarm.

docker node ls

ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS

ijjmqthkdya65h9rjzyngdn48 node2 Ready Active

rzyy572arjko2w0j82zvjkc6u \* node1 Ready Active Leader

The ID and HOSTNAME values may be different in your lab. The important thing to check is that both nodes have joined the Swarm and are *ready* and *active*.

Step 2: Create an overlay network

Now that you have a Swarm initialized it’s time to create an **overlay** network.

Create a new overlay network called “overnet” by running docker network create -d overlay overnet.

docker network create -d overlay overnet

wlqnvajmmzskn84bqbdi1ytuy

Use the docker network ls command to verify the network was created successfully.

docker network ls

NETWORK ID NAME DRIVER SCOPE

3430ad6f20bf bridge bridge local

a4d584350f09 docker\_gwbridge bridge local

a7449465c379 host host local

8hq1n8nak54x ingress overlay swarm

06c349b9cc77 none null local

wlqnvajmmzsk overnet overlay swarm

The new “overnet” network is shown on the last line of the output above. Notice how it is associated with the **overlay** driver and is scoped to the entire Swarm.

**NOTE:** The other new networks (ingress and docker\_gwbridge) were created automatically when the Swarm cluster was created.

Run the same docker network ls command from the second terminal.

docker network ls

NETWORK ID NAME DRIVER SCOPE

55f10b3fb8ed bridge bridge local

b7b30433a639 docker\_gwbridge bridge local

a7449465c379 host host local

8hq1n8nak54x ingress overlay swarm

06c349b9cc77 none null local

Notice that the “overnet” network does **not** appear in the list. This is because Docker only extends overlay networks to hosts when they are needed. This is usually when a host runs a task from a service that is created on the network. We will see this shortly.

Use the docker network inspect <network> command to view more detailed information about the “overnet” network. You will need to run this command from the first terminal.

docker network inspect overnet

[

{

"Name": "overnet",

"Id": "wlqnvajmmzskn84bqbdi1ytuy",

"Created": "0001-01-01T00:00:00Z",

"Scope": "swarm",

"Driver": "overlay",

"EnableIPv6": false,

"IPAM": {

"Driver": "default",

"Options": null,

"Config": []

},

"Internal": false,

"Attachable": false,

"Containers": null,

"Options": {

"com.docker.network.driver.overlay.vxlanid\_list": "4097"

},

"Labels": null

}

]

Step 3: Create a service

Now that we have a Swarm initialized and an overlay network, it’s time to create a service that uses the network.

Execute the following command from the first terminal to create a new service called *myservice* on the *overnet* network with two tasks/replicas.

docker service create --name myservice \

--network overnet \

--replicas 2 \

ubuntu sleep infinity

ov30itv6t2n7axy2goqbfqt5e

Verify that the service is created and both replicas are up by running docker service ls.

docker service ls

ID NAME MODE REPLICAS IMAGE

ov30itv6t2n7 myservice replicated 2/2 ubuntu:latest

The 2/2 in the REPLICAS column shows that both tasks in the service are up and running.

Verify that a single task (replica) is running on each of the two nodes in the Swarm by running docker service ps myservice.

docker service ps myservice

ID NAME IMAGE NODE DESIRED STATE CURRENT STATE ERROR PORTS

riicggj5tuta myservice.1 ubuntu:latest node2 Running Running about a minute ago

nlozn82wsttv myservice.2 ubuntu:latest node1 Running Running about a minute ago

The ID and NODE values might be different in your output. The important thing to note is that each task/replica is running on a different node.

