CCGC 5001 - Virtualization

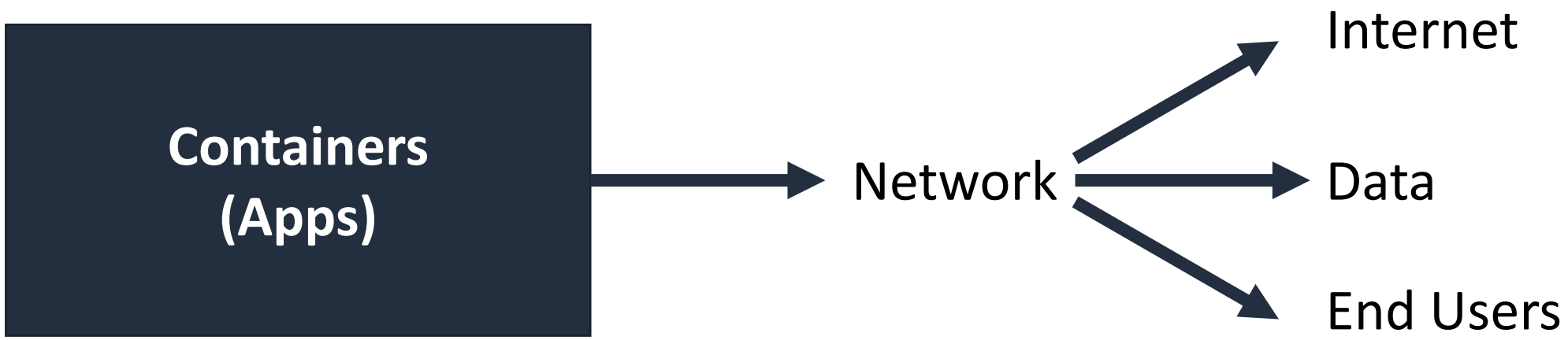
Module 6: Container Networks

Module objectives

At the end of this module, you should be able to:

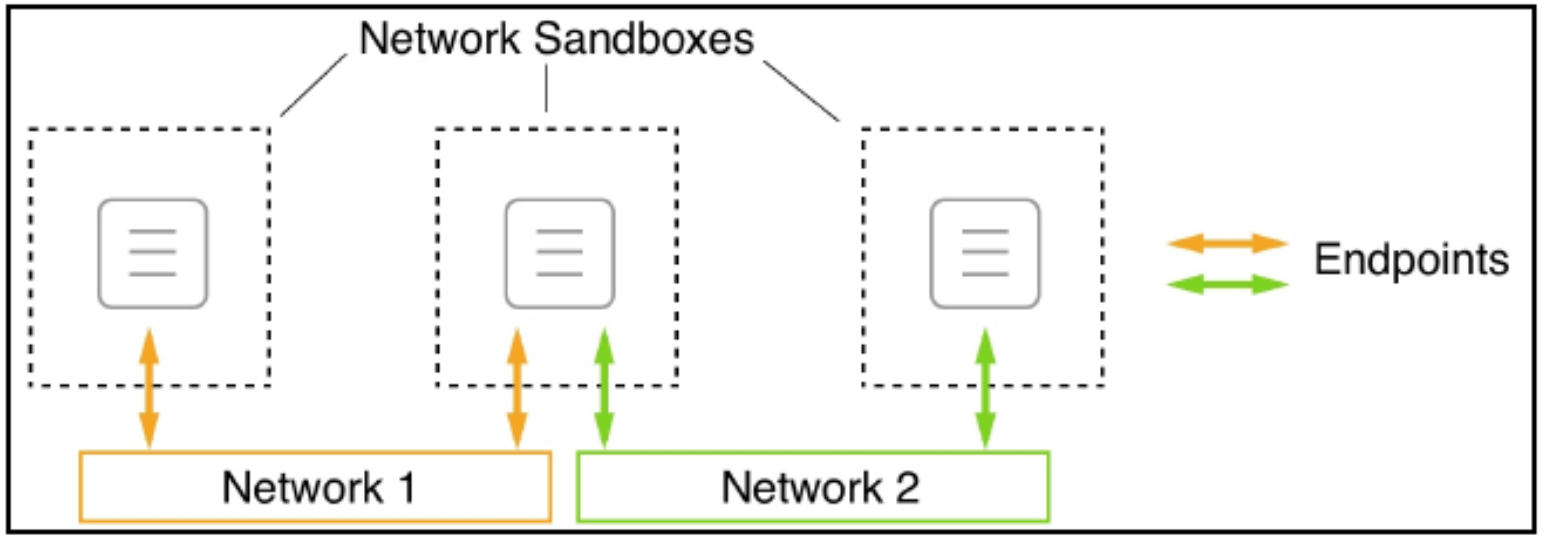
- Create, inspect, and delete a custom bridge network
- Run a container attached to a custom bridge network
- Isolate containers from each other by running them on different bridged networks
- Publish a container port to a host port of your choice

