Lab 7 - Accessing Docker Networks

Introduction

In this lab, you will learn how to create your own, user-defined networks. In this lab we will be exploring the docker bridge networking options along with useful show commands to see the state of the system.

1. Docker Networking Model

The Docker networking architecture is built on a set of interfaces called the Container Networking Model (CNM). The philosophy of CNM is to provide application portability across diverse infrastructures. This model strikes a balance to achieve application portability and takes advantage of special features and capabilities of the infrastructure.

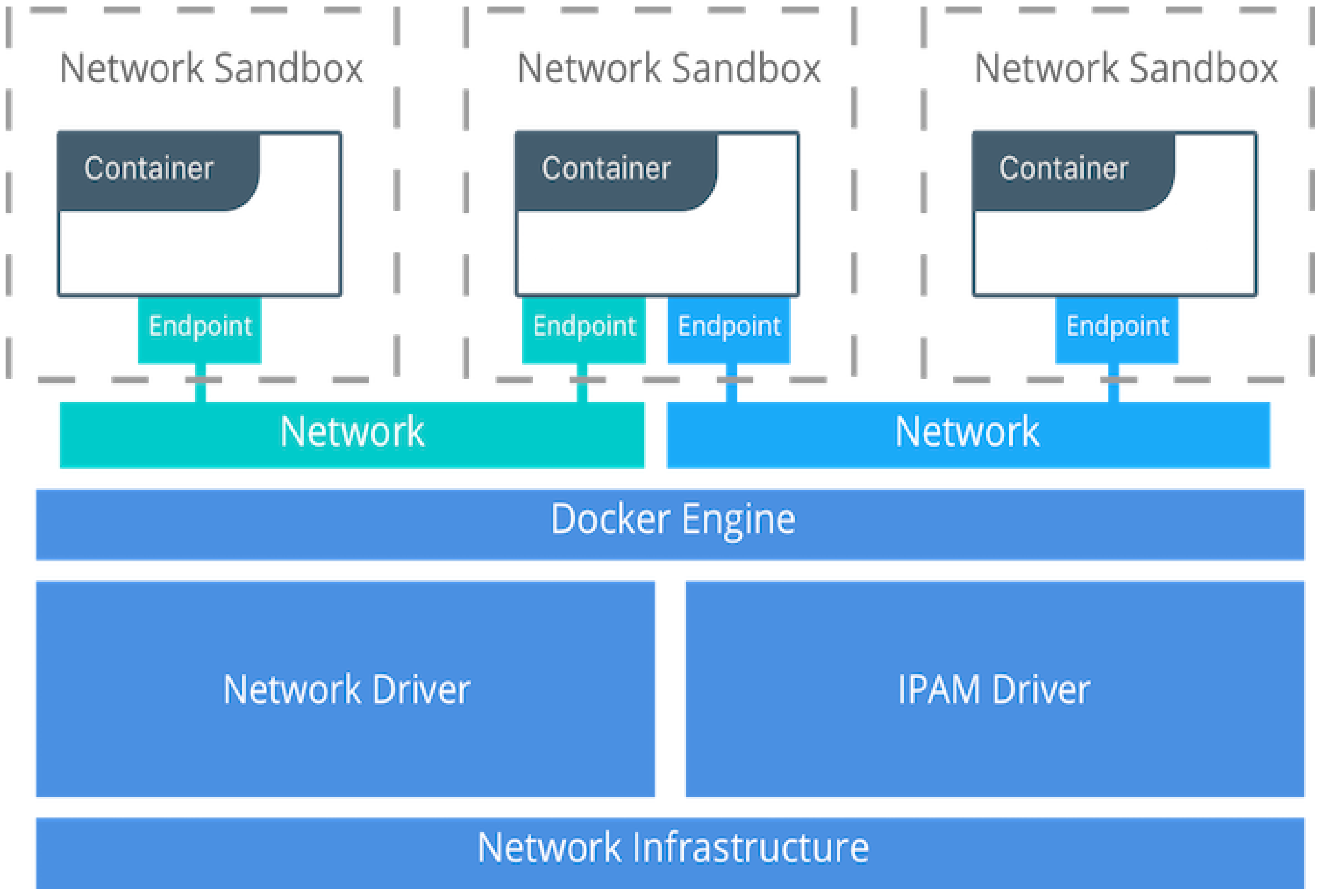


Fig 1: Architecture of Container Networking Model

- **Sandbox** — A Sandbox contains the configuration of a container's network stack. This includes management of the container's interfaces, routing table, and DNS settings. An implementation of a Sandbox could be a Linux Network Namespace, a FreeBSD Jail, or another similar concept. A Sandbox may contain many endpoints from multiple networks.
- **Endpoint** — An Endpoint joins a Sandbox to a Network. The Endpoint construct exists so the actual connection to the network can be abstracted away from the application. This helps maintain portability so that a service can use different types of network drivers without being concerned with how it's connected to that network.
- **Network** — The CNM does not specify a Network in terms of the OSI model. An implementation of a Network could be a Linux bridge, a VLAN, etc. A Network is a collection of endpoints that have connectivity between them. Endpoints that are not connected to a network will not have connectivity on a Network.

2. Identify Container IP using inspect

Docker provides the networking primitives to allow you to specify how different containers network with each other. “It’s evolving very quickly to the point where you can take a complex application, Dockerize each of its components, and distribute them across many servers and have coordination between them.

2.1 Login as **“root”** user on **aio110** host:

Copy

```
ssh root@aio110
```

2.2 Explore existing networks

Copy

```
docker network ls
```

Output:

```
NETWORK ID NAME DRIVER SCOPE

440c94c79422 bridge bridge local

d5198b0f55c4 host host local

342a5cd17896 none null local
```

The above were created by default when docker was installed.

2.3 Inspect the docker network id, to display detailed information

Copy

```
