Understanding Docker Networking

16 minute read Updated: May 5, 2021

Ashish Choudhary

Docker is the de facto model for building and running containers at scale in most enterprise organizations today. At a very high level, Docker is a combination of CLI and a daemon process that solves common software problems like installing, publishing, removing, and managing containers. It's perfect for microservices, where you have many services handling a typical business functionality; Docker makes the packaging easier, enabling you to encapsulate those services in containers.

Once the application is inside a container, it's easier to scale and even runs on different cloud platforms, like AWS, GCP, and Azure. In this article, let's focus on the **networking aspect of Docker**.

What Is a Docker Network?

Networking is about communication among processes, and Docker's networking is no different. Docker networking is primarily used to establish communication between Docker containers and the outside world via the host machine where the Docker daemon is running.

Docker supports different types of networks, each fit for certain use cases. We'll be exploring the network drivers supported by Docker in general, along with some coding examples.

Docker networking differs from virtual machine (VM) or physical machine networking in a few ways:

1. Virtual machines are more flexible in some ways as they can support configurations like **NAT and host networking**. Docker typically uses a bridge network, and while it can support host networking, that option is **only available on Linux**.
2. When using Docker containers, network isolation is achieved using a network namespace, not an entirely separate networking stack.
3. You can run hundreds of containers on a single-node Docker host, so it's required that the host can support networking at this scale. VMs usually don't run into these network limits as they typically run fewer processes per VM.

What Are Docker Network Drivers?

Docker handles communication between containers by creating a default bridge network, so you often don't have to deal with networking and can instead focus on creating and running containers. This default bridge network works in most cases, but it's not the only option you have.

Docker allows you to create three different types of network drivers out-of-the-box: bridge, host, and none. However, they may not fit every use case, so we'll also explore user-defined networks such as `overlay` and `macvlan`. Let's take a closer look at each one.

The Bridge Driver

This is the default. Whenever you start Docker, a bridge network gets created and all newly started containers will connect automatically to the default bridge network.

You can use this whenever you want your containers running in isolation to connect and communicate with each other. Since containers run in isolation, the bridge network solves the port conflict problem. Containers running in the same bridge network can communicate with each other, and Docker **uses iptables** on the host machine to prevent access outside of the bridge.

Let’s look at some examples of how a bridge network driver works.

1. Check the available network by running the `docker network ls` command
2. Start two busybox containers named `busybox1` and `busybox2` in detached mode by passing the `-dit` flag.

```
$ docker network ls
NETWORK ID     NAME     DRIVER   SCOPE
5077a7b25ae6   bridge   bridge   local
7e25f334b07f   host     host     local
475e50be0fe0   none     null     local
```

```
docker run -dit --name busybox1 busybox /bin/sh
docker run -dit --name busybox2 busybox /bin/sh
```

3. Run the `docker ps` command to verify that containers are up and running.

```
$ docker ps
CONTAINER ID   IMAGE     COMMAND     CREATED          STATUS          P
9e6464e82c4c   busybox   "/bin/sh"   5 seconds ago    Up 5 seconds
7fea14032748   busybox   "/bin/sh"   26 seconds ago   Up 26 seconds
```

4. Verify that the containers are attached to the bridge network.

```
$ docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "5077a7b25ae67abd46cff0fde160303476c8a9e2e1d52ad01ba2b4bf
        "Created": "2021-03-05T03:25:58.232446444Z",
        "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": {
            "7fea140327487b57c3cf31d7502cfaf701e4ea4314621f0c726309e396
                "Name": "busybox1",
                "EndpointID": "05f216032784786c3315e30b3d54d50a25c1efc7
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            },
            "9e6464e82c4ca647b9fb60a85ca25e71370330982ea497d51c1238d073
                "Name": "busybox2",
                "EndpointID": "3dcc24e927246c44a2063b5be30b5f5e1787dcd4
                "MacAddress": "02:42:ac:11:00:03",
                "IPv4Address": "172.17.0.3/16",
                "IPv6Address": ""
            }
        },
        "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": {}
    }
]
```

5. Under the container's key, you can observe that two containers (**`busybox1`** and **`busybox2`**) are listed with information about IP addresses. Since

containers are running in the background, attach to the **`busybox1`** container and try to ping to **`busybox2`** with its IP address.

```
$ docker attach busybox1
/ # whoami
root
/ # hostname -i
172.17.0.2
/ # ping 172.17.0.3
PING 172.17.0.3 (172.17.0.3): 56 data bytes
64 bytes from 172.17.0.3: seq=0 ttl=64 time=2.083 ms
64 bytes from 172.17.0.3: seq=1 ttl=64 time=0.144 ms
/ # ping busybox2
ping: bad address 'busybox2'
```

6. Observe that the ping works by passing the IP address of **`busybox2`** but fails when the container name is passed instead.

