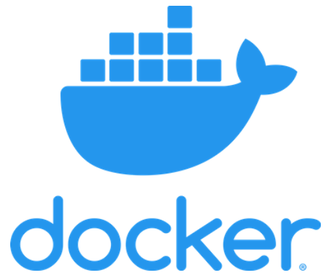
Docker Deep Dive

Chapter #9

Docker Networking



Docker

NETWORKING

www.networknuts.net

DOCKER NETWORKING

Components in Docker Networking

DOCKER NETWORKING

Building docker networks

DOCKER NETWORK

Containers inside the docker need to communicate over lot of different networks.

STRONG NETWORKING

Docker has solutions for container-to-container networks, & connecting to existing networks and VLANs.

CONTAINER NETWORK MODEL

Based on an open-source pluggable architecture called the Container Network Model (CNM).

Docker Networking

Docker networking: deep dive

Docker runs applications inside of containers, and these need to communicate over lots of different networks. This means Docker needs strong networking capabilities.

Docker has solutions for container-to-container networks, as well as connecting to existing networks and VLANs. The latter is important for containerized apps that need to communicate with functions and services on external systems such as VM’s and physicals.

Docker networking is based on an open-source pluggable architecture called the Container Network Model (CNM). libnetwork is Docker’s real-world implementation of the CNM, and it provides all of Docker’s core networking capabilities. Drivers plug in to libnetwork to provide specific network topologies.

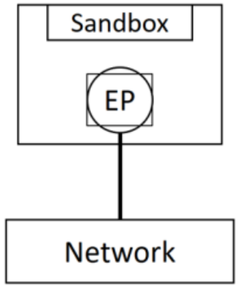
Docker networking comprises three major components:

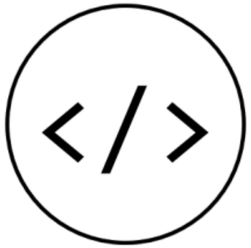
1. The Container Network Model (CNM)
2. libnetwork
3. Drivers

The **CNM** is the design specification. It outlines the fundamental building blocks of a Docker network.

libnetwork is a real-world implementation of the CNM, and is used by Docker. It's written in Go, and implements the core components outlined in the CNM.

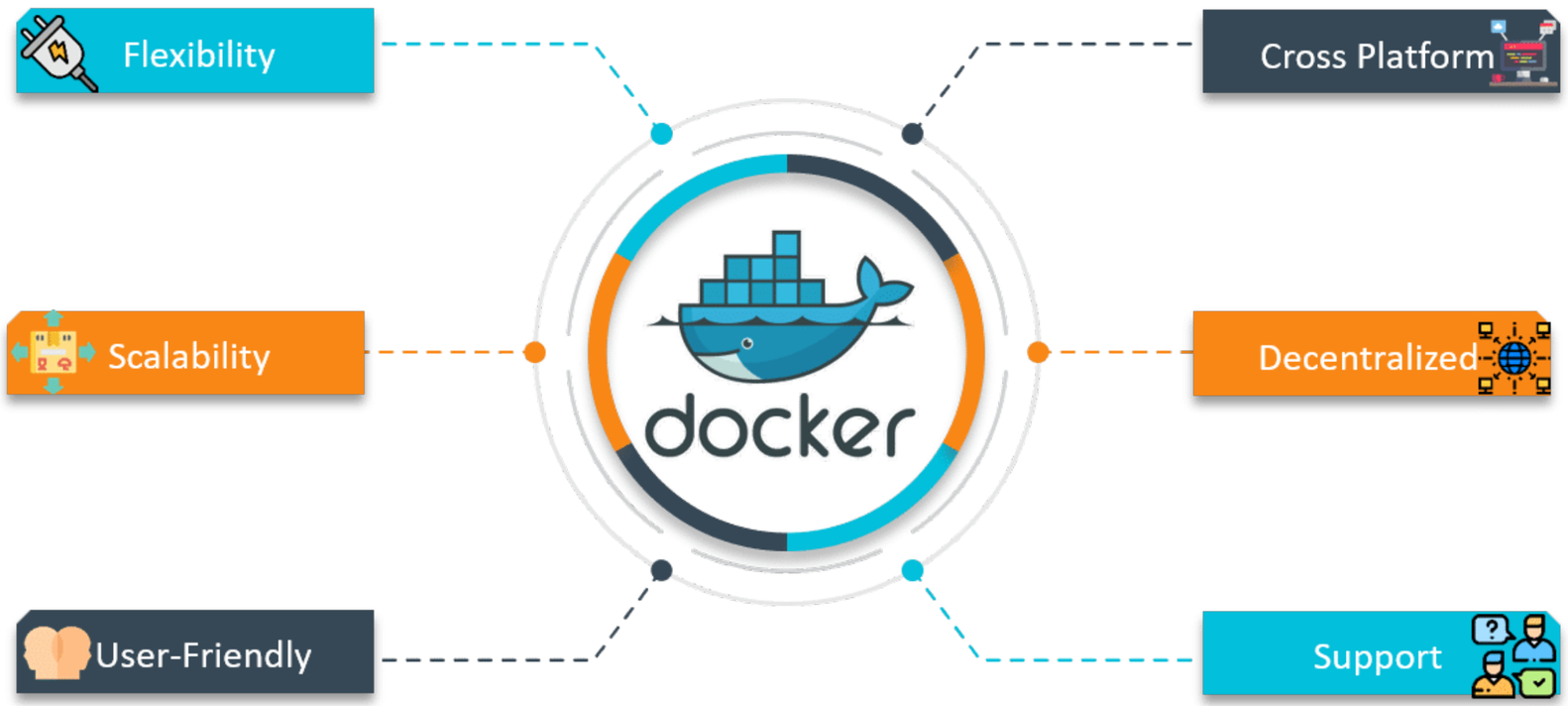
Drivers extend the model by implementing specific network topologies such as VXLAN-based overlay networks.





CNM	libnetwork	Drivers
Design (DNA)	Primitives Control & Management plane	Networks Data plane

Goals of Docker Networking



Container Network Model (CNM)