Now that the second node is running a task on the “overnet” network it will be able to see the “overnet” network. Lets run docker network ls from the second terminal to verify this.

docker network ls

NETWORK ID NAME DRIVER SCOPE

55f10b3fb8ed bridge bridge local

b7b30433a639 docker\_gwbridge bridge local

a7449465c379 host host local

8hq1n8nak54x ingress overlay swarm

06c349b9cc77 none null local

wlqnvajmmzsk overnet overlay swarm

We can also run docker network inspect overnet on the second terminal to get more detailed information about the “overnet” network and obtain the IP address of the task running on the second terminal.

docker network inspect overnet

[

{

"Name": "overnet",

"Id": "wlqnvajmmzskn84bqbdi1ytuy",

"Created": "2017-04-04T09:35:47.526642642Z",

"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,

"Attachable": false,

"Containers": {

"fbc8bb0834429a68b2ccef25d3c90135dbda41e08b300f07845cb7f082bcdf01": {

"Name": "myservice.1.riicggj5tutar7h7sgsvqg72r",

"EndpointID": "8edf83ebce77aed6d0193295c80c6aa7a5b76a08880a166002ecda3a2099bb6c",

"MacAddress": "02:42:0a:00:00:03",

"IPv4Address": "10.0.0.3/24",

"IPv6Address": ""

}

},

"Options": {

"com.docker.network.driver.overlay.vxlanid\_list": "4097"

},

"Labels": {},

"Peers": [

{

"Name": "node1-f6a6f8e18a9d",

"IP": "10.0.0.5"

},

{

"Name": "node2-507a763bed93",

"IP": "10.0.0.6"

}

]

}

]

You should note that as of Docker 1.12, docker network inspect only shows containers/tasks running on the local node. This means that 10.0.0.3 is the IPv4 address of the container running on the second node. Make a note of this IP address for the next step (the IP address in your lab might be different than the one shown here in the lab guide).

Step 4: Test the network

To complete this step you will need the IP address of the service task running on **node2** that you saw in the previous step (10.0.0.3).

Execute the following commands from the first terminal.

docker network inspect overnet

[

{

"Name": "overnet",

"Id": "wlqnvajmmzskn84bqbdi1ytuy",

"Created": "2017-04-04T09:35:47.362263887Z",

"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,

"Attachable": false,

"Containers": {

"d676496d18f76c34d3b79fbf6573a5672a81d725d7c8704b49b4f797f6426454": {

"Name": "myservice.2.nlozn82wsttv75cs9vs3ju7vs",

"EndpointID": "36638a55fcf4ada2989650d0dde193bc2f81e0e9e3c153d3e9d1d85e89a642e6",

"MacAddress": "02:42:0a:00:00:04",

"IPv4Address": "10.0.0.4/24",

"IPv6Address": ""

}

},

"Options": {

"com.docker.network.driver.overlay.vxlanid\_list": "4097"

},

"Labels": {},

"Peers": [

{

"Name": "node1-f6a6f8e18a9d",

"IP": "10.0.0.5"

},

{

"Name": "node2-507a763bed93",

"IP": "10.0.0.6"

}

]

}

]

Notice that the IP address listed for the service task (container) running is different to the IP address for the service task running on the second node. Note also that they are on the same “overnet” network.

Run a docker ps command to get the ID of the service task so that you can log in to it in the next step.

docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

d676496d18f7 ubuntu@sha256:dd7808d8792c9841d0b460122f1acf0a2dd1f56404f8d1e56298048885e45535 "sleep infinity" 10 minutes ago Up 10 minutes myservice.2.nlozn82wsttv75cs9vs3ju7vs

<Snip>

Log on to the service task. Be sure to use the container ID from your environment as it will be different from the example shown below. We can do this by running docker exec -it <CONTAINER ID> /bin/bash.

docker exec -it yourcontainerid /bin/bash

root@d676496d18f7:/#

Install the ping command and ping the service task rundicning on the second node where it had a IP address of 10.0.0.3 from the docker network inspect overnet command.

apt-get update && apt-get install -y iputils-ping

Now, lets ping 10.0.0.3.

root@d676496d18f7:/# ping -c5 10.0.0.3

PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.

^C

--- 10.0.0.3 ping statistics ---

4 packets transmitted, 0 received, 100% packet loss, time 2998ms

The output above shows that both tasks from the **myservice** service are on the same overlay network spanning both nodes and that they can use this network to communicate.

Step 5: Test service discovery

Now that you have a working service using an overlay network, let’s test service discovery.

If you are not still inside of the container, log back into it with the docker exec -it <CONTAINER ID> /bin/bash command.