The downside with the bridge driver is that it's not recommended for production; the containers communicate via IP address instead of automatic service discovery to resolve an IP address to the container name. Every time you run a container, a different IP address gets assigned to it. It may work well for local development or CI/CD, but it's definitely not a sustainable approach for applications running in production.

Another reason not to use it in production is that it will allow unrelated containers to communicate with each other, which could be a security risk. I'll cover how you can create custom bridge networks later.

The Host Driver

As the name suggests, host drivers use the networking provided by the host machine. And it removes network isolation between the container and the host machine where Docker is running. For example, If you run a container that binds to port 80 and uses host networking, the container's application is available on port 80 on the host's IP address. You can use the host network if you don't want to rely on Docker's networking but instead rely on the host machine networking.

One limitation with the host driver is that it doesn't work on Docker desktop: you need a Linux host to use it. This article focuses on Docker desktop, but I'll show you the commands required to work with the Linux host.

The following command will start an Nginx image and listen to port 80 on the host machine:

```
docker run --rm -d --network host --name my_nginx nginx
```

You can access Nginx by hitting the `http://localhost:80/ url`.

The downside with the host network is that you can't run multiple containers on the same host having the same port. Ports are shared by all containers on the host machine network.

The None Driver

The none network driver does not attach containers to any network. Containers do not access the external network or communicate with other containers. You can use it when you want to disable the networking on a container.

The Overlay Driver

The Overlay driver is for multi-host network communication, as with Docker Swarm or Kubernetes. It allows containers across the host to communicate with each other without worrying about the setup. Think of an overlay network as a distributed virtualized network that's built on top of an existing computer network.

To create an overlay network for Docker Swarm services, use the following command:

```
docker network create -d overlay my-overlay-network
```

To create an overlay network so that standalone containers can communicate with each other, use this command:

```
docker network create -d overlay --attachable my-attachable-overlay
```

The Macvlan Driver

This driver connects Docker containers directly to the physical host network. As per the Docker documentation:

> *"Macvlan networks allow you to assign a MAC address to a container, making it appear as a physical device on your network. The Docker daemon routes traffic to containers by their MAC addresses. Using the `macvlan` driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network, rather than routed through the Docker host's network stack."*

Macvlan networks are best for legacy applications that need to be modernized by containerizing them and running them on the cloud because they need to be attached to a physical network for performance reasons. A macvlan network is also not supported on Docker desktop for macOS.

Basic Docker Networking Commands

To see which commands list, create, connect, disconnect, inspect, or remove a Docker network, use the `docker network help` command.

```
$ docker network help
```

```
Usage:  docker network COMMAND

Manage networks

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
```

Let's run through some examples of each command, starting with **`docker network connect`** .

Connecting a Container to a Network

Let's try to connect a container to the **`mynetwork`** we have created. First, we would need a running container that can connect to **`mynetwork`** .

```
$ docker run -it ubuntu bash
root@0f8d7a833f42:/#
```

Now we have an Ubuntu Linux image and started a login shell as root inside it. We have the container running in interactive mode with the help of the **`-it`** flags.

```
$ docker ps
CONTAINER ID   IMAGE    COMMAND   CREATED         STATUS         PORTS
0f8d7a833f42   ubuntu   "bash"    9 seconds ago   Up 7 seconds
```

Run the **`docker network connect 0f8d7a833f42`** command to connect the container named **`wizardly_greider`** with **`mynetwork`** . To verify that this container is connected to **`mynetwork`** , use the **`docker inspect`** command.

```
"mynetwork": {
                    "IPAMConfig": {},
                    "Links": null,
                    "Aliases": [
                        "0f8d7a833f42"
                    ],
                    "NetworkID": "97a158252c995d3632560852c62bd140984769c8714b1f990c8133a5c8ae65d3",
                    "EndpointID": "db21f395ca781523c115706b11063ebe879cf5ef246c24fd128fe621a582cade",
                    "Gateway": "172.20.0.1",
                    "IPAddress": "172.20.0.2",
                    "IPPrefixLen": 16,
                    "IPv6Gateway": "",
                    "GlobalIPv6Address": "",
                    "GlobalIPv6PrefixLen": 0,
                    "MacAddress": "02:42:ac:14:00:02",
                    "DriverOpts": {}
                }
```