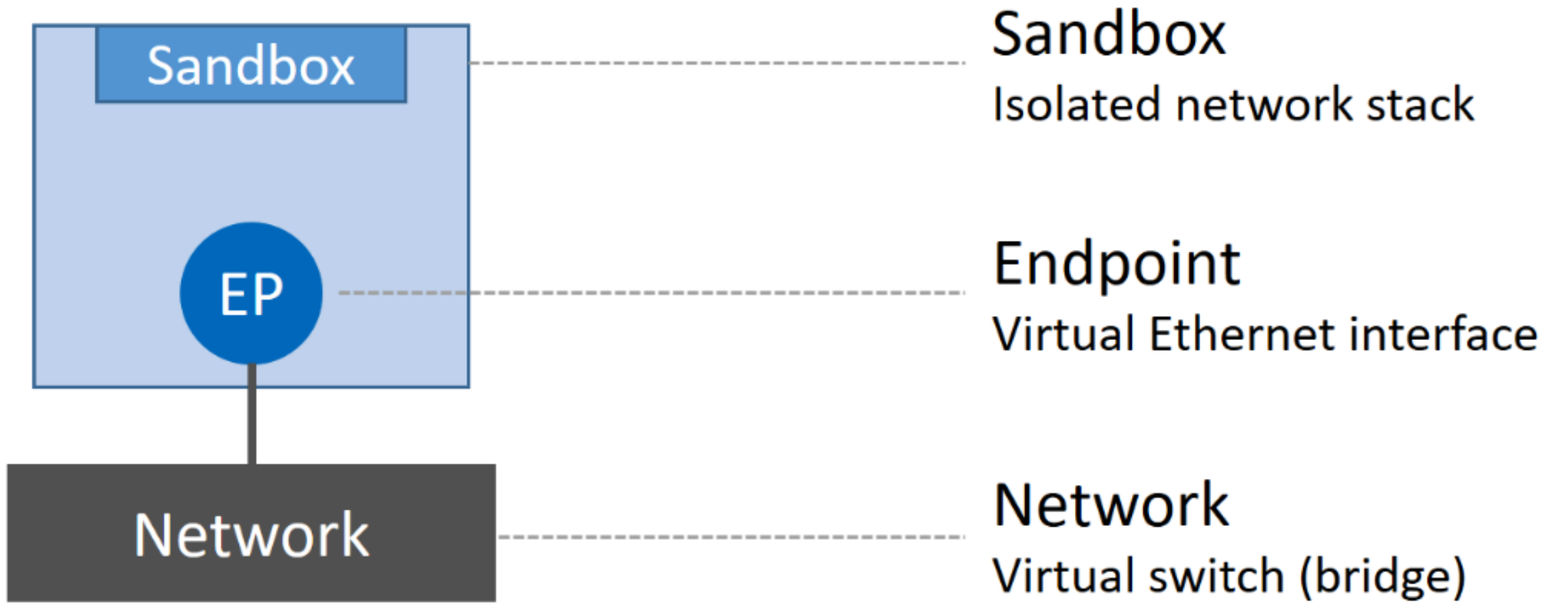
The design guide for Docker networking is the CNM. It outlines the fundamental building blocks of a Docker network. It defines three building blocks:

1. Sandboxes
2. Endpoints
3. Networks

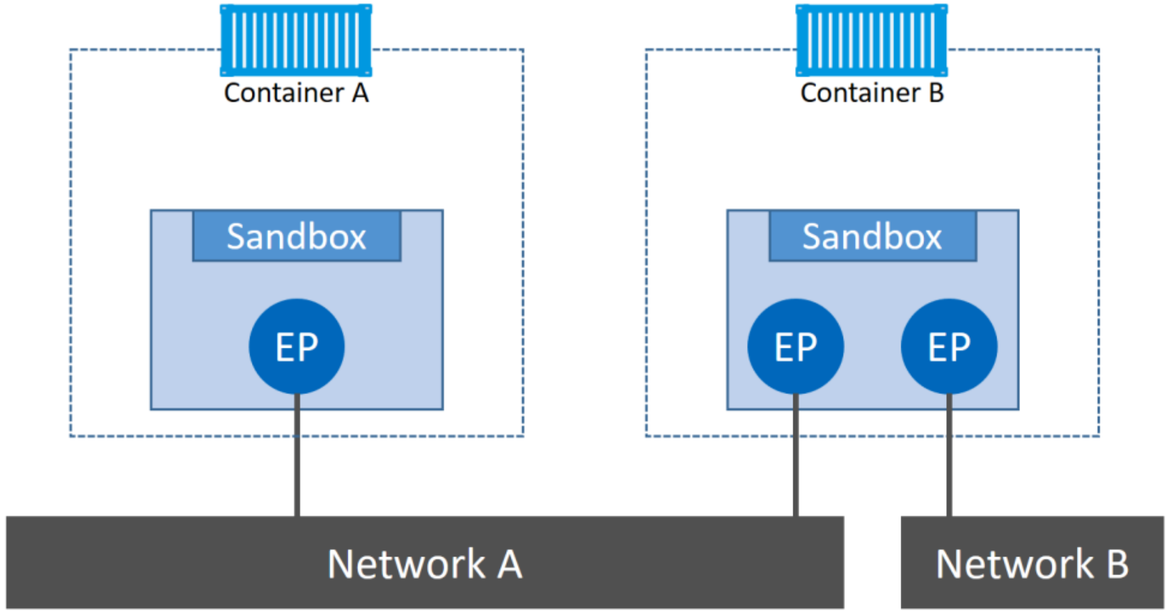
A **sandbox** is an isolated network stack. It includes; Ethernet interfaces, ports, routing tables, and DNS config.

Endpoints are virtual network interfaces (E.g. veth). Like normal network interfaces, they're responsible for making connections. In the case of the CNM, it's the job of the endpoint to connect a sandbox to a network.

Networks are a software implementation of an 802.1d bridge (more commonly known as a switch). As such, they group together, and isolate, a collection of endpoints that need to communicate.



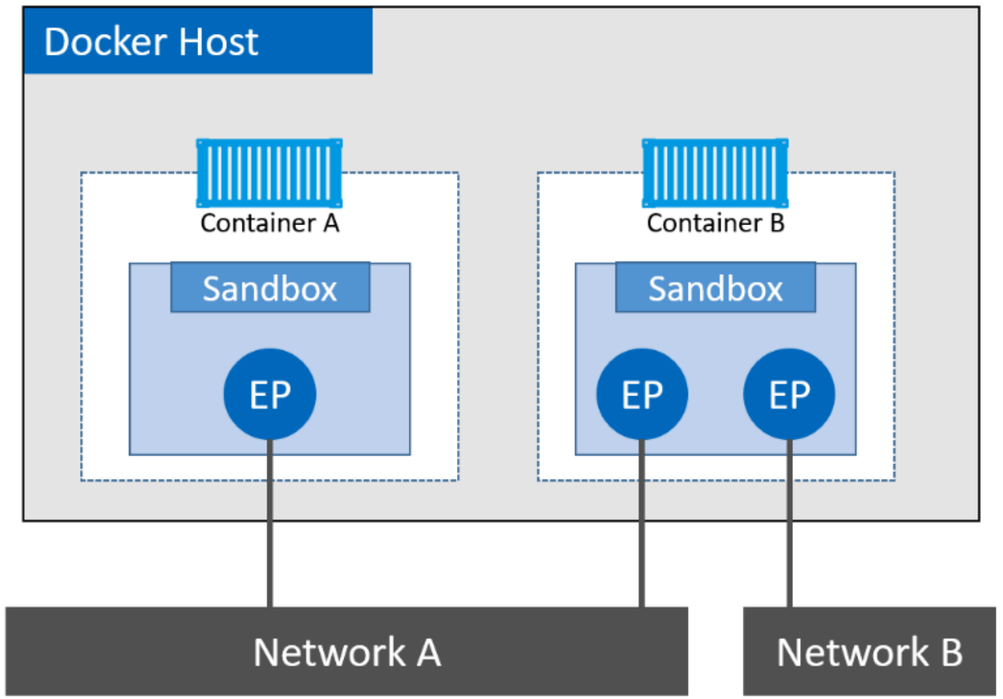
The atomic unit of scheduling in a Docker environment is the container, & as the name suggests, the Container Network Model is all about providing networking to containers.



Container A has a single interface (endpoint) and is connected to Network A. Container B has two interfaces (endpoints) and is connected to Network A and Network B. The containers will be able to communicate because they are both connected to Network A. However, the two endpoints in Container B cannot communicate with each other without the assistance of a layer 3 router.

