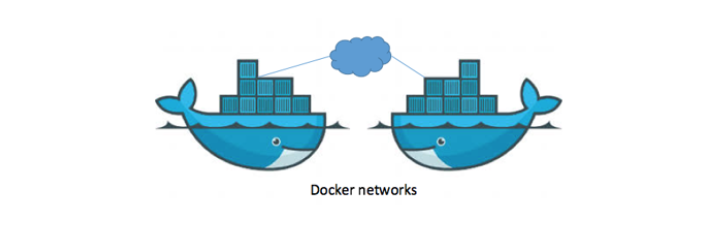
## **Docker Networking**

### **Networking overview**

One of the reasons [Docker containers](https://lms.clarusway.com/mod/lesson/view.php?id=2158) and services are so powerful is that you can connect them together, or connect them to non-Docker workloads. [Docker containers](https://lms.clarusway.com/mod/lesson/view.php?id=2158) and services do not even need to be aware that they are deployed on Docker, or whether their peers are also Docker workloads or not. Whether your Docker hosts run Linux, Windows, or a mix of the two, you can use Docker to manage them in a platform-agnostic way.



Network drivers

Docker’s networking subsystem is pluggable, using drivers. Several drivers exist by default, and provide core networking functionality:

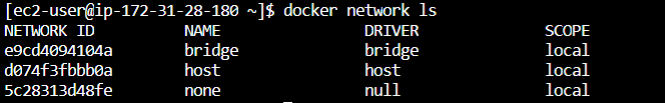
* **bridge:** The default network driver. If you don’t specify a driver, this is the type of network you are creating. Bridge networks are usually used when your applications run in standalone containers that need to communicate.
* **host:** For standalone containers, remove network isolation between the container and the Docker host, and use the host’s networking directly.
* **overlay:** Overlay networks connect multiple Docker daemons together and enable swarm services to communicate with each other. You can also use overlay networks to facilitate communication between a swarm service and a standalone container, or between two standalone containers on different Docker daemons. This strategy removes the need to do OS-level routing between these containers.
* **macvlan:** 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.
* **none:** For this container, disable all networking. Usually used in conjunction with a custom network driver.
* **Network plugins:** You can install and use third-party network plugins with Docker. These plugins are available from Docker Hub or from third-party vendors.

### **Use the default bridge network**

In this example, you start two different alpine containers on the same Docker host and do some tests to understand how they communicate with each other. You need to have Docker installed and running.

1. Open a terminal window. List current networks before you do anything else. Here’s what you should see if you’ve never added a network.

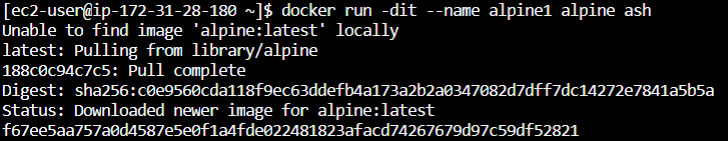
$ docker network ls



The default bridge network is listed, along with host and none. The latter two are not fully-fledged networks but are used to start a container connected directly to the Docker daemon host’s networking stack or to start a container with no network devices.

1. Start two alpine containers running ash, which is Alpine’s default shell rather than bash. The -dit flags mean to start the container detached (in the background), interactive (with the ability to type into it), and with a TTY (so you can see the input and output). Since you are starting it detached, you won’t be connected to the container right away. Instead, the container’s ID will be printed. Because you have not specified any --network flags, the containers connect to the default bridge network.

$ docker run -dit --name alpine1 alpine ash



$ docker run -dit --name alpine2 alpine ash



$ docker network inspect bridge



[

{

"Name": "bridge",

"Id": "57e7697201a40c4d6acb28f6ecaafac18d2ca282fd5eb3d8ced1e0ea5fc7b199",

"Created": "2020-07-13T10:10:39.886577521Z",

"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": {

"4bacf9c5b304ef197d8de32631e074805bfe154b8a5dc8458b71b30215ae25ac": {

"Name": "alpine2",

"EndpointID": "9bc6784ddf823bd7e7c6ac37ceddf87cfea63292f23ca3d5b178a367474a1299",

"MacAddress": "02:42:ac:11:00:03",

"IPv4Address": "172.17.0.3/16",

"IPv6Address": ""

},

"75846ffef56a6a0404bfebdd80ff0598f768a077b5c6d1e6b0c552990695ff40": {

"Name": "alpine1",

"EndpointID": "8426e9df26bf373f00dd185e23683e9c41ea45aee3aefed68ec1b28a397306fb",

"MacAddress": "02:42:ac:11:00:02",

"IPv4Address": "172.17.0.2/16",

"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": {}

}

]

Near the top, information about the bridge network is listed, including the IP address of the gateway between the Docker host and the bridge network (172.17.0.1). Under the Containers key, each connected container is listed, along with information about its IP address (172.17.0.2 for alpine1 and 172.17.0.3 for alpine2).