bridge_id=`docker network ls | grep bridge | cut -d " " -f 1`

echo "Default Bridge ID is: $bridge_id"
```

Sample Output:

```
Default Bridge ID is: 64b7de50b902
```

Copy

```
docker inspect $bridge_id
```

Output:

```
[

    {

        "Name": "bridge",

        "Id":
"440c94c79422c9ddbf878384cce1cdbe7a15e94fe12db2d30e3e5d01b34dc8ff",

        "Created": "2018-02-26T17:50:19.214618428Z",

        "Scope": "local",

        "Driver": "bridge",

        "EnableIPv6": false,

        "IPAM": {

            "Driver": "default",

            "Options": null,

            "Config": [

                {

                    "Subnet": "172.17.0.0/16"

                }

            ]

        },

        "Internal": false,
        "Attachable": false,

        "Ingress": 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": {}

    }

]
```

Copy

```
ifconfig docker0
```

Sample Output

```
docker0: flags=4099  mtu 1500

        inet 172.17.0.1  netmask 255.255.0.0  broadcast 0.0.0.0

        inet6 fe80::42:8aff:fe78:c51e  prefixlen 64  scopeid 0x20

        ether 02:42:8a:78:c5:1e  txqueuelen 0  (Ethernet)
        RX packets 16613  bytes 861251 (841.0 KiB)

        RX errors 0  dropped 0  overruns 0  frame 0

        TX packets 29844  bytes 41337094 (39.4 MiB)

        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Above shows the **docker0** bridge created.

Note: 172.17.0.1 has been assigned to the bridge and will serve as the default gaetway for the containers connected to this network.

2.4 Create a Docker network

Syntax :

```
docker network create [OPTIONS] NETWORK
```

Note: Options are –

—driver string : Driver to manage the Network (default "bridge")
—ip-range value : Allocate container ip from a sub-range (default [])
—subnet value : Subnet in CIDR format that represents a network segment (default [])
—gateway value : IPv4 or IPv6 Gateway for the master subnet (default [])
—label value : Set metadata on a network (default [])
-ipv6 : Enable IPv6 networking

.

Copy

```
docker network create --subnet 172.20.0.0/16 --ip-range
172.20.240.0/20 simple-network
```

Sample Output:

```
5cbc067b2c5b1536bced47da3190186f7b7328ba4c7ea36c7d92de2aa03a34ec
```

2.5 List Docker networks

Syntax :

```
docker network ls [OPTIONS]
```

Note: Options are –

-f, —filter value :Provide filter values (i.e. ‘dangling=true’) (default [])
—no-trunc : Do not truncate the output

.

Lists all the networks the Engine daemon knows about. This includes the networks that span across multiple hosts in a cluster,

Copy