Remember that endpoints behave like regular network adapters, meaning they can only be connected to a single network. Therefore, if a container needs connecting to multiple networks, it will need multiple endpoints.

Now lets add a Docker host. Although Container A and Container B are still running on the same host, their network stacks are completely isolated at the OS-level via the sandboxes.



Libnetwork

The CNM is the design doc, and libnetwork is the canonical implementation. It's open-source, written in Go, cross-platform (Linux and Windows), and used by Docker.

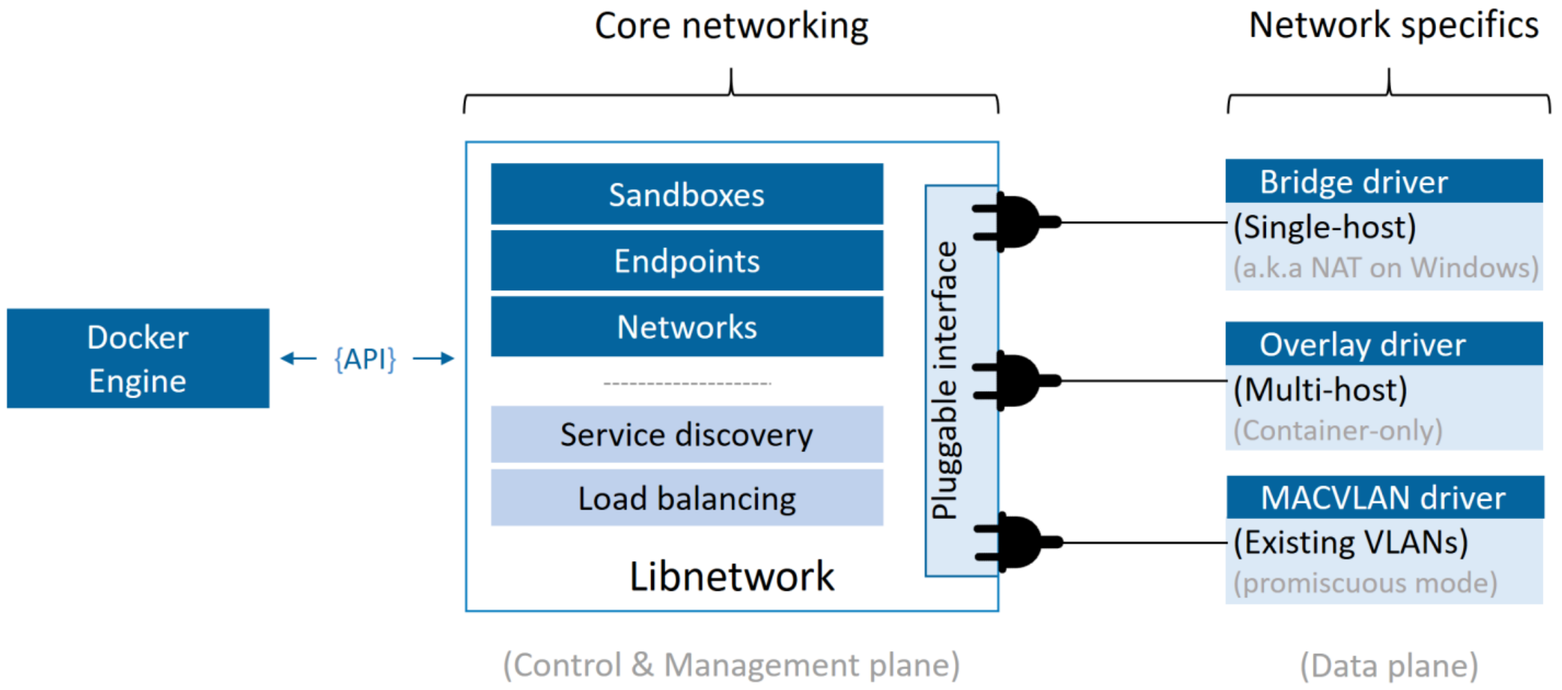
In the early days of Docker, all the networking code existed inside the daemon. This was a nightmare — the daemon became bloated, and it didn't follow the Unix principle of building modular tools that can work on their own, but also be easily composed into other projects.

As a result, it all got ripped out and refactored into an external library called **libnetwork**. Now, all of the core Docker networking code lives in libnetwork .

It implements all three of the components defined in the CNM. It also implements native service discovery, ingress-based container load balancing, & the network control plane and management plane functionality.

Drivers

If libnetwork implements the control plane and management plane functions, then drivers implement the data plane. For example, connectivity and isolation is all handled by drivers. So is the actual creation of network objects. The relationship is shown:



Docker ships with several built-in drivers, known as native drivers or local drivers. On Linux they include; bridge , overlay , and macvlan . On Windows they include; nat , overlay , transparent , and l2bridge.

3rd-parties can also write Docker network drivers. These are known as remote drivers, and examples include calico , contiv , kuryr , and weave .

Each driver is in charge of the actual creation and management of all resources on the networks it is responsible for. For example, an overlay network called “prod-fe-cuda” will be owned and managed by the overlay driver. This means the overlay driver will be invoked for the creation, management, and deletion of all resources on that network.

libnetwork allows multiple network drivers to be active at the same time. This means Docker environment can sport a wide range of heterogeneous networks.

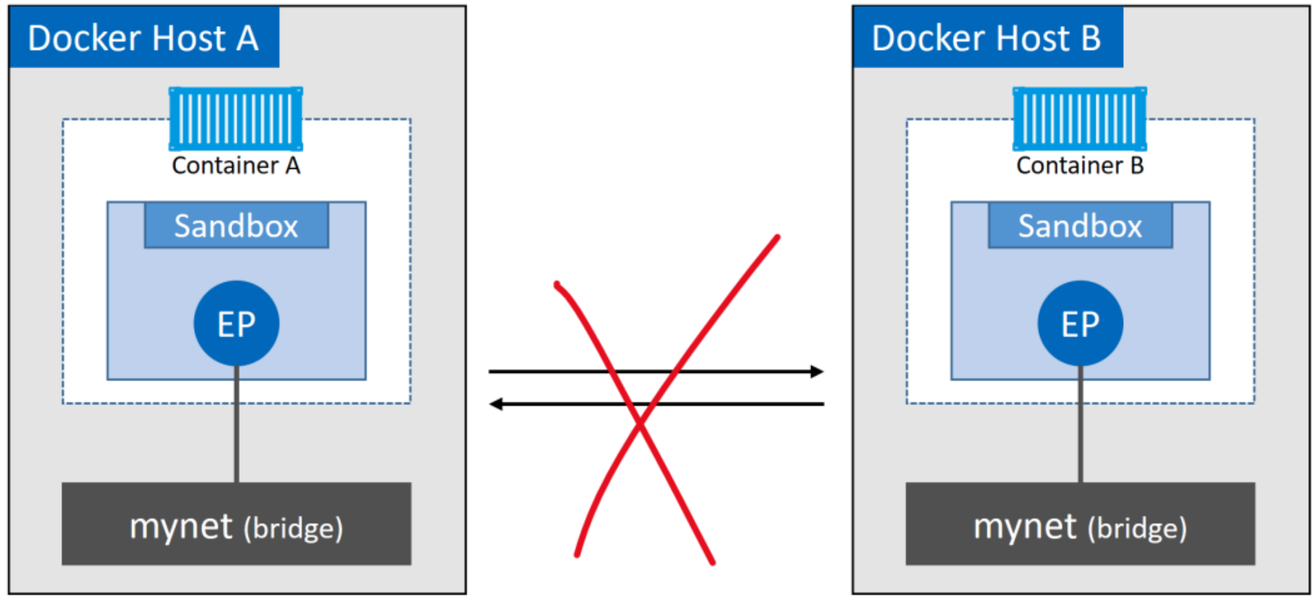
Single host bridge network

Simplest type of Docker network is the single-host bridge network.