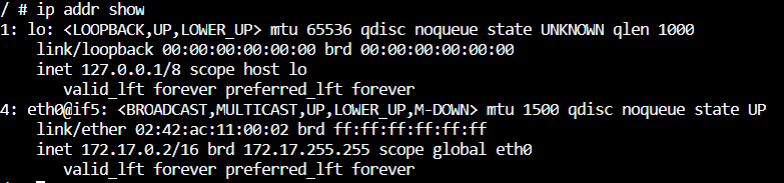
1. The containers are running in the background. Use the docker attach command to connect to alpine1.

$ docker attach alpine1



The prompt changes to # to indicate that you are the root user within the container. Use the ip addr show command to show the network interfaces for alpine1 as thet look from within the container:

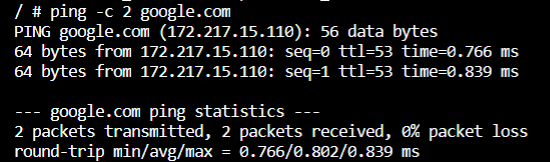
$ ip addr show



The first interface is the loopback device. Ignore it for now. Notice that the second interface has the IP address 172.17.0.2, which is the same address shown for alpine1 in the previous step.

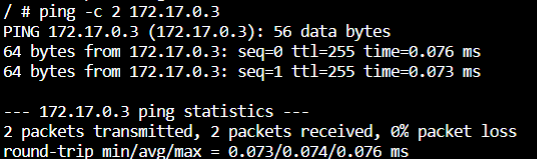
1. From within alpine1, make sure you can connect to the internet by pinging google.com. The -c 2 flag limits the command to two ping attempts.

$ ping -c 2 google.com



1. Now try to ping the second container. First, ping it by its IP address, 172.17.0.3:

$ ping -c 2 172.17.0.3



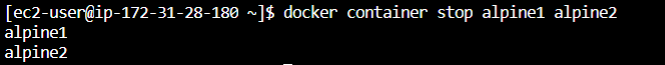
his succeeds. Next, try pinging the alpine2 container by container name. This will fail.

$ ping -c 2 alpine2

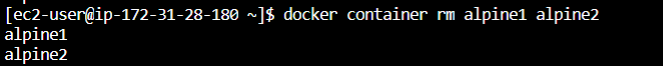


1. Detach from alpine1 without stopping it by using the detach sequence, CTRL + p + q (hold down CTRL and type p followed by q). If you wish, attach to alpine2 and repeat steps 4, 5, and 6 there, substituting alpine1 for alpine2.
2. Stop and remove both containers.

$ docker container stop alpine1 alpine2



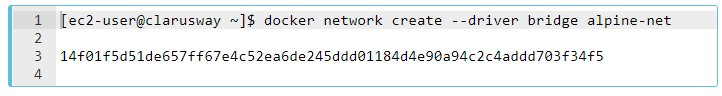
$ docker container rm alpine1 alpine2



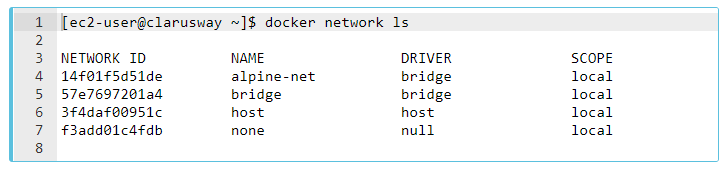
### **Use user-defined bridge networks**

In this example, we again start two alpine containers but attach them to a user-defined network called alpine-net which we have already created. These containers are not connected to the default bridge network at all. We then start a third alpine container which is connected to the bridge network but not connected to alpine-net and a fourth alpine container which is connected to both networks.

1. Create the alpine-net network. You do not need the --driver bridge flag since it’s the default, but this example shows how to specify it.