```
docker network ls
```

Example Output

```
NETWORK ID          NAME                DRIVER              SCOPE

f497e6b1850c        bridge              bridge              local

011c219a6d14        host                host                local

9f62c54f2fa5        none                null                local

41a709ce7140        simple-network      bridge              local
```

2.6 Use the —no-trunc option to display the full network id:

Copy

```
docker network ls --no-trunc
```

Example Output

```
NETWORK ID
NAME                DRIVER

18a2866682b85619a026c81b98a5e375bd33e1b0936a26cc497c283d27bae9b3
none                null
```

```
c288470c46f6c8949c5f7e5099b5b7947b07eabe8d9a27d79a9cbf111adcbf47
host                host

7b369448dccbf865d397c8d2be0cda7cf7edc6b0945f77d2529912ae917a0185
bridge              bridge

95e74588f40db048e86320c6526440c504650a1ff3e9f7d60a497c4d2163e5bd   foo
bridge

63d1ff1f77b07ca51070a8c227e962238358bd310bde1529cf62e6c307ade161   dev
bridge
```

2.7 Verify that, another bridge created

Copy

```
ifconfig -a | grep -A 6 "^br"
```

Sample Output

```
br-d783d432fb59: flags=4099  mtu 1500

        inet 172.20.240.0  netmask 255.255.0.0  broadcast 0.0.0.0

        ether 02:42:df:f1:96:eb  txqueuelen 0  (Ethernet)

        RX packets 0  bytes 0 (0.0 B)

        RX errors 0  dropped 0  overruns 0  frame 0

        TX packets 0  bytes 0 (0.0 B)

        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

2.8 There are 2 different ways to connect container to a specific network

- During container creation
- For the existing container

2.9 During container creation

Start a container, use the **—network** flag to connect it to a network. This example adds the **Test** container to the **simple-network**.

Syntax :

```
docker run -dit --network={Network-Name} {Container name} {image-name}
```

Copy

```
docker run -dit --network=simple-network --name Test1 centos
```

Copy

```
docker inspect Test1 | grep IPAddress
```

Output

```
           "SecondaryIPAddresses": null,

            "IPAddress": "",

                    "IPAddress": "172.20.240.1",
```

2.10 Make sure attach to instance and do “ip a” to check IP address which should match the above

Copy

```
docker ps -a
```

Output

```
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES

edf059a250fb        centos              "/bin/bash"         2 minutes
ago       Up 2 minutes                            Test1
```

Copy

```
container_id=`docker ps -a | grep Test1 | cut -d " " -f 1`

echo "Test1 container ID is: $container_id"
```

Sample Output:

```
Test1 container ID is: 262b5028a89b
```

Copy

```
docker attach $container_id
```

Sample Output:

```
[root@edf059a250fb /]#
```

Note: Continue below commands on interactive bash.

2.11 Install net-tools and iproute packages

NET-TOOLS : A collection of programs that form the base set of the NET-3 networking distribution for the Linux operating system.

This package includes the important tools for controlling the network subsystem of the Linux kernel. This includes arp, hostname, ifconfig, netstat, rarp and route. Additionally, this package contains utilities relating to particular network hardware types (plipconfig, slattach, mii-tool) and advanced aspects of IP configuration (iptunnel, ipmaddr).

IPROUTE : Iproute is a collection of utilities for controlling **TCP / IP** networking and traffic control in Linux.

Copy

```
yum install -y net-tools iproute
```

Copy

```
 ip a
```

Output:

```
1: lo:  mtu 65536 qdisc noqueue state UNKNOWN qlen 1

    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

    inet 127.0.0.1/8 scope host lo

       valid_lft forever preferred_lft forever

37: eth0@if38:  mtu 1500 qdisc noqueue state UP

    link/ether 02:42:ac:14:f0:01 brd ff:ff:ff:ff:ff:ff link-netnsid 0

    inet 172.20.240.1/16 scope global eth0

       valid_lft forever preferred_lft forever
```

Copy

```
ip route
```

Output:

```
default via 172.20.240.0 dev eth0

172.20.0.0/16 dev eth0 proto kernel scope link src 172.20.240.1
```