1. **Single-host** tells us it only exists on a single Docker host and can only connect containers that are on the same host.
2. **Bridge** tells us that it's an implementation of an 802.1d bridge (layer 2 switch).

Docker on Linux creates single-host bridge networks with the built-in bridge driver, whereas Docker on Windows creates them using the built-in nat driver. For all intents and purposes, they work the same.

Here we have two Docker hosts with identical local bridge networks called “mynet”. Even though the networks are identical, they are independent isolated networks. This means the containers in the picture cannot communicate directly because they are on different networks.



Every Docker host gets a default single-host bridge network. On Linux it’s called “bridge”, & on Windows it’s called “nat”. By default, this is the network that all new containers will attach to unless you override it on the command line with the --network flag.

docker network ls can be used to view network

```
[root@ip-172-31-15-253 ~]#
[root@ip-172-31-15-253 ~]# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
715fdd3b83dc        bridge              bridge              local
97f2cc93662f        host                host                local
477547c8449b        none                null                local
[root@ip-172-31-15-253 ~]#
[root@ip-172-31-15-253 ~]#
```

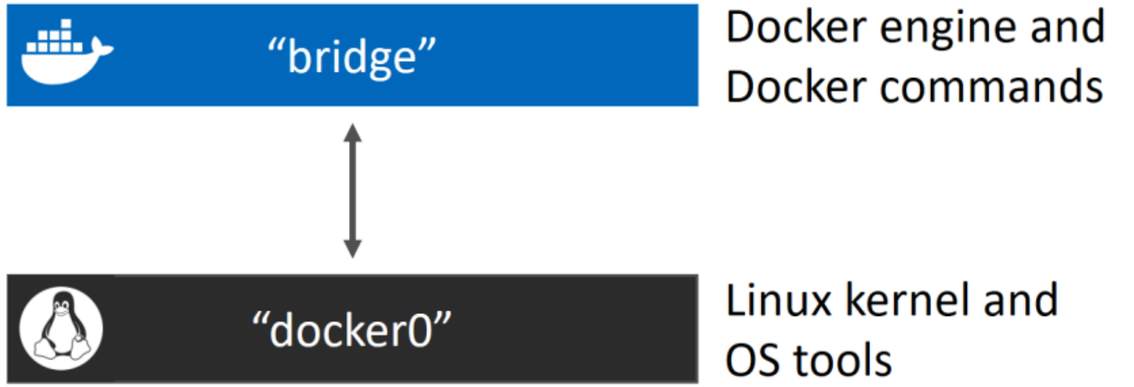
docker network inspect command can be used to get great information about network

```
[root@ip-172-31-15-253 ~]#
[root@ip-172-31-15-253 ~]# docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "715fdd3b83dc1fcb1d11b7b2d59367e412196abfad578f574
        "Created": "2020-06-10T06:11:32.549141039Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16"
                }
            ]
```

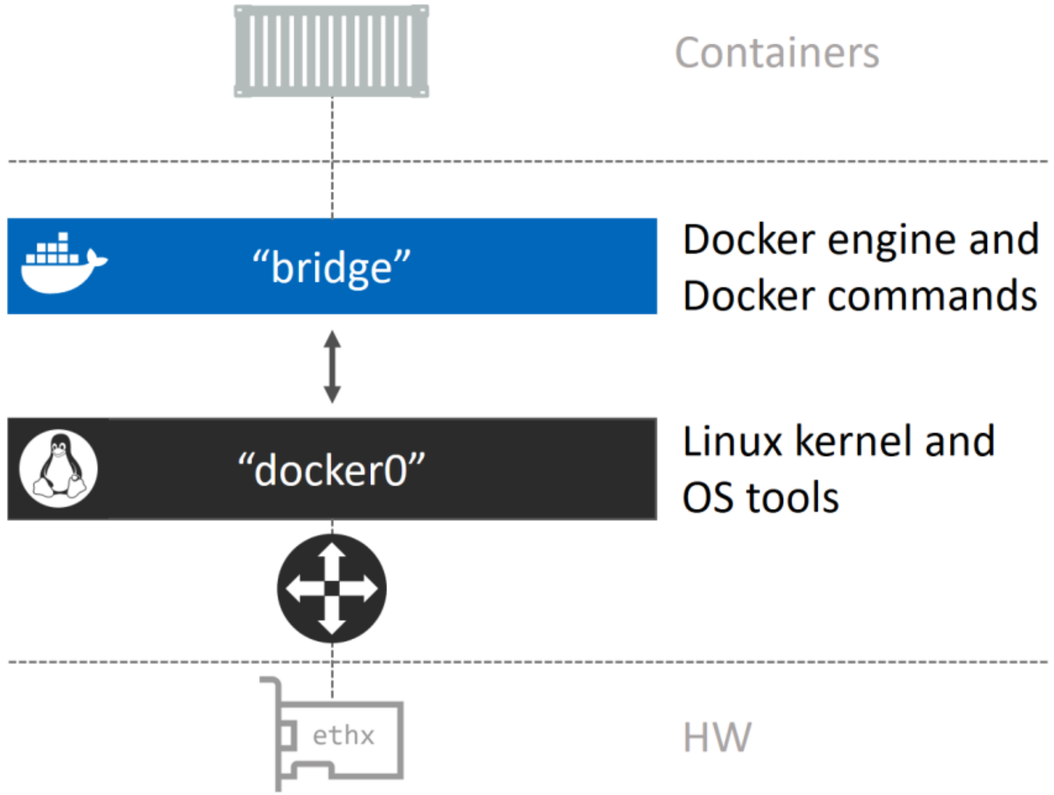
The default “bridge” network, on all Linux-based Docker hosts, maps to an underlying Linux bridge in the kernel called “docker0”. We can see this from the output of docker network inspect .

```
[root@ip-172-31-15-253 ~]# ip link show docker0
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mod
    link/ether 02:42:28:d9:7d:1c brd ff:ff:ff:ff:ff:ff
[root@ip-172-31-15-253 ~]#
[root@ip-172-31-15-253 ~]# docker network inspect bridge | grep bridge.name
            "com.docker.network.bridge.name": "docker0",
[root@ip-172-31-15-253 ~]#
[root@ip-172-31-15-253 ~]#
```

The relationship between Docker's default "bridge" network and the "docker0" bridge in the Linux kernel is shown.



Now extend the learning by adding containers at the top that plug into the “bridge” network. The “bridge” network maps to the “docker0” Linux bridge in the host’s kernel, which can be mapped back to an Ethernet interface on the host via port mappings.