1. List Docker’s networks:

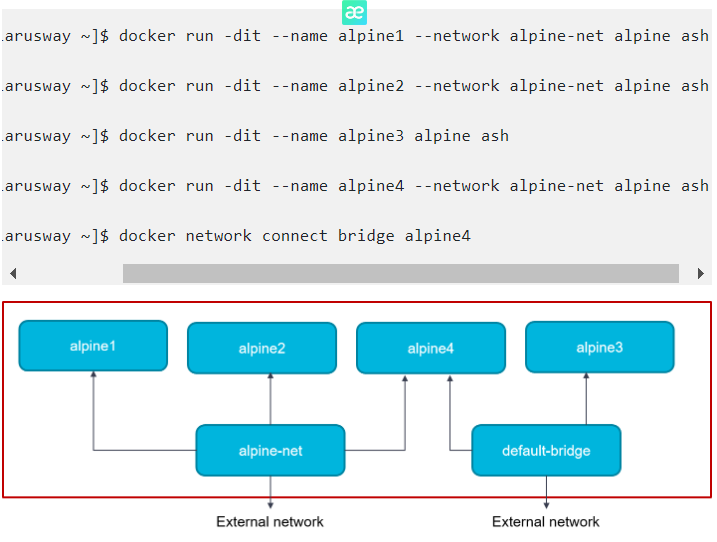


Inspect the alpine-net network. This shows you its IP address and the fact that no containers are connected to it:

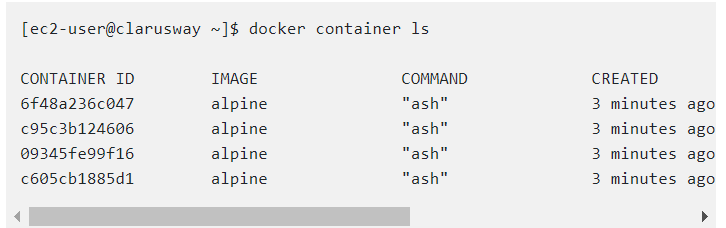


Notice that this network’s gateway is 172.18.0.1, as opposed to the default bridge network, whose gateway is 172.17.0.1. The exact IP address may be different on your system.

1. Create your four containers. Notice the --network flags. You can only connect to one network during the docker run command, so you need to use docker network connect afterward to connect alpine4 to the bridge network as well.



Verify that all containers are running:



1. Inspect the bridge network and the alpine-net network again:

[ec2-user@clarusway ~]$ docker network inspect alpine-net

[

{

"Name": "alpine-net",

"Id": "14f01f5d51de657ff67e4c52ea6de245ddd01184d4e90a94c2c4addd703f34f5",

"Created": "2020-07-13T10:54:23.575226558Z",

"Scope": "local",

"Driver": "bridge",

"EnableIPv6": false,

"IPAM": {

"Driver": "default",

"Options": {},

"Config": [

{

"Subnet": "172.18.0.0/16",

"Gateway": "172.18.0.1"

}

]

},

"Internal": false,

"Attachable": false,

"Ingress": false,

"ConfigFrom": {

"Network": ""

},

"ConfigOnly": false,

"Containers": {

"0891d7a4a8e3650ccb1ea169cd3e250226c734184f30757a60bd0513e2953924": {

"Name": "alpine4",

"EndpointID": "c691b71fb8ff6c0e1b6405feab28031acdaff5d764f0f0cd2901605e1c26a718",

"MacAddress": "02:42:ac:12:00:04",

"IPv4Address": "172.18.0.4/16",

"IPv6Address": ""

},

"ba2425af655ab6af6f74e90dcf51bc8639c663eb978b991d43940a45116ecffd": {

"Name": "alpine2",

"EndpointID": "6a6d0d3014a5e8192316bda4fb5bc19f63adb11f4de3acbc5fc30300efd88e54",

"MacAddress": "02:42:ac:12:00:03",

"IPv4Address": "172.18.0.3/16",

"IPv6Address": ""

},

"f1b109ada2ac11d005f9519846815d2ecb15b57b606d5e1aba418eb4b8f360d8": {

"Name": "alpine1",

"EndpointID": "7dab942e3fc98084552d9e80716eedd036808cdeda71c1068ec1a5b67c373e9d",

"MacAddress": "02:42:ac:12:00:02",

"IPv4Address": "172.18.0.2/16",

"IPv6Address": ""