Run cat /etc/resolv.conf form inside of the container.

docker exec -it yourcontainerid /bin/bash

cat /etc/resolv.conf

search ivaf2i2atqouppoxund0tvddsa.jx.internal.cloudapp.net

nameserver 127.0.0.11

options ndots:0

The value that we are interested in is the nameserver 127.0.0.11. This value sends all DNS queries from the container to an embedded DNS resolver running inside the container listening on 127.0.0.11:53. All Docker container run an embedded DNS server at this address.

**NOTE:** Some of the other values in your file may be different to those shown in this guide.

Try and ping the “myservice” name from within the container by running ping -c5 myservice.

root@d676496d18f7:/# ping -c5 myservice

PING myservice (10.0.0.2) 56(84) bytes of data.

64 bytes from 10.0.0.2: icmp\_seq=1 ttl=64 time=0.020 ms

64 bytes from 10.0.0.2: icmp\_seq=2 ttl=64 time=0.052 ms

64 bytes from 10.0.0.2: icmp\_seq=3 ttl=64 time=0.044 ms

64 bytes from 10.0.0.2: icmp\_seq=4 ttl=64 time=0.042 ms

64 bytes from 10.0.0.2: icmp\_seq=5 ttl=64 time=0.056 ms

--- myservice ping statistics ---

5 packets transmitted, 5 received, 0% packet loss, time 4001ms

rtt min/avg/max/mdev = 0.020/0.042/0.056/0.015 ms

The output clearly shows that the container can ping the myservice service by name. Notice that the IP address returned is 10.0.0.2. In the next few steps we’ll verify that this address is the virtual IP (VIP) assigned to the myservice service.

Type the exit command to leave the exec container session and return to the shell prompt of your Docker host.

root@d676496d18f7:/# exit

Inspect the configuration of the “myservice” service by running docker service inspect myservice. Lets verify that the VIP value matches the value returned by the previous ping -c5 myservice command.

docker service inspect myservice

[

{

"ID": "ov30itv6t2n7axy2goqbfqt5e",

"Version": {

"Index": 19

},

"CreatedAt": "2017-04-04T09:35:47.009730798Z",

"UpdatedAt": "2017-04-04T09:35:47.05475096Z",

"Spec": {

"Name": "myservice",

"TaskTemplate": {

"ContainerSpec": {

"Image": "ubuntu:latest@sha256:dd7808d8792c9841d0b460122f1acf0a2dd1f56404f8d1e56298048885e45535",

"Args": [

"sleep",

"infinity"

],

<Snip>

"Endpoint": {

"Spec": {

"Mode": "vip"

},

"VirtualIPs": [

{

"NetworkID": "wlqnvajmmzskn84bqbdi1ytuy",

"Addr": "10.0.0.2/24"

}

]

},

<Snip>

Towards the bottom of the output you will see the VIP of the service listed. The VIP in the output above is 10.0.0.2 but the value may be different in your setup. The important point to note is that the VIP listed here matches the value returned by the ping -c5 myservice command.

Feel free to create a new docker exec session to the service task (container) running on **node2** and perform the same ping -c5 service command. You will get a response form the same VIP.

Cleaning Up

Hopefully you were able to learn a little about how Docker Networking works during this lab. Lets clean up the service we created, the containers we started, and finally disable Swarm mode.

Execute the docker service rm myservice command to remove the service called *myservice*.

docker service rm myservice

Execute the docker ps command to get a list of running containers.

docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

846af8479944 ubuntu "sleep infinity" 17 minutes ago Up 17 minutes heuristic\_boyd

4e0da45b0f16 nginx "nginx -g 'daemon ..." 12 minutes ago Up 12 minutes 443/tcp, 0.0.0.0:8080->80/tcp web1

You can use the docker kill <CONTAINER ID ...> command to kill the ubunut and nginx containers we started at the beginning.

docker kill yourcontainerid1 yourcontainerid2

Finally, lets remove node1 and node2 from the Swarm. We can use the docker swarm leave --force command to do that.

Lets run docker swarm leave --force.

docker swarm leave --force

Lets also run docker swarm leave --force.

docker swarm leave --force