Now use the docker network create command to create a new single-host bridge network called “localnet”.

docker network create -d bridge localnet

```
[root@ip-172-31-15-253 ~]#
[root@ip-172-31-15-253 ~]# docker network create -d bridge localnet
5943c0a16531970876a4ce9c2e1c183b6ed822db74450617a55efaf04177eb39
[root@ip-172-31-15-253 ~]#
[root@ip-172-31-15-253 ~]# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
715fdd3b83dc        bridge              bridge              local
97f2cc93662f        host                host                local
5943c0a16531        localnet            bridge              local
477547c8449b        none                null                local
[root@ip-172-31-15-253 ~]#
[root@ip-172-31-15-253 ~]#
```

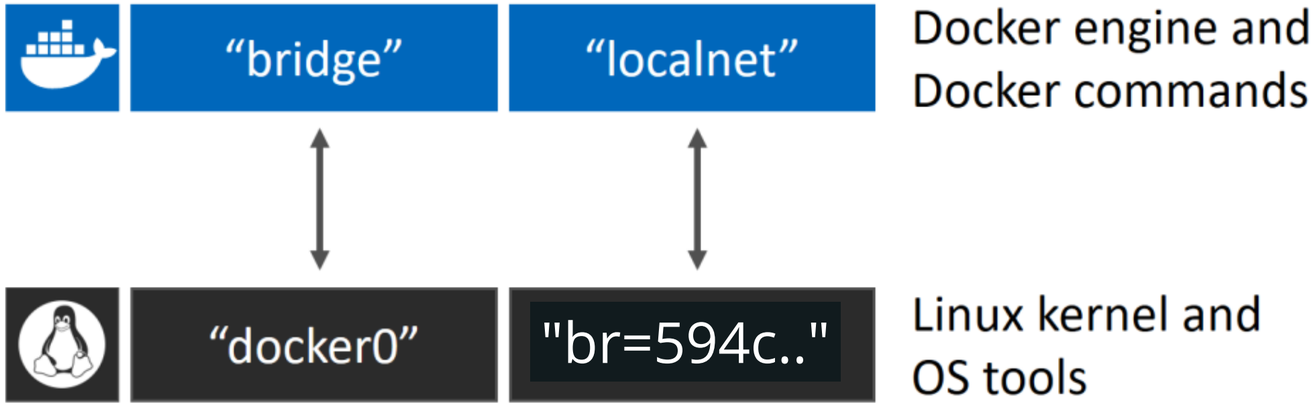
Now use the Linux brctl tool to look at the Linux bridges currently on the system. You may have to manually install the brctl binary.

 # yum install brctl-utils

```
[root@ip-172-31-15-253 ~]# yum install bridge-utils
Loaded plugins: priorities, update-motd, upgrade-helper
amzn-main
amzn-updates
Resolving Dependencies
--> Running transaction check
---> Package bridge-utils.x86_64 0:1.2-10.7.amzn1 will be installed
--> Finished Dependency Resolution

Dependencies Resolved

================================================================================
 Package                        Arch                       Version
================================================================================
Installing:
 bridge-utils                   x86_64                     1.2-10.7.amzn1

Transaction Summary
================================================================================
Install  1 Package
```

```
[root@ip-172-31-15-253 ~]# brctl show
bridge name     bridge id               STP enabled     interfaces
br-5943c0a16531         8000.02427f7acb8a       no
docker0         8000.024228d97d1c       no
[root@ip-172-31-15-253 ~]#
[root@ip-172-31-15-253 ~]#
```

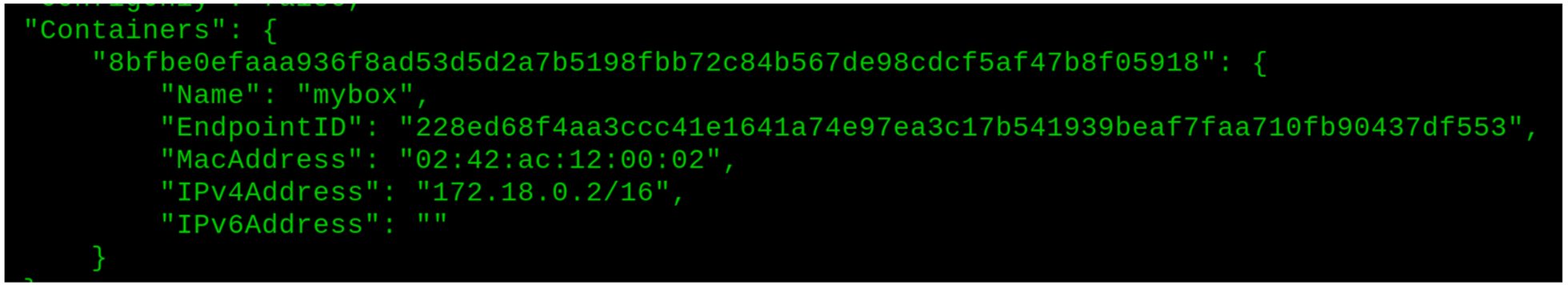
The output shows two bridges. The first line is the “docker0” bridge that we already know about. This relates to the default “bridge” network in Docker. The second bridge (br-5943c..) relates to the new **localnet** Docker bridge network. Neither of them have spanning tree enabled, and neither have any devices connected (interfaces column). At this point, the bridge configuration on the host looks like:



Now create a new container and attach it to the new localnet bridge network.

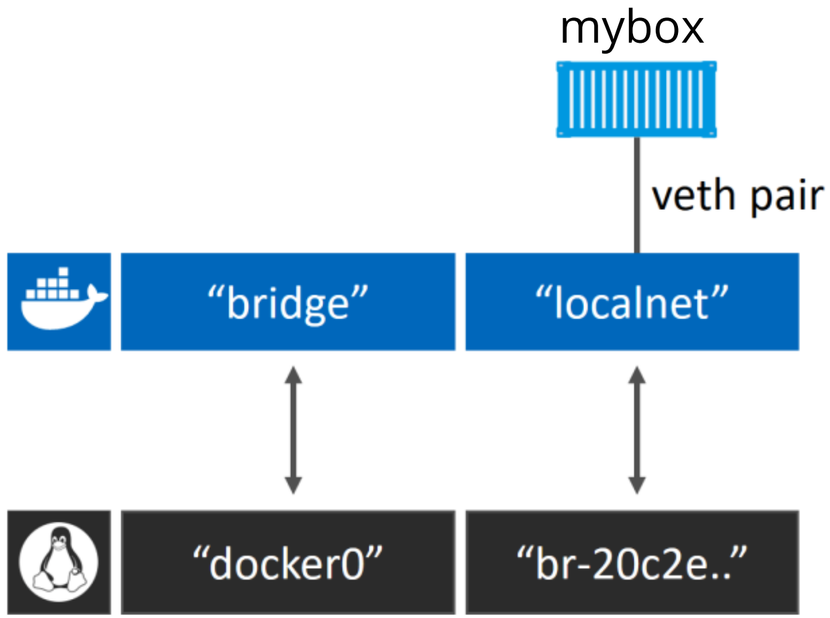
 # docker container run -d --name=mybox --network localnet alpine sleep 1d

This container will now be on the localnet network. Confirm this with a **docker network inspect localnet**



If we run the Linux **brctl show** command again, we'll see mybox interface attached to the br-5943c... bridge.

```
[root@ip-172-31-15-253 ~]#
[root@ip-172-31-15-253 ~]# brctl show
bridge name     bridge id               STP enabled     interfaces
br-5943c0a16531         8000.02427f7acb8a       no              veth31635fc
docker0         8000.024228d97d1c       no
[root@ip-172-31-15-253 ~]#
[root@ip-172-31-15-253 ~]#
```