Applications need the network



Container network model (CNM)

Container network model specifies requirements that any software that implements a container network



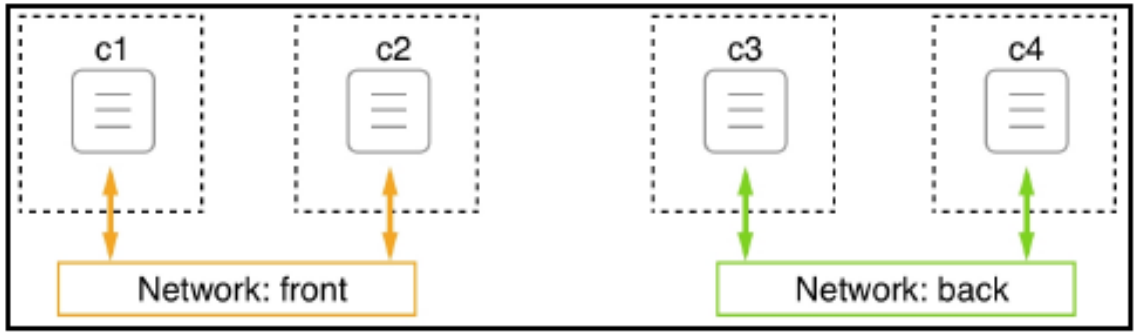
Container network model (CNM)

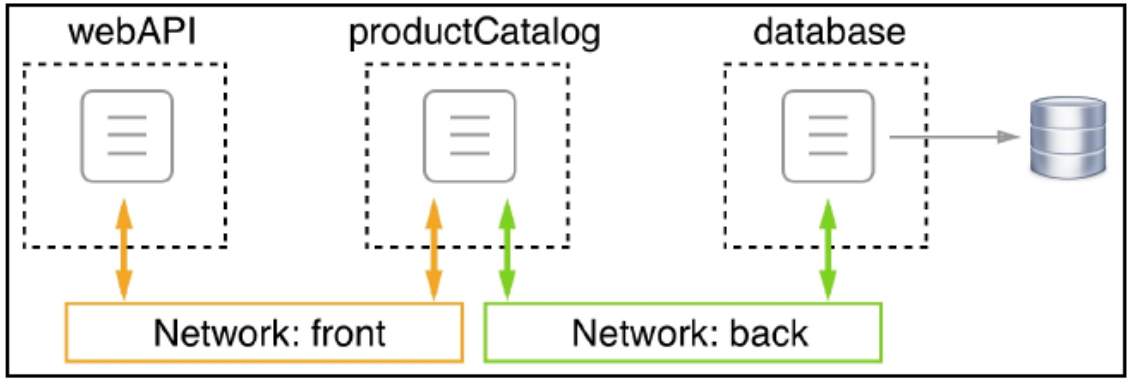
CNM has three elements:

- Sandbox
  - Isolates a container from the outside world
  - Contains configuration of a container's network stack (interfaces, routing table and DNS settings)
- Endpoint
  - Controlled gateway from the outside world
  - Connects the network sandbox to a network
  - Belongs to only one network and one sandbox
- Network
  - Pathway that transports data packet from container to container

Network firewalling

Containers that belong to same network can freely communicate with each other, while others have no means to do so.