}

},

"Options": {},

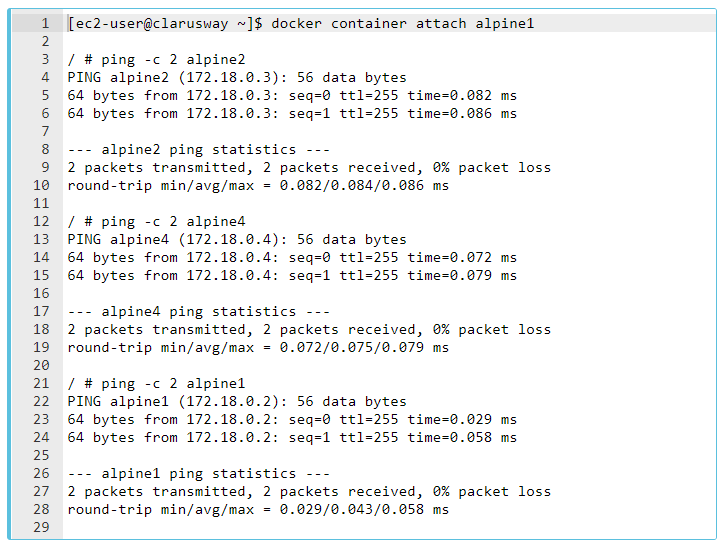
"Labels": {}

}

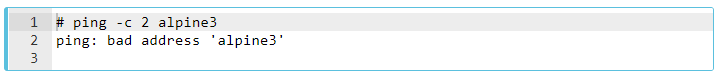
]

Containers alpine1, alpine2, and alpine4 are connected to the alpine-net network.

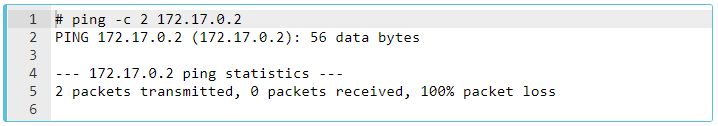
1. On user-defined networks like alpine-net, containers can not only communicate by IP address but can also resolve a container name to an IP address. This capability is called automatic service discovery. Let’s connect to alpine1 and test this out. alpine1 should be able to resolve alpine2 and alpine4 (and alpine1, itself) to IP addresses.



1. From alpine1, you should not be able to connect to alpine3 at all, since it is not on the alpine-net network.

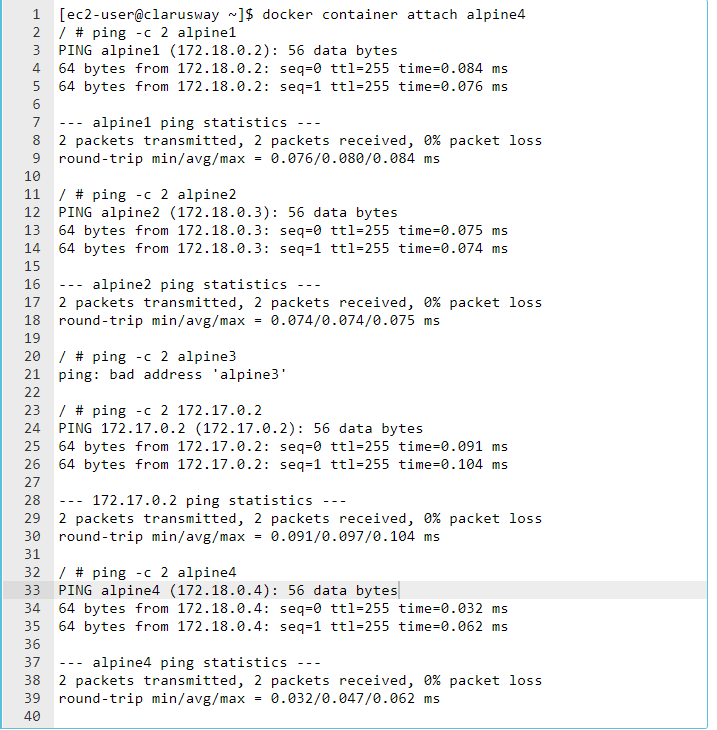


Not only that, but you can’t connect to alpine3 from alpine1 by its IP address either. Look back at the docker network inspect output for the bridge network and find alpine3’s IP address: 172.17.0.2 Try to ping it.

