Explaining Docker Networking Concepts

Docker networking is basically used to establish communication between the docker containers and the outside world via host machine or you can say it is a communication passage through which all the isolated containers communicate with each other in various situations to perform the required actions. In this guide, we will explain basic Docker networking concepts with practical examples on Ubuntu.

If you haven't installed Docker yet, refer the following guide.

- **How to Install Docker in Ubuntu 18.04 LTS Server**

Basics of Docker usage:

- **Getting Started With Docker**

Explaining Docker Networking Concepts

All commands listed below are tested with **root** privileges on **Ubuntu**.

To manage network operations, like creating a new network, connecting a container to a network, disconnect a container from the network, listing available networks and removing networks etc., we use the following command:

```
docker network
```

```
root@CPDockerTEST:/home/ubuntu# docker network

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

Run 'docker network COMMAND --help' for more information on a command.
root@CPDockerTEST:/home/ubuntu#
```

Types of docker network drivers

To list all your networks, run:

```
docker network ls
```

```
root@CPDockerTEST:/home/ubuntu# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
2898a81657c5        bridge              bridge              local
94b2d88e4b68        host                host                local
e8467d9c80b1        none                null                local
root@CPDockerTEST:/home/ubuntu#
```

Let’s have some short introduction on all of them.

1. **Bridge network :** When you start Docker, a default bridge network is created automatically. A newly-started containers will connect automatically to it. You can also create user-defined custom bridge networks. User-defined bridge networks are superior to the default bridge network.
2. **Host network :** It remove network isolation between the container and the Docker host, and use the host’s networking directly. If you run a container which binds to port 80 and you use host networking, the container’s application is available on port 80 on the host’s IP address. Means you will not be able to run multiple web containers on the same host, on the same port as the port is now common to all containers in the host network.

3. **None network :** In this kind of network, containers are not attached to any network and do not have any access to the external network or other containers. So, this network is used when you want to completely disable the networking stack on a container.
4. **Overlay network :** Creates an internal private network that spans across all the nodes participating in the swarm cluster. So, Overlay networks facilitate communication between a docker swarm service and a standalone container, or between two standalone containers on different Docker Daemons.
5. **Macvlan network :** Some applications, especially legacy applications or applications which monitor network traffic, expect to be directly connected to the physical network. In this type of situation, you can use the Macvlan network driver to assign a MAC address to each container's virtual network interface, making it appear to be a physical network interface directly connected to the physical network.

Allow me to show you hands-on exercises to Bridge and Host networks.

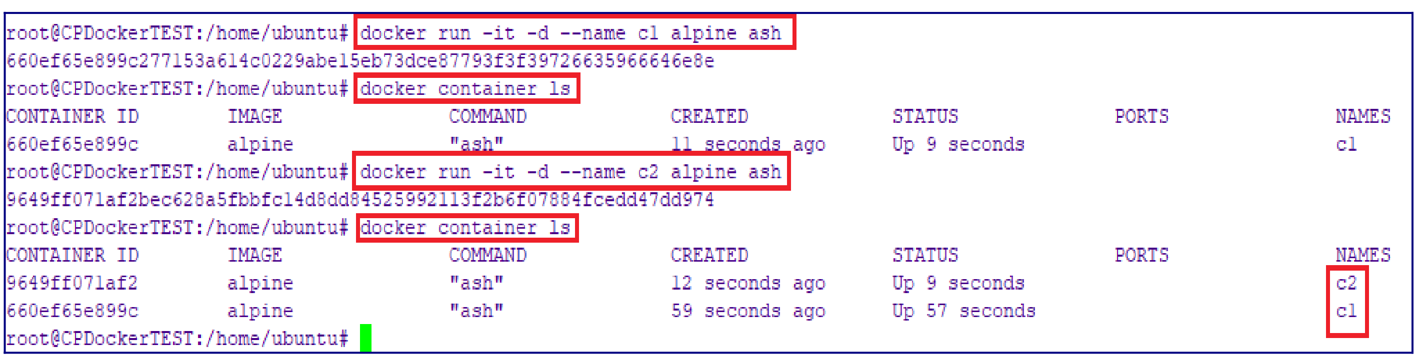
1. Bridge Network

I will be using two Alpine containers to explain this type of network.