Bridge network

Docker daemon during the install creates a Linux bridge and calls it docker0

Docker then creates a network with Linux bridge and calls the network bridge

$ docker network ls

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
928c8ce47bf2        bridge              bridge              local
bdb36adcf70c        host                host                local
af82006f2f2d        none                null                local
$
```

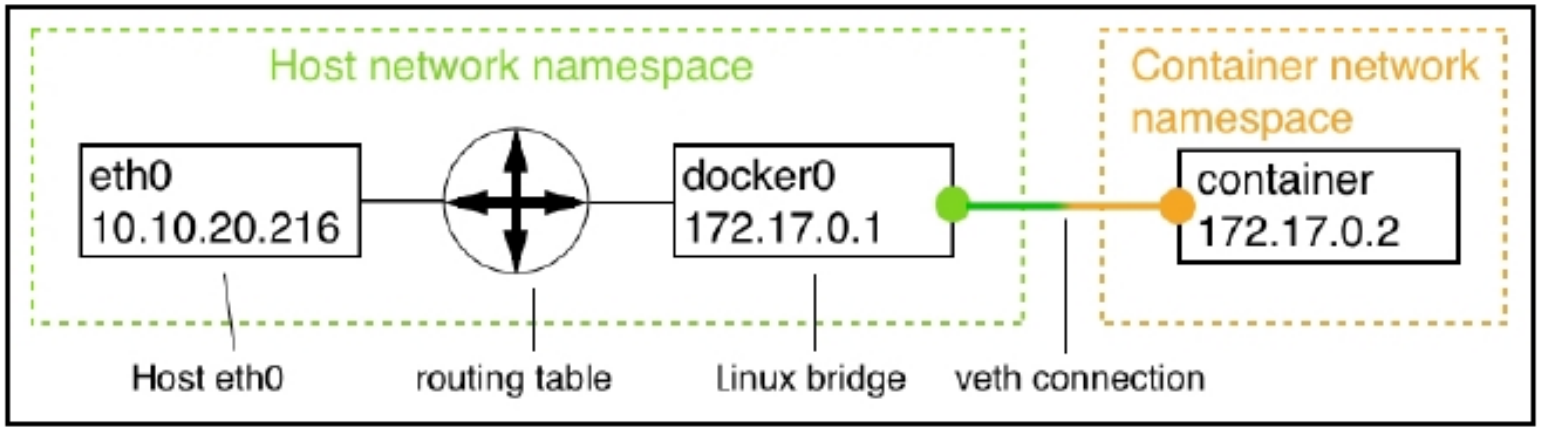
Bridge network

$ docker network inspect bridge

```
C:\Users\admin>docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "3b08c1c711ada84ae859c4bed48b5af1f45b68db89356ca5045dc7ee8672e946",
        "Created": "2018-04-09T09:47:29.9424652Z",
        "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,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": 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": {}
    }
]
```

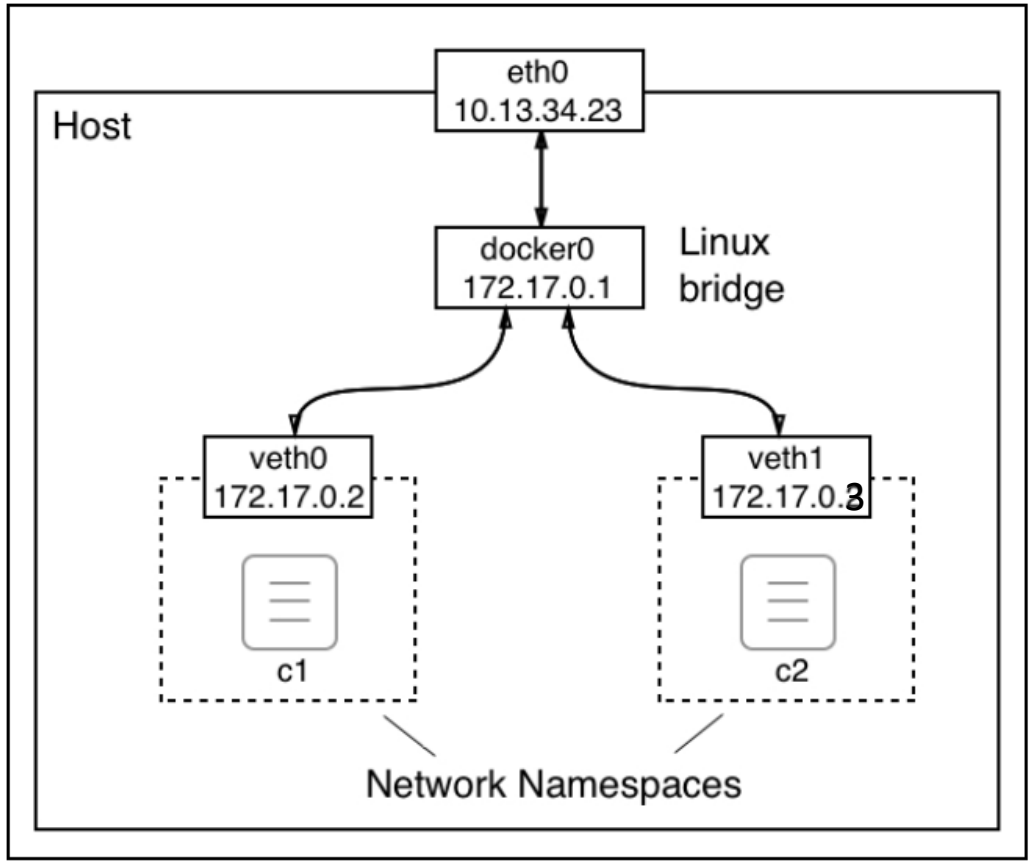
Bridge network

IP address 172.17.0.1 is reserved for the router and is taken by the Linux bridge



Bridge network

By default, only egress traffic is allowed, and all ingress is blocked.



Custom bridge network

Custom bridge network can be created.