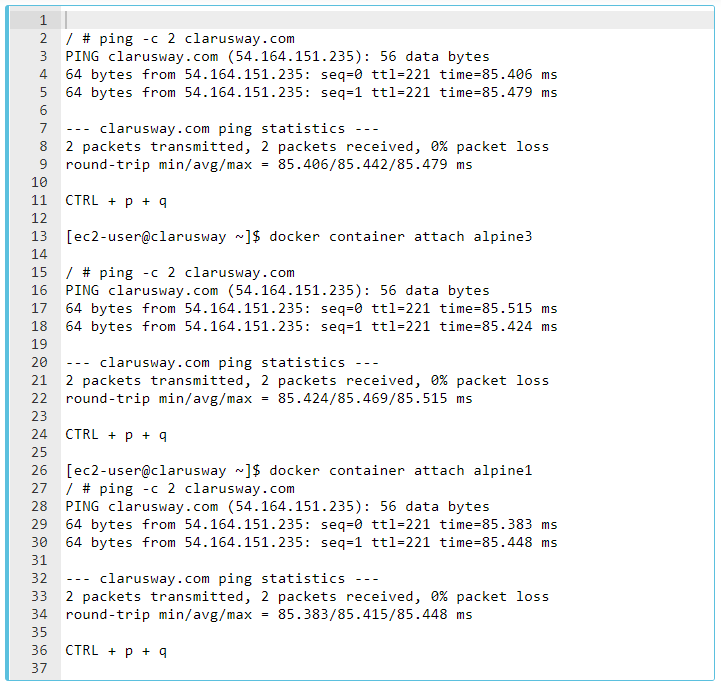


Detach from alpine1 using detach sequence, CTRL + p + q (hold down CTRL and type p followed by q).

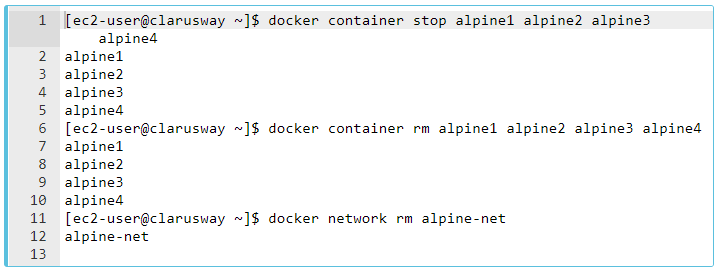
1. Remember that alpine4 is connected to both the default bridge network and alpine-net. It should be able to reach all of the other containers. However, you will need to address alpine3 by its IP address. Attach to it and run the tests.



8. As a final test, make sure your containers can all connect to the internet by pinging clarusway.com. You are already attached to alpine4 so start by trying from there. Next, detach from alpine4 and connect to alpine3 (which is only attached to the bridge network) and try again. Finally, connect to alpine1 (which is only connected to the alpine-net network) and try again.



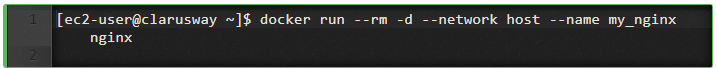
9. Stop and remove all containers and the alpine-net network.



### **Using the host network**

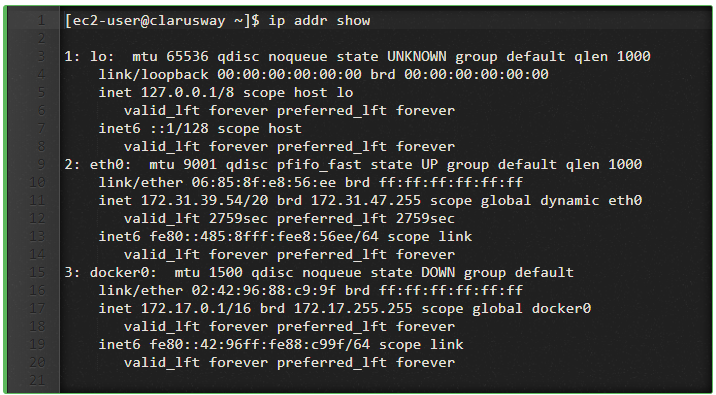
Let's start a nginx container which binds directly to port 80 on the Docker host. From a networking point of view, this is the same level of isolation as if the nginx process were running directly on the Docker host and not in a container. However, in all other ways, such as storage, process namespace, and user namespace, the nginx process is isolated from the host.