Copy

```
ping -c 4 8.8.8.8
```

Output:

```
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.

64 bytes from 8.8.8.8: icmp_seq=1 ttl=57 time=1.63 ms

64 bytes from 8.8.8.8: icmp_seq=2 ttl=57 time=1.71 ms
```

```
64 bytes from 8.8.8.8: icmp_seq=3 ttl=57 time=1.65 ms

64 bytes from 8.8.8.8: icmp_seq=4 ttl=57 time=1.73 ms



--- 8.8.8.8 ping statistics ---

4 packets transmitted, 4 received, 0% packet loss, time 3005ms

rtt min/avg/max/mdev = 1.639/1.686/1.734/0.056 ms




[root@74b65cc83784 /]#
```

Copy

```
exit
```

2.12 Docker network connect

Connect a container to a network

Syntax :

```
docker network connect [OPTIONS] NETWORK CONTAINER
```

Note: Options are –

—ip string :IP Address
—ip6 string :IPv6 Address
—link value :Add link to another container (default [])
—link-local-ip value :Add a link-local address for the container (default [])

.

Copy

```
docker run -dit --name Test2 centos
```

Example Output

```
f2870c98fd504370fb86e59f32cd0753b1ac9b69b7d80566ffc7192a82b3ed27
```

Copy

```
docker inspect Test2 | grep IPAddress
```

Output

```
          "SecondaryIPAddresses": null,

            "IPAddress": "172.17.0.2",

                    "IPAddress": "172.17.0.2",
```

Copy

```
docker network connect --ip 172.20.128.2 simple-network Test2
```

Copy

```
docker inspect Test2 | grep IPAddress
```

Output

```
           "SecondaryIPAddresses": null,

            "IPAddress": "172.17.0.2",

                    "IPAddress": "172.17.0.2",

                    "IPAddress": "172.20.128.2",
```

2.13 Docker network inspect

Display detailed information on one or more networks

Syntax :

```
docker network inspect [OPTIONS] NETWORK [NETWORK...]
```

Note: Options are –

-f, **—format string** : Format the output using the given go template
—help : Print usage

Returns information about one or more networks. By default, this command renders all results in a JSON object.

The **network inspect** command shows the containers, by id, in its results. This command also shows the container endpoints in other hosts in the cluster. These endpoints are represented as “ep-{endpoint-id}” in the output.

Copy

```
docker network inspect simple-network
```

Example Output

```
[

    {

        "Name": "simple-network",

        "Id":
"41a709ce714038dad585c87862a0194d23f4242f742069d81e8de3817292b4df",

        "Created": "2018-02-27T05:33:58.730552437Z",

        "Scope": "local",

        "Driver": "bridge",

        "EnableIPv6": false,

        "IPAM": {

            "Driver": "default",

            "Options": {},
            "Config": [
                {
                    "Subnet": "172.20.0.0/16",
                    "IPRange": "172.20.240.0/20"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "Containers": {
"3905767e5e34814307568aa06a54ae37d0f4bea820f72b548506df8f1397a475": {
                "Name": "Test2",
                "EndpointID":
"1d2a625dd916877cbce3d4a2d4d095d8d88dc67c513b65f4cee5602dac692be2",
                "MacAddress": "02:42:ac:14:80:02",
                "IPv4Address": "172.20.128.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {}
```

```
    }

]
```

2.14 Docker network disconnect

Disconnect a container from a network

Syntax :

```
docker network disconnect [OPTIONS] NETWORK CONTAINER
```

Note: Options are –

-f, —force :Force the container to disconnect from a network

Copy

```
docker network disconnect simple-network Test1
```

Copy

```
docker network disconnect simple-network Test2
```

2.15 Docker network rm

Remove one or more networks

Syntax :

```
 docker network rm NETWORK [NETWORK...]
```

Removes one or more networks by name or identifier. To remove a network, you must first disconnect any containers connected to it. To remove the network named 'my-network'.