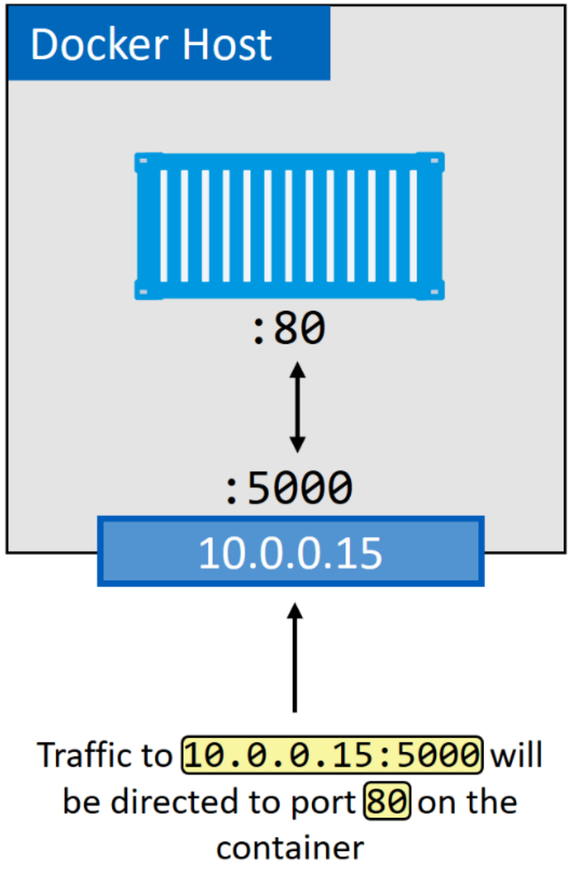


If we add another new container to the same network, it should be able to ping the “mybox” container by name. This is because all new containers are registered with the embedded Docker DNS service so can resolve the names of all other containers on the same network.

```
[root@ip-172-31-15-253 ~]#
[root@ip-172-31-15-253 ~]# docker container run -it --name=mybox2 --network localnet alpine sh
/ # ping mybox
PING mybox (172.18.0.2): 56 data bytes
64 bytes from 172.18.0.2: seq=0 ttl=255 time=0.082 ms
64 bytes from 172.18.0.2: seq=1 ttl=255 time=0.071 ms
64 bytes from 172.18.0.2: seq=2 ttl=255 time=0.071 ms
^C
--- mybox ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.071/0.074/0.082 ms
/ #
```

Port mappings

Port mappings let you map a container port to a port on the Docker host. Any traffic hitting the Docker host on the configured port will be directed to the container.



Application running in the container is operating on port 80. This is mapped to port 5000 on the host's 10.0.0.15 interface. The end result is all traffic hitting the host on 10.0.0.15:5000 being redirected to the container on port 80.

Run a new web server container and map port 80 on the container to port 5000 on the Docker host.

```
[root@ip-172-31-15-253 ~]# docker container run -d --name=webserver \
> --network localnet \
> --publish 5000:80 \
> lovelearnlinux/webserver:v1
Unable to find image 'lovelearnlinux/webserver:v1' locally
v1: Pulling from lovelearnlinux/webserver
8a29a15cefae: Pull complete
eabe6c48556e: Pull complete
dd4435497983: Pull complete
Digest: sha256:ac9ab88604c58e12ada773e284f2f19d7e7a1564f741b2a0cc7f120ad8740b88
Status: Downloaded newer image for lovelearnlinux/webserver:v1
2d27eeb5d802562d3ffa08121b260876d4c775a5ef4e5edd22af72356b763e61
[root@ip-172-31-15-253 ~]#
[root@ip-172-31-15-253 ~]# docker port webserver
80/tcp -> 0.0.0.0:5000
[root@ip-172-31-15-253 ~]#
```

Test the configuration by pointing a web browser to port 5000 on the Docker host.

```
[root@ip-172-31-15-253 ~]# curl http://172.31.15.253:5000
<html>
<title>
Network Nuts - Kubernetes training
</title>
<h1>
welcome to networknuts
</h1>
<h2>
you are learning kubernetes<br>
lets love learn linux
</h2>
</html>

[root@ip-172-31-15-253 ~]#
```

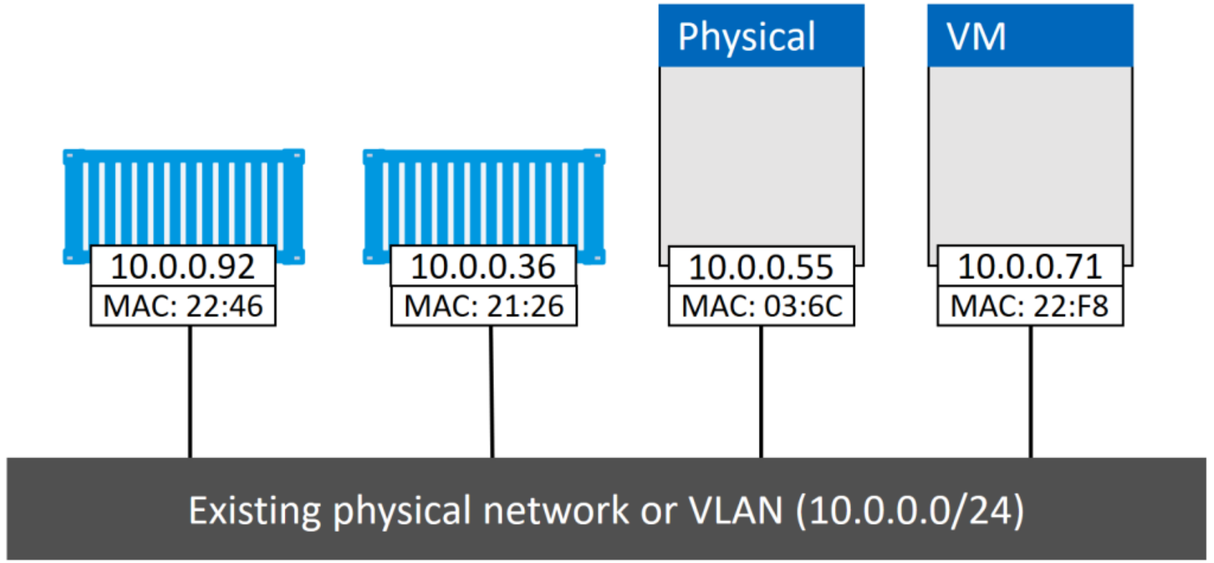
Mapping ports like this works, but it’s clunky and doesn’t scale. For example, only a single container can bind to any port on the host. This means no other containers will be able to use port 5000 on the host we’re running the APACHE container on.

This is one of the reason’s that single-host bridge networks are only useful for local development and very small applications.

Connecting to existing networks

The ability to connect containerized apps to external systems and physical networks is vital. A common example is a partially containerized app, it will need a way to communicate with the non-containerized parts still running on existing physical networks and VLANs.

The built-in MACVLAN driver was created with this in mind. It makes containers talk on the existing physical networks by giving each one its own MAC and IP addresses.