```
$ docker network create --driver bridge sample-net
```

```
$ docker network inspect sample-net | grep Subnet
```

```
"Subnet": "172.18.0.0/16",
```

Custom bridge network

Custom bridge network can be created with our own customized subnet range.

```
$ docker network create --driver bridge --subnet "10.1.0.0/16" test-net
```

How containers attach to networks

Let’s interactively run an Alpine container without specifying the network.

```
$ docker container run --name c1 -it --rm alpine:latest /bin/sh
```

```
$ docker container inspect c1
```

How containers attach to networks

Run the ip addr command and observe the output.

/# ip addr

/# ip addr show eth0

/# ip route

How containers attach to networks

Let’s run another Alpine container called c2 without specifying the network.

```
$ docker container run --name c2 -d alpine:latest ping 127.0.0.1
```

```
$ docker container inspect --format "{{.NetworkSettings.IPAddress}}" c2
```

We now have c1 and c2 attached to the bridge network. Inspect this network to find list of all containers.

```
$ docker network inspect bridge
```

How containers attach to networks

Let’s create two additional containers, c3 and c4, and attach them to test-net.

```
$ docker container run --name c3 -d --network test-net alpine:latest ping 127.0.0.1
```

```
$ docker container run --name c4 -d --network test-net alpine:latest ping 127.0.0.1
```

Let’s inspect network test-net and confirm that containers c3 and c4 are indeed attached to it.

```
$ docker network inspect test-net
```

Container communication

Let’s verify that containers are able to communicate with each other.

```
$ docker container exec -it c3 /bin/sh
```

```
/# ping c4
```

To demonstrate that bridge and test-net networks are firewalled from each other

```
/# ping c2
```

```
/# ping c1
```

Container communication

A container can be attached to multiple networks.

```
$ docker container run --name c5 -d \
--network sample-net \
--network test-net \
alpine:latest ping 127.0.0.1
```

```
$ docker container run --name c5 –d --network sample-net alpine:latest ping 127.0.0.1
```

```
$ docker network connect test-net c5
```

Remove a network

A network cannot be removed if a container is attached to it.

```
$ docker network rm test-net
```

Clean up all containers first and then remove the network.

```
$ docker container rm -f $(docker container ls -aq)

$ docker network rm sample-net
$ docker network rm test-net

$ docker network prune --force
```

Host network

- Allows us to run a container in the network namespace of the host
- For security purposes, recommendation is that you do not run container attached to host network on a production environment

```
$ docker container run --rm -it --network host alpine:latest /bin/sh
```

Use the ip tool to analyze the network namespace from within the container

```
/ # ip addr show eth0
eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc pfifo_fast state UP qlen 1000
    link/ether 12:94:36:c2:ec:93 brd ff:ff:ff:ff:ff:ff
    inet 172.31.85.201/20 brd 172.31.95.255 scope global dynamic eth0
       valid_lft 2761sec preferred_lft 2761sec
    inet6 fe80::1094:36ff:fec2:ec93/64 scope link
       valid_lft forever preferred_lft forever
```

IP address and MAC address shown here corresponds to that of the host.

Host network

Use ip route command to inspect the route

```
/ # ip route
default via 172.31.80.1 dev eth0
169.254.169.254 dev eth0
172.17.0.0/16 dev docker0 scope link  src 172.17.0.1
172.31.80.0/20 dev eth0 scope link  src 172.31.85.201
```

Null network

- Application services that do not need any network connection
- Container will be completely isolated and thus safe from outside access