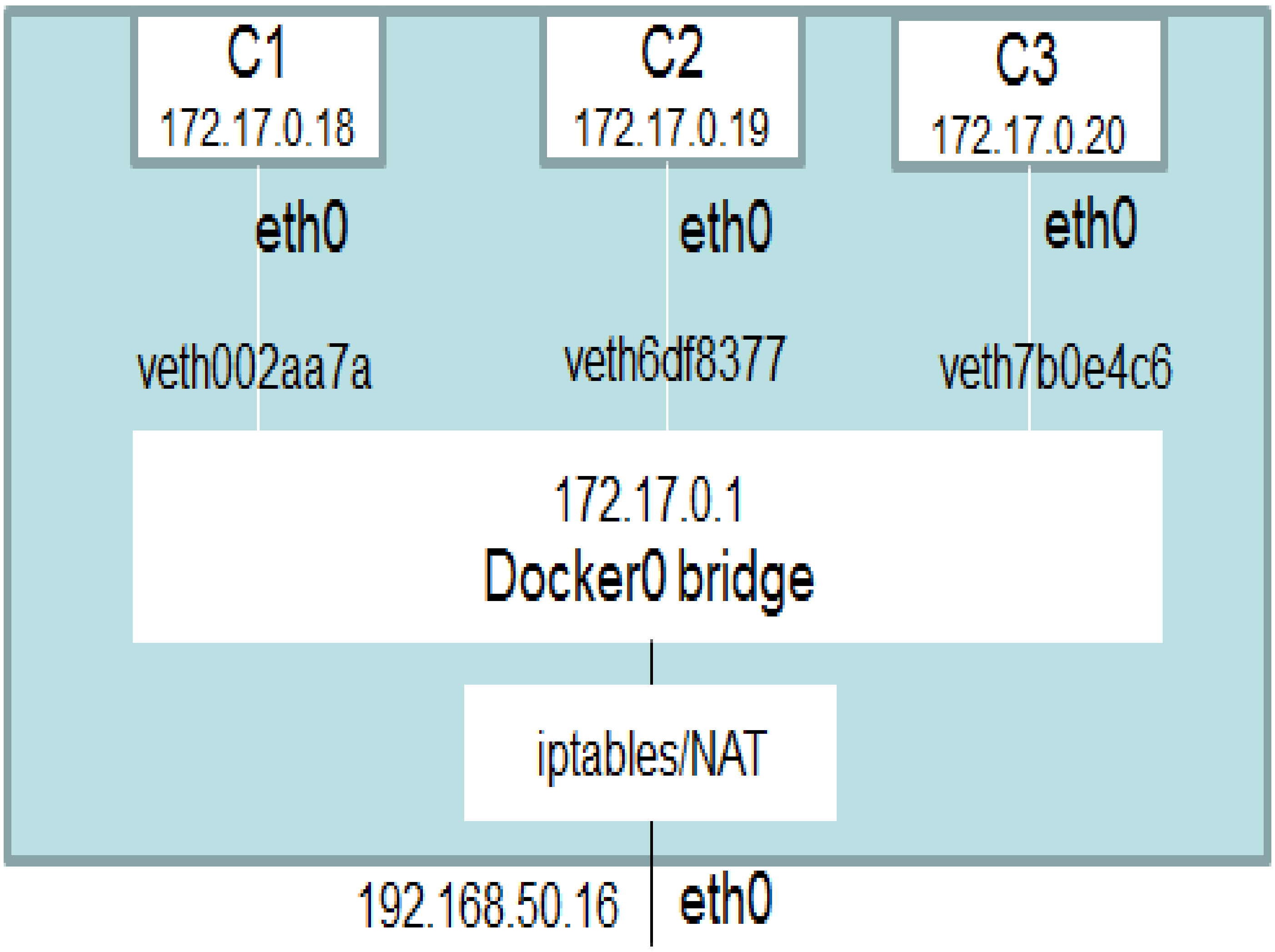
Copy

```
docker network rm simple-network
```

3. Exploring Virtual Ethernet Bridge

The Docker server creates and configures the host system’s **docker0** interface as an Ethernet bridge inside the Linux kernel that could be used by the Docker containers to communicate with each other and with the outside world.

The **docker0 bridge** is virtual interface created by Docker, it randomly chooses an address and subnet from the private range defined by RFC 1918 that are not in use on the host machine, and assigns it to docker0. All the Docker containers will be connected to the docker0 bridge by default, the Docker containers connected to the docker0 bridge could use the **iptables NAT** rules created by Docker to communicate with the outside world.



