Docker Networking Basics

Difficulty: Beginner

Time: Approximately 10 minutes

In this lab you'll look at the most basic networking components that come with a fresh installation of Docker.

You will complete the following steps as part of this lab.

Prerequisites

You will need all of the following to complete this lab:

- A Linux-based Docker Host running Docker 1.12 or higher

 Step 1: The `docker network` command

The `docker network` command is the main command for configuring and managing container networks.

Run a simple `docker network` command from any of your lab machines.

```
$ docker network

Usage:  docker network COMMAND

Manage Docker 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 net
works
  ls          List 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              S
COPE
1befe23acd58        bridge              bridge              l
ocal
726ead8f4e6b        host                host                l
ocal
ef4896538cc7        none                null                l
ocal
```

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` 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": "1befe23acd58cbda7290c45f6d1f5c37a3b43de645d48d
e6c1ffebd985c8af4b",
        "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,
        "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 a `docker info` command on any of your Docker hosts and locate the list of network plugins.

```
$ docker info
Containers: 0
 Running: 0
 Paused: 0
 Stopped: 0
Images: 0
Server Version: 1.12.3
Storage Driver: aufs
<Snip>
Plugins:
 Volume: local
 Network: bridge host null overlay    <<<<<<<<
Swarm: inactive
Runtimes: runc
<Snip>
```

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

Bridge networking

Difficulty: Intermediate

Time: Approximately 15 minutes

In this lab you'll learn how to build, manage, and use **bridge** networks.

You will complete the following steps as part of this lab.

Prerequisites

You will need all of the following to complete this lab:

- A Linux-based Docker host running Docker 1.12 or higher
- The lab was built and tested using Ubuntu 16.04

Step 1: The default bridge network

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

```
$ docker network ls
NETWORK ID          NAME                DRIVER              S
COPE
1befe23acd58        bridge              bridge              l
ocal
726ead8f4e6b        host                host                l
ocal
ef4896538cc7        none                null                l
ocal
```

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.

```
# Install the brctl tools
```

```
$ apt-get install bridge-utils
<Snip>

# List the bridges on your Docker host

$ brctl show
bridge name     bridge id               STP enabled     inter
faces
docker0         8000.0242f17f89a6       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` command to view details of the **docker0** bridge.

```
$ ip a
<Snip>
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdis
c noqueue state DOWN group default
    link/ether 02:42:f1:7f:89:a6 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:f1ff:fe7f:89a6/64 scope link
       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.

```
$ docker run -dt ubuntu sleep infinity
6dd93d6cdc806df6c7812b6202f6096e43d9a013e56e5e638ee4bfb4ae877
9ce
```

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. 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     inter
faces
docker0         8000.0242f17f89a6       no              veth3
a080f
```

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

Inspect the **bridge** network again to see the new container attached to it.

```
$ docker network inspect bridge
<Snip>
        "Containers": {
            "6dd93d6cdc806df6c7812b6202f6096e43d9a013e56e5e63
8ee4bfb4ae8779ce": {
                "Name": "reverent_dubinsky",
                "EndpointID": "dda76da5577960b30492fdf1526c7d
d7924725e5d654bed57b44e1a6e85e956c",
                "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. Remember to use the IP of the container in **your** environment.

```
$ ping 172.17.0.2
64 bytes from 172.17.0.2: icmp_seq=1 ttl=64 time=0.069 ms
64 bytes from 172.17.0.2: icmp_seq=2 ttl=64 time=0.052 ms
64 bytes from 172.17.0.2: icmp_seq=3 ttl=64 time=0.050 ms
64 bytes from 172.17.0.2: icmp_seq=4 ttl=64 time=0.049 ms
64 bytes from 172.17.0.2: icmp_seq=5 ttl=64 time=0.049 ms
^C
--- 172.17.0.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 3999m
s
rtt min/avg/max/mdev = 0.049/0.053/0.069/0.012 ms
```

Press `Ctrl-C` to stop the ping. The replies above show that the Docker host can ping the container over the **bridge** network.

Log in to the container, install the `ping` program and ping `www.dockercon.com`.

```
# Get the ID of the container started in the previous step.
$ docker ps
CONTAINER ID    IMAGE    COMMAND             CREATED  STATUS
 NAMES
```

```
6dd93d6cdc80    ubuntu   "sleep infinity"    5 mins   Up
 reverent_dubinsky

# Exec into the container
$ docker exec -it 6dd93d6cdc80 /bin/bash

# Update APT package lists and install the iputils-ping packa
ge
root@6dd93d6cdc80:/# apt-get update
<Snip>

apt-get install iputils-ping
Reading package lists... Done
<Snip>

# Ping www.dockercon.com from within the container
root@6dd93d6cdc80:/# ping www.dockercon.com
PING www.dockercon.com (104.239.220.248) 56(84) bytes of data
.
64 bytes from 104.239.220.248: icmp_seq=1 ttl=39 time=93.9 ms
64 bytes from 104.239.220.248: icmp_seq=2 ttl=39 time=93.8 ms
64 bytes from 104.239.220.248: icmp_seq=3 ttl=39 time=93.8 ms
^C
--- www.dockercon.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002m
s
rtt min/avg/max/mdev = 93.878/93.895/93.928/0.251 ms
```

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.

```
$ docker run --name web1 -d -p 8080:80 nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
386a066cd84a: Pull complete
7bdb4b002d7f: Pull complete
49b006ddea70: Pull complete
Digest: sha256:9038d5645fa5fcca445d12e1b8979c87f46ca42cfb17be
b1e5e093785991a639
Status: Downloaded newer image for nginx:latest
b747d43fa277ec5da4e904b932db2a3fe4047991007c2d3649e3f0c615961
```

```
038
```

Check that the container is running and view the port mapping.

```
$ docker ps
CONTAINER ID    IMAGE                COMMAND
CREATED              STATUS               PORTS
           NAMES
b747d43fa277   nginx                "nginx -g 'daemon off"   3
 seconds ago       Up 2 seconds         443/tcp, 0.0.0.0:8080-
>80/tcp   web1
6dd93d6cdc80        ubuntu               "sleep infinity"
    About an hour ago   Up About an hour
               reverent_dubinsky
```

There are two containers listed in the output above. 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. if your lab is in AWS this will need to be the instance's Public IP).

Point your web browser to the IP and port 8080 of your Docker host. The following example shows a web browser pointed to `52.213.169.69:8080`

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` command.

```
$ curl 127.0.0.1:8080
<!DOCTYPE html>
<html>
<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).*