Now, I am going to run two Alpine containers namely **C1** and **C2** using commands:

```
docker run -it -d --name c1 alpine ash
```

```
docker run -it -d --name c2 alpine ash
```



```
root@CPDockerTEST:/home/ubuntu# docker run -it -d --name c1 alpine ash
660ef65e899c277153a614c0229abe15eb73dce87793f3f39726635966646e8e
root@CPDockerTEST:/home/ubuntu# docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
660ef65e899c        alpine              "ash"               11 seconds ago      Up 9 seconds                            c1
root@CPDockerTEST:/home/ubuntu# docker run -it -d --name c2 alpine ash
9649ff071af2bec628a5fbbfc14d8dd84525992113f2b6f07884fcedd47dd974
root@CPDockerTEST:/home/ubuntu# docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
9649ff071af2        alpine              "ash"               12 seconds ago      Up 9 seconds                            c2
660ef65e899c        alpine              "ash"               59 seconds ago      Up 57 seconds                           c1
root@CPDockerTEST:/home/ubuntu#
```

Next, let us find out the IP address of those running containers. To do so, run:

```
docker exec -it c1 sh –c “ip a”

docker exec -it c2 sh –c “ip a”
```

```
root@CPDockerTEST:/home/ubuntu# docker exec -it c1 sh -c "ip a"
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    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
140: eth0@if141: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
root@CPDockerTEST:/home/ubuntu# docker exec -it c2 sh -c "ip a"
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    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
136: eth0@if137: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
root@CPDockerTEST:/home/ubuntu#
```

As you can see, the IP address of C1 container is **172.17.0.2** and IP address of C2 is **172.17.0.3**.

Now let us go ahead and try to ping each other to ensure if they can be able to communicate.

First, attach to the running C1 container and try to ping the C2 container:

```
docker attach c1

Ping –c 2 172.17.0.3
```

```
root@CPDockerTEST:/home/ubuntu# docker attach c1
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    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
140: eth0@if141: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
/ # ping -c 2 172.17.0.3
PING 172.17.0.3 (172.17.0.3): 56 data bytes
64 bytes from 172.17.0.3: seq=0 ttl=64 time=0.150 ms
64 bytes from 172.17.0.3: seq=1 ttl=64 time=0.109 ms

--- 172.17.0.3 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.109/0.129/0.150 ms
/ #
```

Similarly, attach to C2 container and try to ping C1 container.

```
docker attach c2

Ping –c 2 172.17.0.2
```

```
root@CPDockerTEST:/home/ubuntu# docker attach c2
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    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
144: eth0@if145: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
/ # ping -c 3 172.17.0.2
PING 172.17.0.2 (172.17.0.2): 56 data bytes
64 bytes from 172.17.0.2: seq=0 ttl=64 time=0.129 ms
64 bytes from 172.17.0.2: seq=1 ttl=64 time=0.110 ms
64 bytes from 172.17.0.2: seq=2 ttl=64 time=0.094 ms

--- 172.17.0.2 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.094/0.111/0.129 ms
/ #
```

As you see in the above screenshots, the communication is happening between the containers with in the same network.

We can also verify it by inspecting the bridge network using command:

`docker network inspect bridge`

The above command will display all information about the network, such as network type, subnet, gateway, containers name and iip addresses etc.

```
root@CPDockerTEST:/home/ubuntu# docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "5664b4902fca11e325599dd93568b2d005212e1cd084ccf1a4503ac5bd60fd79",
        "Created": "2019-10-13T05:46:26.876094Z",
        "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": {
            "660ef65e899c277153a614c0229abe15eb73dce87793f3f39726635966646e8e": {
                "Name": "c1",
                "EndpointID": "43ff4b8e1aff0d5fed150910af96b0053475a1789af365cae0ed8566d1dfec13",
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            },
            "9649ff071af2bec628a5fbbfc14d8dd84525992113f2b6f07884fcedd47dd974": {
                "Name": "c2",
                "EndpointID": "8807e30fdfcd9ed893e8e894a351842801fb5cc7e1eb32f2cdf6ba8ced6c8b98",
                "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",
```

1.1 Creating user-defined bridge network

Like I already said, when you start Docker, a **default bridge network is created automatically**. All newly-started containers will connect automatically to it. However, you can also create user-defined custom bridge networks.

To create new network driver, simply run:

```
docker network create my_net
```

Or,

```
docker network create --driver bridge dhruv_net
```

Both commands will do the same work. If you will not specify the driver name, it will create in the default network driver i.e. **bridge**.

```
root@CPDockerTEST:/home/ubuntu# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
5664b4902fca        bridge              bridge              local
94b2d88e4b68        host                host                local
e8467d9c80b1        none                null                local
root@CPDockerTEST:/home/ubuntu# docker network create my_net
261a6ece3cceefd8e75551e82951df8bd1113e1b5ce176ac4565090fb6d5eccf
root@CPDockerTEST:/home/ubuntu# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
5664b4902fca        bridge              bridge              local
94b2d88e4b68        host                host                local
261a6ece3cce        my_net              bridge              local
e8467d9c80b1        none                null                local
root@CPDockerTEST:/home/ubuntu# docker network create --driver bridge dhruv_net
e2e9b5886a9a894c4e526be8b4415ba042a88f168d776fae6bf4b53d97cf19a9
root@CPDockerTEST:/home/ubuntu# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
5664b4902fca        bridge              bridge              local
e2e9b5886a9a        dhruv_net           bridge              local
94b2d88e4b68        host                host                local
261a6ece3cce        my_net              bridge              local
e8467d9c80b1        none                null                local
root@CPDockerTEST:/home/ubuntu#
```

On user-defined networks like dhruv_net, containers can not only communicate by IP address, but can also resolve a container name to an IP address. This capability is called **automatic service discovery**.