Creating a Network

You can use **`docker network create mynetwork`** to create a Docker network. Here, we've created a network named **`mynetwork`** . Let's run **`docker network ls`** to verify that the network is created successfully.

```
$ docker network ls
NETWORK ID     NAME        DRIVER    SCOPE
b995772ac197   bridge      bridge    local
7e25f334b07f   host        host      local
97a158252c99   mynetwork   bridge    local
475e50be0fe0   none        null      local
```

Now we have a new custom network named **`mynetwork`** , and its type is bridge.

Disconnecting a Container from the Network

This command disconnects a Docker container from the custom **`mynetwork`** :

```
docker network disconnect mynetwork 0f8d7a833f42
```

Inspecting the Network

The **`docker network inspect`** command displays detailed information on one or more networks.

```
$ docker network inspect mynetwork
[
    {
        "Name": "mynetwork",
        "Id": "97a158252c995d3632560852c62bd140984769c8714b1f990c8133a5
        "Created": "2021-03-02T17:36:30.090173896Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.20.0.0/16",
                    "Gateway": "172.20.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "0f8d7a833f4283202e905e621e6fd5b29a8e3d4eccc6be6ea0f209f5cb
                "Name": "wizardly_greider",
                "EndpointID": "db21f395ca781523c115706b11063ebe879cf5ef
                "MacAddress": "02:42:ac:14:00:02",
                "IPv4Address": "172.20.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {}
    }
]
```

List Available Networks

Docker installation comes with three networks: none, bridge, and host. You can verify this by running the command `docker network ls`:

```
docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
b995772ac197   bridge    bridge    local
7e25f334b07f   host      host      local
475e50be0fe0   none      null      local
```

Removing a Network

The following are the Docker commands to remove a specific or all available networks:

```
$ docker network rm mynetwork
mynetwork
$ docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
b995772ac197   bridge    bridge    local
7e25f334b07f   host      host      local
475e50be0fe0   none      null      local


$ docker network prune
WARNING! This will remove all custom networks not used by at least one
Are you sure you want to continue? [y/N]
```

Public Networking

Let's talk about how to publish a container port and IP addresses to the outside world. When you start a container using the **`docker run`** command, none of its ports are exposed. Your Docker container can connect to the outside world, but the outside world cannot connect to the container. To make the ports accessible for external use or with other containers not on the same network, you will have to use the **`-P`** (publish all available ports) or **`-p`** (publish specific ports) flag.

For example, here we've mapped the TCP port 80 of the container to port 8080 on the Docker host:

```
docker run -it --rm nginx -p 8080:80
```

Here, we’ve mapped container TCP port 80 to port 8080 on the Docker host for connections to host IP 192.168.1.100:

```
docker run -p 192.168.1.100:8085:80 nginx
```

You can verify this by running the following curl command:

```
$ curl 192.168.1.100:8085
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Let me briefly mention DNS configuration for containers. Docker provides your containers with the ability to make basic name resolutions:

```
$ docker exec busybox2 ping www.google.com
PING www.google.com (216.58.216.196): 56 data bytes
64 bytes from 216.58.216.196: seq=0 ttl=37 time=9.672 ms
64 bytes from 216.58.216.196: seq=1 ttl=37 time=6.110 ms
$ ping www.google.com
PING www.google.com (216.58.216.196): 56 data bytes
64 bytes from 216.58.216.196: icmp_seq=0 ttl=118 time=4.722 ms
```

Docker containers inherit DNS settings from the host when using a bridge network, so the container will resolve DNS names just like the host by default. To add custom host records to your container, you'll need to use the **relevant `--dns*` flags outlined here**.

Docker Compose Networking

Docker Compose is a tool for running multi-container applications on Docker, which are defined using the compose YAML file. You can start your applications with a single command: `docker-compose up`.

By default, Docker Compose creates a single network for each container defined in the compose file. All the containers defined in the compose file connect and communicate through the default network.

If you're not sure about the commands supported with Docker Compose, you can run the following:

```
$ docker compose help
Docker Compose

Usage:
  docker compose [command]

Available Commands:
  build       Build or rebuild services
  convert     Converts the compose file to a cloud format (default: clo
  down        Stop and remove containers, networks
  logs        View output from containers
  ls          List running compose projects
  ps          List containers
  pull        Pull service images
  push        Push service images
  run         Run a one-off command on a service.
  up          Create and start containers