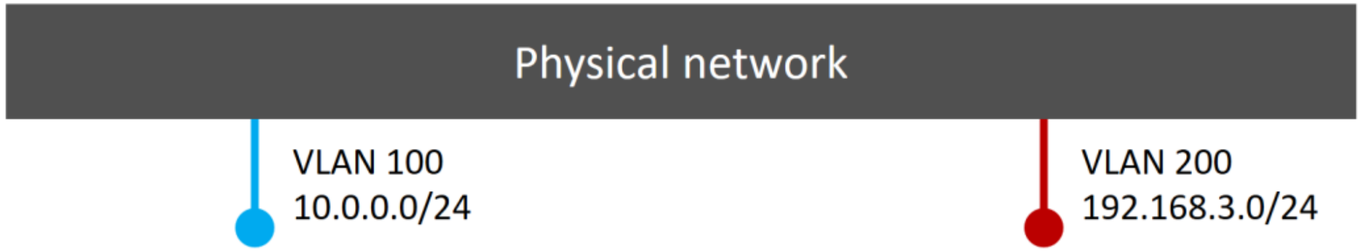


MACVLAN performance is good as it doesn't require port mappings or additional bridges — you connect the container interface through to the hosts interface (or a sub-interface).

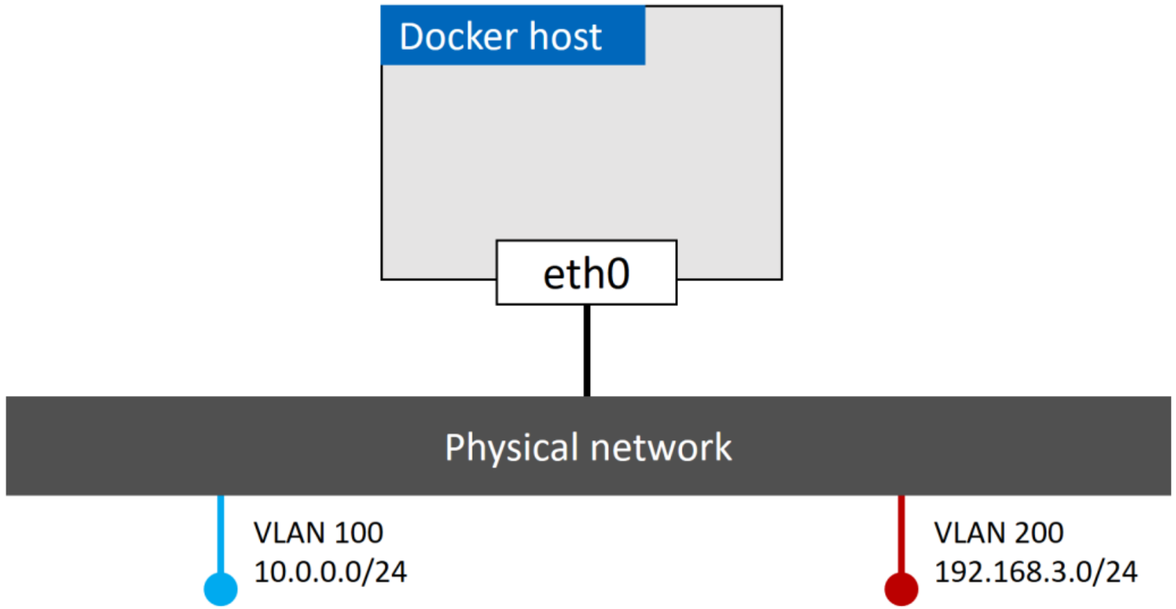
However, on the negative side, it requires the host NIC to be in promiscuous mode, which isn't allowed on most public cloud platforms. So MACVLAN is great for your corporate data center networks, but it won't work in the public cloud.

Assume we have an existing physical network with two VLANS:

- VLAN 100: 10.0.0.0/24
- VLAN 200: 192.168.3.0/24



Now if we add a Docker host and connect it to the network.



We then have a requirement for a container (app service) to be plumbed into VLAN 100. To do this, we create a new Docker network with the macvlan driver. However, the macvlan driver needs us to tell it a few things about the network we're going to associate it with. Things like:

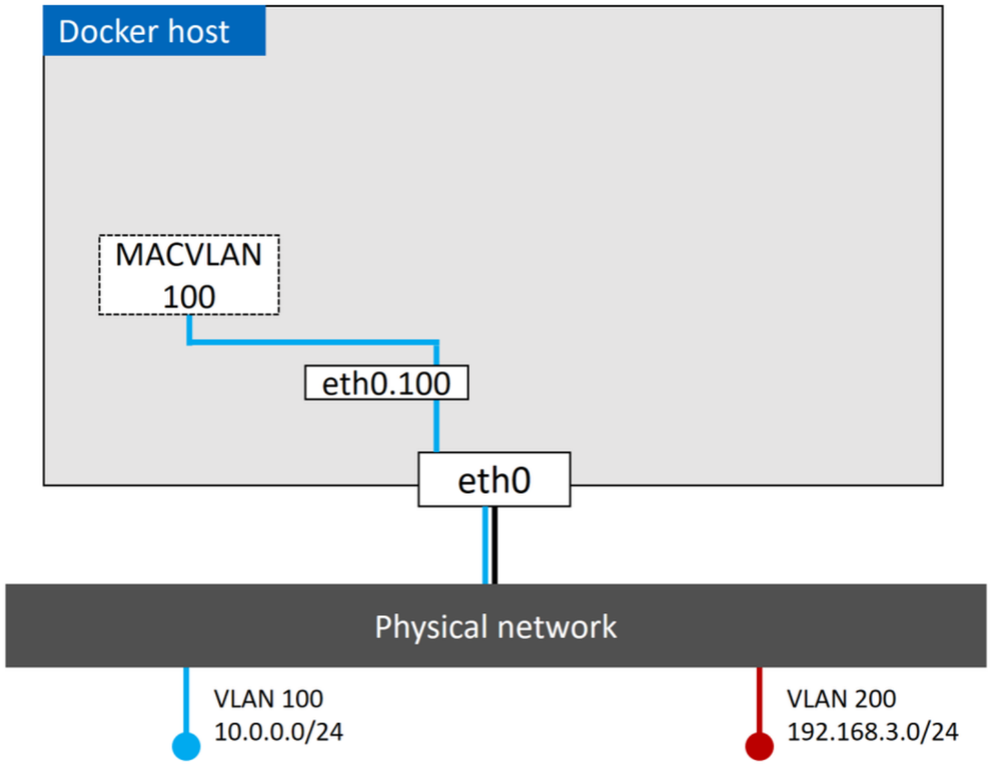
- Subnet info
- Gateway
- Range of IP's it can assign to containers
- Which interface or sub-interface on the host to use

So lets create a new MACVLAN network called "macvlan100" that will connect containers to VLAN 100.

```
[root@docker ~]# docker network create -d macvlan \
> --subnet=10.0.0.0/24 \
> --ip-range=10.0.0.0/25 \
> --gateway=10.0.0.1 \
> -o parent=enp0s3.100 \
> macvlan100
eee202c82503c68359db242af66d680fee36d81d2f633eaff8000f54606785f5
[root@docker ~]#
[root@docker ~]#
```

This will create the “macvlan100” network and the enp0s3.100 sub-interface. The config now looks like this.

Most cloud providers block macvlan networking. You may need physical access to your networking equipment.