Creating the docker0 bridge

3.1 The docker0 bridge will be created when the Docker service is started.

Copy

```
yum install bridge-utils -y
```

Copy

```
brctl show
```

Example Output

```
bridge name     bridge id               STP enabled     interfaces

docker0         8000.02421f09cf2e       no              veth07bc80b

                                                        veth308b44b

                                                        vetha67c827
```

3.2 Docker also creates the iptables NAT rules on the Docker host that could be used by the docker containers connected to docker0 bridge to connect to the outside world.

Copy

```
iptables -t nat -L
```

Example Output

```
Chain PREROUTING (policy ACCEPT)

target     prot opt source               destination

DOCKER     all  --  anywhere             anywhere             ADDRTYPE
match dst-type LOCAL



Chain INPUT (policy ACCEPT)

target     prot opt source               destination
```

```
Chain OUTPUT (policy ACCEPT)

target     prot opt source               destination

DOCKER     all  --  anywhere            !loopback/8           ADDRTYPE
match dst-type LOCAL



Chain POSTROUTING (policy ACCEPT)

target     prot opt source               destination

MASQUERADE  all  --  172.17.0.0/16        anywhere

MASQUERADE  tcp  --  172.17.0.2           172.17.0.2           tcp
dpt:http



Chain DOCKER (2 references)

target     prot opt source               destination

RETURN     all  --  anywhere             anywhere

DNAT       tcp  --  anywhere             anywhere             tcp
dpt:http to:172.17.0.2:80
```

3.3 Connects the Docker containers to docker0 bridge

By default, Docker will attach all containers to the docker0 bridge, unless the DOCKER_OPTS in Docker configuration file explicitly specifies to use the other network than docker0 bridge, in this case you could use –net=bridge with Docker run command to connect the containers to the docker0 bridge.

Copy

```
docker run -dit --name docker3 --net=bridge centos
```

Example Output

```
2b4b9e7612e5593f31515e46e54ffc4ab7902ffdc6162d663f87e741b3b91117
```

Copy

```
docker inspect docker3
```

Example Output

```
... ... ...

 "Networks": {

 "bridge": {

 "IPAMConfig": null,

 "Links": null,

 "Aliases": null,

 "NetworkID":
"a6eb0368ed0cbfea8d665e3ec7720942e37a556204652f850a534fcb8863e20e",

 "EndpointID":
"f0174d4c507449da55008d646bbff216064450d21d6506af1b1b0bdab4176c93",

 "Gateway": "172.17.0.1",

 "IPAddress": "172.17.0.4",

 "IPPrefixLen": 16,

 "IPv6Gateway": "",

 "GlobalIPv6Address": "",

 "GlobalIPv6PrefixLen": 0,

 "MacAddress": "02:42:ac:11:00:04"
 }
```

3.4 Use brctl show docker0 to verify if the docker container connects to the docker0 bridge.

Copy

```
brctl show docker0
```

Example Output

```
bridge name     bridge id               STP enabled     interfaces

docker0         8000.0242c22f1810       no              veth020bf90

                                                        veth6e8869b
```

3.5 Use ethtool to check the prefix number of the veths connected to the bridge docker0:

Copy

```
docker_interface=`brctl show docker0 | grep docker0 | cut -f 6`

echo "docker0 interface ID is: $docker_interface"
```

Example Output

```
docker0 interface ID is: veth020bf90
```

Copy

```
ethtool -S $docker_interface
```

Example Output

```
NIC statistics:
     peer_ifindex: 68
```

3.6 Create an Isolated Container

Copy

```
docker run -dit --name isolated --net=none centos
```

Copy

```
docker inspect isolated | grep IPAddress
```

Sample Output:

```
            "SecondaryIPAddresses": null,

            "IPAddress": "",

                    "IPAddress": "",
```

3.7 To remove all the containers run the below commands,

Copy

```
docker rm `docker ps -a -q` -f
```

3.8 To remove all the images run the below commands,

Copy

```
docker rmi `docker images -q` -f
```

This brings us to the end of this lab. We will be exploring the overlay networking driver in a later lab.