Flags:
  -h, --help   help for compose

Global Flags:
      --config DIRECTORY   Location of the client config files DIRECTOR
  -c, --context string     context
  -D, --debug              Enable debug output in the logs
  -H, --host string        Daemon socket(s) to connect to

Use "docker compose [command] --help" for more information about a comm
```

Let's understand this with an example. In the following **`docker-compose.yaml`** file, we have a WordPress and a MySQL image.

When deploying this setup, **`docker-compose`** maps the WordPress container port 80 to port 80 of the host as specified in the compose file. We haven't defined any custom network, so it should create one for you. Run **`docker-compose up -d`** to bring up the services defined in the YAML file:

```
version: '3.7'
services:
  db:
    image: mysql:8.0.19
    command: '--default-authentication-plugin=mysql_native_password'
    restart: always
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    environment:
      - MYSQL_ROOT_PASSWORD=somewordpress
      - MYSQL_DATABASE=wordpress
      - MYSQL_USER=wordpress
      - MYSQL_PASSWORD=wordpress
  wordpress:
    image: wordpress:latest
    ports:
      - 80:80
    restart: always
    environment:
      - WORDPRESS_DB_HOST=db
      - WORDPRESS_DB_USER=wordpress
      - WORDPRESS_DB_PASSWORD=wordpress
      - WORDPRESS_DB_NAME=wordpress
volumes:
  db_data:
```

As you can see in the following output, a network named **`downloads_default`** is created for you:

```
$ docker-compose up -d
Creating network "downloads_default" with the default driver
Creating volume "downloads_db_data" with default driver
Pulling db (mysql:8.0.19)...
```

Verify that we have two containers up and running with the **`docker ps`** command:

```
$ docker ps
CONTAINER ID   IMAGE              COMMAND                  CREATED
f68265cd6219   wordpress:latest   "docker-entrypoint.s…"   6 minutes ag
2838f5586c73   mysql:8.0.19       "docker-entrypoint.s…"   6 minutes ag
```

Navigate to **`http://localhost:80`** in your web browser to access WordPress.

Now let’s inspect this network with the **`docker network inspect`** command. The following is the output:

```
$ docker network inspect downloads_default
[
    {
        "Name": "downloads_default",
        "Id": "717ea814aae357ceca3972342a64335a0c910455abf160ed820018b8
        "Created": "2021-03-05T03:43:42.541707419Z",
"Scope": "local",
"Driver": "bridge",
"EnableIPv6": false,
"IPAM": {
    "Driver": "default",
    "Options": null,
    "Config": [
        {
            "Subnet": "172.18.0.0/16",
            "Gateway": "172.18.0.1"
        }
    ]
},
"Internal": false,
"Attachable": true,
"Ingress": false,
"ConfigFrom": {
    "Network": ""
},
"ConfigOnly": false,
"Containers": {
    "2838f5586c735894051498c8ed0e5e103209cd22ee5718cbb29e8c6d16
        "Name": "downloads_db_1",
        "EndpointID": "10033e064387892253d69ac5813be6bc820a95df
        "MacAddress": "02:42:ac:12:00:02",
        "IPv4Address": "172.18.0.2/16",
        "IPv6Address": ""
    },
    "f68265cd6219fb60491c7ebbdae2d7f4c5ea4a74aec9987f5a5082c13b
        "Name": "downloads_wordpress_1",
        "EndpointID": "a4a673479ab3d812713955d461cc4d7032aba380
        "MacAddress": "02:42:ac:12:00:03",
        "IPv4Address": "172.18.0.3/16",
        "IPv6Address": ""
    }
},
"Options": {},
"Labels": {
    "com.docker.compose.network": "default",
    "com.docker.compose.project": "downloads",
    "com.docker.compose.version": "1.27.4"
}
```

```
    }
  ]
```

In the container sections, you can see that two containers (**downloads_db_1** and **downloads_wordpress_1**) are attached to the default **downloads_default** network driver, which is the bridge type. Run the following commands to clean up everything:

```
$ docker-compose down
Stopping downloads_wordpress_1 ... done
Stopping downloads_db_1        ... done
Removing downloads_wordpress_1 ... done
Removing downloads_db_1        ... done
Removing network downloads_default
```

You can observe that the network created by Compose is deleted, too:

```
$ docker-compose down -v
Removing network downloads_default
WARNING: Network downloads_default not found.
Removing volume downloads_db_data
```

The volume created earlier is deleted, and since the network is already deleted after running the previous command, it shows a warning that the default network is not found. That’s fine.

The example we’ve looked at so far covers the default network created by Compose, but what if we want to create our custom network and connect services to it? You will define the user-defined networks using the Compose file. The following is the **docker-compose** YAML file:

```
version: '3.7'
services:
  db:
    image: mysql:8.0.19
    command: '--default-authentication-plugin=mysql_native_password'
    restart: always
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    networks:
      - mynetwork
    environment:
      - MYSQL_ROOT_PASSWORD=somewordpress
      - MYSQL_DATABASE=wordpress
      - MYSQL_USER=wordpress
      - MYSQL_PASSWORD=wordpress
  wordpress:
    image: wordpress:latest
    ports:
      - 80:80
    networks:
      - mynetwork
    restart: always
    environment:
      - WORDPRESS_DB_HOST=db
      - WORDPRESS_DB_USER=wordpress
      - WORDPRESS_DB_PASSWORD=wordpress
      - WORDPRESS_DB_NAME=wordpress
volumes:
  db_data:
networks:
  mynetwork:
```

I've defined a user-defined network under the top-level networks section at the end of the file and called the network **`mynetwork`**. It's a bridge type, meaning it's a network on the host machine separated from the rest of the host network stack. Following each service, I added the network key to specify that these services should connect to **`mynetwork`**.

Let's bring up the services again after the changing the Docker Compose YAML file:

```
$ docker-compose up -d
Creating network "downloads_mynetwork" with the default driver
Creating volume "downloads_db_data" with default driver
Creating downloads_wordpress_1 ... done
Creating downloads_db_1        ... done
```

As you can see, Docker Compose has created the new custom **`mynetwork`**, started the containers, and connected them to the custom network. You can inspect it by using the Docker inspect command:

```
$ docker network inspect downloads_mynetwork
[
    {
        "Name": "downloads_mynetwork",
        "Id": "cb24ed3832dfdd34ca6fdcfcf0065c8e0df6b9b72f7b29a2aaada749
        "Created": "2021-03-05T04:23:14.354570267Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.19.0.0/16",
                    "Gateway": "172.19.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": true,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {
            "334bd5bf1689b067fa24705af0b6ab444976b288d25846349ce5b8f691
                "Name": "downloads_wordpress_1",
                "EndpointID": "10674d2ddd9feb67c98e3c08fcf451f32bda58e9
                "MacAddress": "02:42:ac:13:00:03",
                "IPv4Address": "172.19.0.3/16",
                "IPv6Address": ""
            },
            "9ffc94adab6896620648a1d08d215ba3d9423fee934b905b7bc2a44dd3
                "Name": "downloads_db_1",
                "EndpointID": "927943a0202bfb69692bc172a5c26e05676df696
                "MacAddress": "02:42:ac:13:00:02",
                "IPv4Address": "172.19.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {
            "com.docker.compose.network": "mynetwork",
            "com.docker.compose.project": "downloads",
            "com.docker.compose.version": "1.27.4"
        }
    }
]
```

Inspecting the network, you can see there are now two containers connected to the custom network.

Conclusion

In this article, we've covered the what and how of Docker networking in detail, starting with Docker's network drivers available out-of-the-box and then some advanced concepts such as overlay and macvlan. We ran through some examples of the most common Docker network commands, and then discussed some common use cases and general pitfalls of the available network drivers. We also covered port publishing, which allows the outside world to connect with containers, and how Docker resolves DNS names. Finally, we explored Docker Compose networking with some examples.

That should provide you with a decent overview of how Docker networking provides different modes of network drivers so that your containers can communicate on a single or multi-host setup. With this knowledge, you can pick and choose a network driver that fits your use case.

While you're here:

Ashish Choudhary

Ashish Choudhary is a software engineer with over 10 years of experience in tech, including design, development, and web apps. He's an active blogger and technical writer who loves Java, Spring Boot, DevOps, and the cloud, and is a strong advocate for open source.

Published: May 5, 2021

You may also enjoy

Using MongoDB with Docker

11 minute read

Docker is a powerful development platform that enables users to containerize software. These containers can be run on any machine, as well as in a public or...

Property-Based Testing In Go

3 minute read

Have you ever wanted your unit tests written for you? Property based testing is a powerful testing technique that, in a sense, is just that. You describe th...

Follow: Twitter GitHub Feed Write For Us