```
$ docker container run --rm -it --network none alpine:latest /bin/sh
```

verify that there is no eth0 network endpoint available

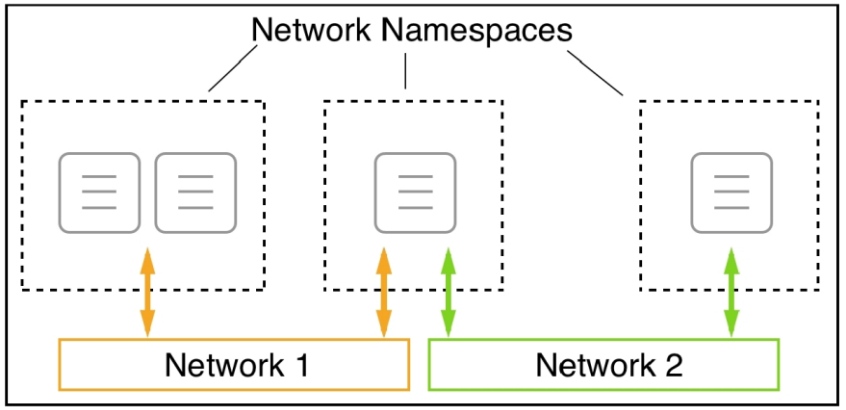
```
/ # ip addr show eth0
ip: can't find device 'eth0'
```

There is also no routing information available.

Running in an existing network namespace

- Docker creates a new network namespace for each container
- Network namespace of the container corresponds to the sandbox of the CNM
- Endpoint connects the container network namespace with actual network

We can create new containers that can be included in the network namespace of an existing container.



Running in an existing network namespace

Create a new bridge network:

```
$ docker network create --driver bridge test-net
```

Run a container attached to this network:

```
$ docker container run --name web -d --network test-net nginx:alpine
```

Run another container and attach it to the network of our web container:

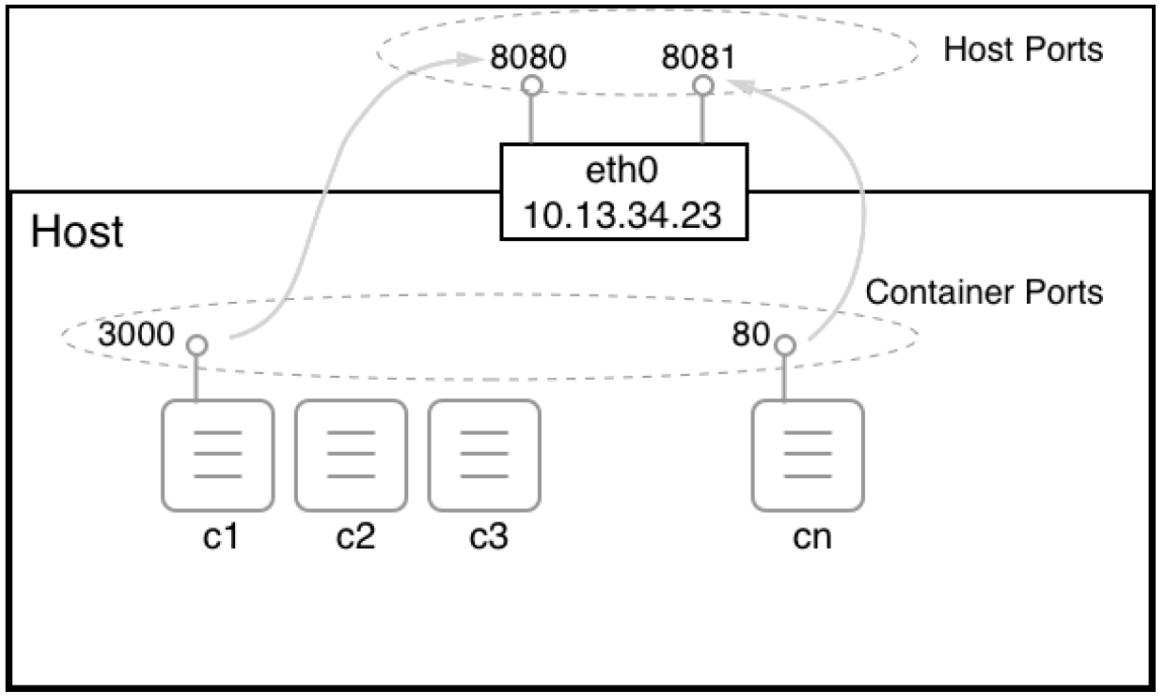
```
$ docker container run -it --rm --network container:web alpine:latest /bin/sh
```

Use the following command to prove that we are able to access nginx on localhost:

```
/ # wget -qO - localhost
```

Managing container ports

- We can create a gate by mapping a container port to an available port on the host
- Host ports exist completely independently and by default have nothing in common with container ports
- We can wire a container port with a free host port and funnel external traffic through this link



Managing container ports

We can map a container port to a specific host port. We can do this by using the -p parameter (or --publish).

```
$ docker container run --name web2 -p 8080:80 -d nginx:alpine
```

Module summary

In summary, in this module, you learned:

- How containers running on a single host can communicate with each other
- CNM, which defines the requirements of a container network
- How the bridge network functions in detail and what kind of information Docker provides us with about the networks
- How host and none network functions
- How to map container ports to Docker host

Thank you