To ensure if the containers can communicate with each other, let us run three alpine containers namely **A1**, **A2** and **A3** on **dhruv_net** network which we created earlier.

```
docker run -it -d --name A1 --network dhruv_net alpine ash

docker run -it -d --name A2 --network dhruv_net alpine ash

docker run -it -d --name A3 --network dhruv_net alpine ash
```

```
root@CPDockerTEST:/home/ubuntu# docker run -it -d --name A1 --network dhruv_net alpine ash
59aed907a148838e6a2c7d0e45ee870657cfa9c502fff2bb47c6a8ec3b6b4a1c
root@CPDockerTEST:/home/ubuntu# docker run -it -d --name A2 --network dhruv_net alpine ash
88652fcd409767a43ceac85658db2a746d2ad08dd2c3e40091243e032d5bd6ba
root@CPDockerTEST:/home/ubuntu# docker run -it -d --name A3 --network dhruv_net alpine ash
fb0a5b78133bab40bb6f99ddbb71f9035fc7be3213707295fe4804d469b66916
root@CPDockerTEST:/home/ubuntu# docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
fb0a5b78133b        alpine              "ash"               16 seconds ago      Up 13 seconds                           A3
88652fcd4097        alpine              "ash"               28 seconds ago      Up 25 seconds                           A2
59aed907a148        alpine              "ash"               37 seconds ago      Up 34 seconds                           A1
9649ff071af2        alpine              "ash"               9 hours ago         Up 4 hours                              c2
660ef65e899c        alpine              "ash"               9 hours ago         Up 4 hours                              c1
root@CPDockerTEST:/home/ubuntu#
```

Now try to attach to any one of the containers and ping the other two using container name.

```
root@CPDockerTEST:/home/ubuntu# docker container attach A1
/ # ping -c 2 A2
PING A2 (172.23.0.3): 56 data bytes
64 bytes from 172.23.0.3: seq=0 ttl=64 time=0.202 ms
64 bytes from 172.23.0.3: seq=1 ttl=64 time=0.146 ms

--- A2 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.146/0.174/0.202 ms
/ # ping -c 2 A3
PING A3 (172.23.0.4): 56 data bytes
64 bytes from 172.23.0.4: seq=0 ttl=64 time=0.150 ms
64 bytes from 172.23.0.4: seq=1 ttl=64 time=0.100 ms

--- A3 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.100/0.125/0.150 ms
/ #
```

```
root@CPDockerTEST:/home/ubuntu# docker container attach A2
/ # ping -c 2 A1
PING A1 (172.23.0.2): 56 data bytes
64 bytes from 172.23.0.2: seq=0 ttl=64 time=0.118 ms
64 bytes from 172.23.0.2: seq=1 ttl=64 time=0.129 ms

--- A1 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.118/0.123/0.129 ms
/ # ping -c 2 A3
PING A3 (172.23.0.4): 56 data bytes
64 bytes from 172.23.0.4: seq=0 ttl=64 time=0.139 ms
64 bytes from 172.23.0.4: seq=1 ttl=64 time=0.091 ms

--- A3 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.091/0.115/0.139 ms
/ #
```

From the above screenshots, it is proved that containers can be able to to communicate with each other.

2. Host Network

We are running a container which binds to port 80 using host networking, the container’s application is available on port 80 on the host’s IP address.

```
root@CPDockerTEST:/home/ubuntu# netstat -ntlp
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      12240/sshd
tcp        0      0 0.0.0.0:25324           0.0.0.0:*               LISTEN      14851/ruby
tcp6       0      0 :::22                   :::*                    LISTEN      12240/sshd
root@CPDockerTEST:/home/ubuntu# docker run -it -d --network host --name my_nginx nginx
2a59dd04feffe0bf9c6fa290edc3ba660e4014e48529d8f0d8c7fa807924cb45
root@CPDockerTEST:/home/ubuntu# netstat -ntlp
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN      4921/nginx: master
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      12240/sshd
tcp        0      0 0.0.0.0:25324           0.0.0.0:*               LISTEN      14851/ruby
tcp6       0      0 :::22                   :::*                    LISTEN      12240/sshd
root@CPDockerTEST:/home/ubuntu#
```

Host network is only needed when you are running programs with very specific network. The application running inside the Docker container look like they are running on the host itself, from the perspective of the network. It allows the container greater network access than it can normally get.

Here, we used **netstat -ntlp** command to display the listening port on the server. To find which service is listening on a particular port, **this guide**.

We've only covered the basics of Docker networking concepts. For more details, I suggest you to look into the Docker networking guide attached below.

- **Docker Container Networking**