1. Create and start the container as a detached process. The --rm option means to remove the container once it exits/stops. The -d flag means to start the container detached (in the background)

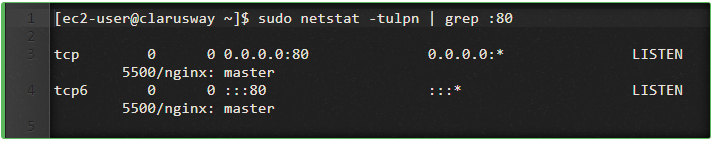


1. Access Nginx by browsing to http://localhost:80/. (< ip number of ec2 instance >:80/)
2. Examine your network stack using the following commands:

* Examine all network interfaces and verify that a new one was not created.



Verify which process is bound to port 80, using the netstat command. You need to use sudo because the process is owned by the Docker daemon user and you otherwise won’t be able to see its name or PID.



4. Stop the container. It will be removed automatically as it was started using the --rm option.

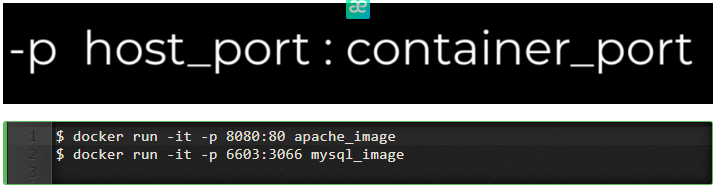


### **Container networking**

The type of network a container uses, whether it is a bridge, an overlay, a macvlan network, or a custom network plugin, is transparent from within the container. From the container’s point of view, it has a network interface with an IP address, a gateway, a routing table, DNS services, and other networking details (assuming the container is not using the none network driver). This lesson is about networking concerns from the point of view of the container.

### **Published ports**

By default, when you create a container, it does not publish any of its ports to the outside world. To make a port available to services outside of Docker, or to [Docker containers](https://lms.clarusway.com/mod/lesson/view.php?id=2158) which are not connected to the container’s network, use the --publish or -p flag. This creates a firewall rule which maps a container port to a port on the Docker host.



Here are some examples.

| **Flag value** | **Description** |
| --- | --- |
| -p 8080:80 | Map TCP port 80 in the container to port 8080 on the Docker host. |
| -p 192.168.1.100:8080:80 | Map TCP port 80 in the container to port 8080 on the Docker host for connections to host IP 192.168.1.100. |
| -p 8080:80/udp | Map UDP port 80 in the container to port 8080 on the Docker host. |
| -p 8080:80/tcp -p 8080:80/udp | Map TCP port 80 in the container to TCP port 8080 on the Docker host, and map UDP port 80 in the container to UDP port 8080 on the Docker host. |

### **IP address and hostname**

By default, the container is assigned an IP address for every Docker network it connects to. The IP address is assigned from the pool assigned to the network, so the Docker daemon effectively acts as a DHCP server for each container. Each network also has a default subnet mask and gateway.

When the container starts, it can only be connected to a single network, using --network. However, you can connect a running container to multiple networks using docker network connect. When you start a container using the --network flag, you can specify the IP address assigned to the container on that network using the --ip or --ip6 flags.

When you connect an existing container to a different network using docker network connect, you can use the --ip or --ip6 flags on that command to specify the container’s IP address on the additional network.

In the same way, a container’s hostname defaults to be the container’s ID in Docker. You can override the hostname using--hostname. When connecting to an existing network using docker network connect, you can use the --alias flag to specify an additional network alias for the container on that network.

### **DNS services**

By default, a container inherits the DNS settings of the host, as defined in the /etc/resolv.conf configuration file. Containers that use the default bridge network get a copy of this file, whereas containers that use a custom network use Docker’s embedded DNS server, which forwards external DNS lookups to the DNS servers configured on the host.

Custom hosts defined in /etc/hosts are not inherited. To pass additional hosts into your container, refer to add entries to container hosts file in the docker run reference documentation. You can override these settings on a per-container basis. settings on a per-container basis.

| **Flag value** | **Description** |
| --- | --- |
| --dns | The IP address of a DNS server. To specify multiple DNS servers, use multiple --dns flags. If the container cannot reach any of the IP addresses you specify, Google’s public DNS server 8.8.8.8 is added, so that your container can resolve internet domains. |
| --dns-search | A DNS search domain to search non-fully-qualified hostnames. To specify multiple DNS search prefixes, use multiple --dns-search flags. |
| --dns-opt | A key-value pair representing a DNS option and its value. See your operating system’s documentation for resolv.conf for valid options. |
| --hostname | The hostname a container uses for itself. Defaults to the container’s ID if not specified. |