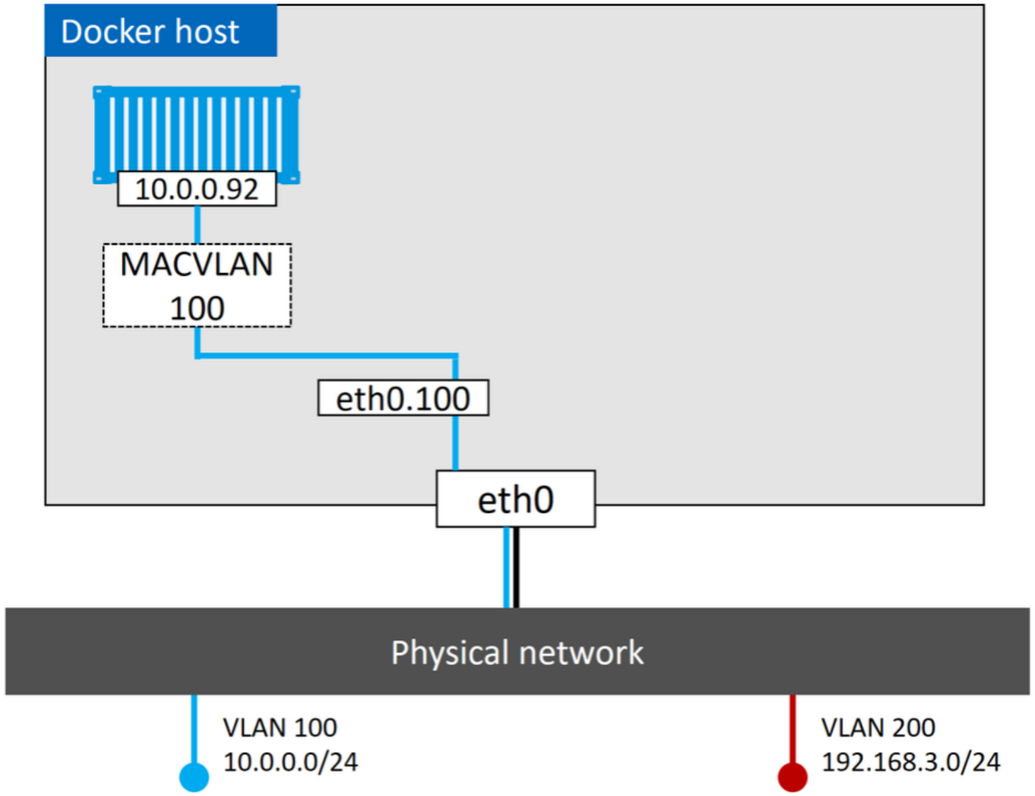


MACVLAN uses standard Linux sub-interfaces, and you have to tag them with the ID of the VLAN they will connect to. In this example we're connecting to VLAN 100, so we tag the sub-interface with .100 (enp0s3.100).

Now let’s deploy one container with the following command.

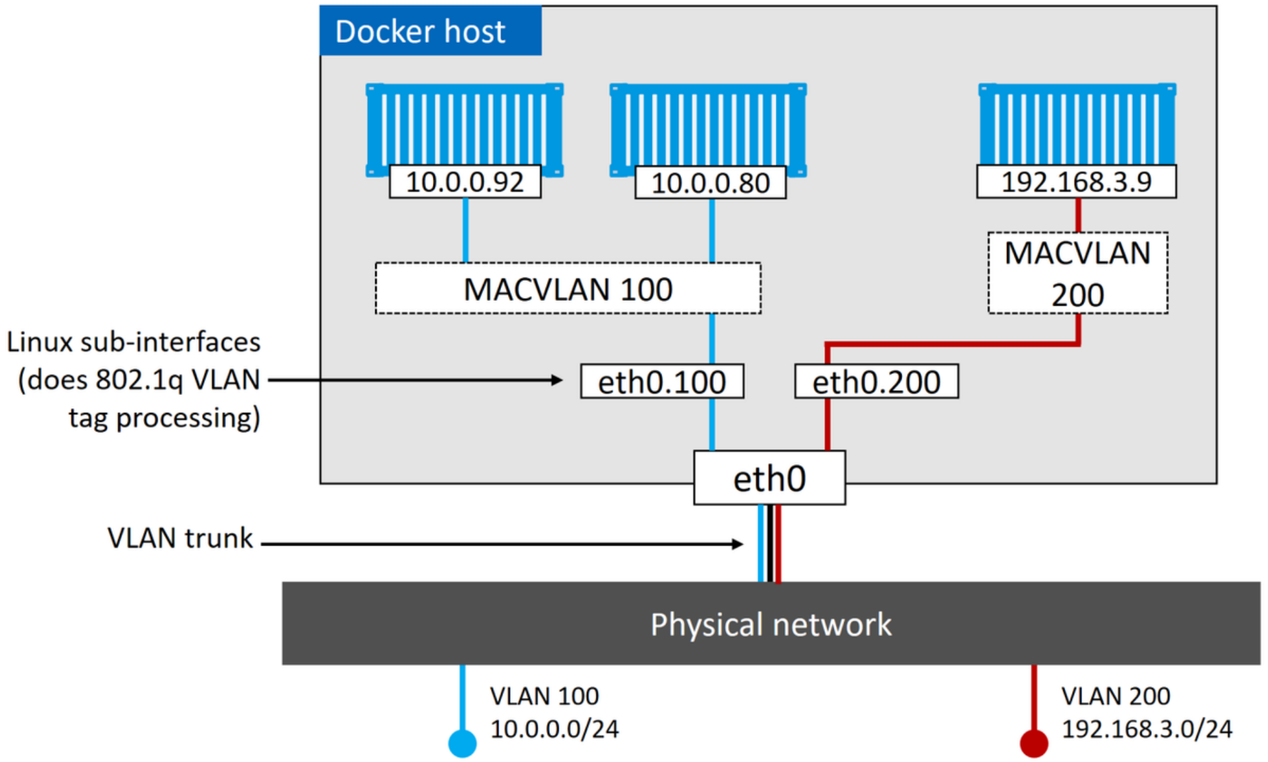
```
[root@docker ~]#
[root@docker ~]# docker container run -d --name=myboxmac \
> --network macvlan100 \
> alpine sleep 1d
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
df20fa9351a1: Pull complete
Digest: sha256:185518070891758909c9f839cf4ca393ee977ac378609f700f
Status: Downloaded newer image for alpine:latest
5fd475f17f996d6236f94fbe7dc59a1290f27713130249b7cec9294ac36f6ed1
[root@docker ~]#
[root@docker ~]#
```

“myboxmac” container will be able to ping and communicate with any other systems on VLAN 100.



At this point, we've got a MACVLAN network and used it to connect a new container to an existing VLAN.

The Docker MACVLAN driver supports VLAN trunking. This means we can create multiple MACVLAN networks and connect containers on the same Docker host.



WHAT WE LEARNED

Docker networking and defines the three major constructs that are used to build Docker networks — sandboxes, endpoints, and networks.

libnetwork is the open-source library, written in Go, that implements the CNM.

Single-host bridge networks are the most basic type of Docker network and are suitable for local development and very small applications. They do not scale, and they require port mappings if you want to publish your services outside of the network.

The macvlan driver (transparent on Windows) allows you to connect containers to existing physical networks and VLANs by giving them their own MAC and IP addresses. Unfortunately, they require promiscuous on the host NIC, meaning they won’t work in the public cloud.