CONTAINER NETWORKING

Overview

The goal of this lab is to provide you with a concise overview of how container networking works, how it differs from networking at the level of the Docker host, and how containers can leverage Docker networking to provide direct network connectivity to other containerized services. You will know how to deploy containers using networking configurations such as bridge, none, and host. You will learn the benefits of different networking drivers and under which circumstances you should choose certain network drivers.

Procedure

Networking basics

We will first examine the available networks in Docker environment.

- List the networks that are currently configured in your Docker environment using the docker network ls command:

  $ docker network ls

How many networks do you see in your environment? Briefly, explain their purpose.

When creating a container using Docker without specifying a network or networking driver, Docker will create the container using a bridge network. This network exists behind a bridge network interface configured in your host OS.

- Use ifconfig in a Linux or macOS Bash shell, or ipconfig in Windows PowerShell, Docker host, to see which interface the Docker bridge is configured as. It is generally called docker0:

  $ ifconfig

What is the Docker bridge interface called and state its IP address?

- Use the docker run command to create a simple NGINX web server container, using the latest image tag. Set the container to start in the background using the -d flag and give it a human-readable name of webserver1 using the --name flag:

  $ docker container run -d --name webserver1 -p 80:80 nginx:latest

- Execute the docker inspect command to check what networking configuration this container has by default:

  $ docker container inspect webserver1

Record the following network settings of your container.

  Gateway:

IP Address:

MAC Address:

Ports:

NetworkID:

What Docker network this container lives in? How were you able to identify?

- Create another container and expose a port to the outside world, use the docker run command followed by the -d flag to start the container in the background. The -p flag will enable you to specify a port on the host, separated by a colon and the port on the container that you wish to expose. Also, give this container a unique name, webserver2:

  $ docker container run -d -p 8080:80 --name webserver2 nginx:latest

- Inspect the configuration of the webserver2 instance using the docker inspect command followed by the container name or ID:

  $ docker container inspect webserver2

Record the following network settings of your container.

Gateway:

IP Address:

MAC Address:

Ports:

NetworkID:

Both the containers live on the same Docker network (bridge) and have the same default gateway, which is the docker0 bridge interface on the host machine.

- Since both containers live on the same subnet, you can test communication between the containers within the Docker bridge network. Run the docker exec command to gain access to a shell on the webserver1 container:

  docker container exec -it webserver1 /bin/bash

The prompt should noticeably change to a root prompt, indicating you are now in a Bash shell on the webserver1 container:

/#

- At the root shell prompt, use the apt package manager to install the ping utility in this container:

  /# apt-get update && apt-get install -y inetutils-ping

The apt package manager will then install the ping utility in the webserver1 container.

- Once the ping utility has successfully installed, use it to ping the IP address of the other container:

  /# ping 172.17.0.3

Were you able to ping the webserver2?

- You can also access the NGINX default web interface using the curl command. Install curl using the apt package manager:

  /# apt-get install -y curl

- After installing curl, use it to curl the IP address of webserver2:

  /# curl 172.17.0.3

You should see the Welcome to nginx! page displayed in HTML format, indicating that you were able to successfully contact the IP address of the webserver2 container through the Docker bridge network.

Working with Docker DNS Service

In this part of the lab, we will look at native Docker DNS and use human-readable DNS names to reliably send network traffic to other container instances.

- Create a Docker network called dnsnet and deploy two Alpine containers within that network. First, use the docker network create command to create a new Docker network using a 192.168.56.0/24 subnet and using the IP address 192.168.54.1 as the default gateway:

  $ docker network create dnsnet --subnet 192.168.54.0/24 --gateway 192.168.54.1

- Use the docker network ls command to list the Docker networks available in this environment:

  $ docker network ls

The list of Docker networks should be returned, including the dnsnet network you just created.

- Run the docker network inspect command to view the configuration for this network:

  $ docker network inspect dnsnet

The details of the dnsnet network should be displayed.

Are the Subnet and Gateway parameters being same that you had specified in earlier step?

What kind of network is this and specify its interface name? (hint: use Docker host for this)

- Now that a new Docker network has been created, use the docker run command to start a new container (alpinedns1) within this network. Use the docker run command with the --network flag to specify the dnsnet network that was just created, and the --network-alias flag to give your container a custom DNS name:

$ docker container run -itd --network dnsnet --network-alias alpinedns1 --name alpinedns1 alpine:latest

- Start a second container (alpinedns2) using the same --network and --network-alias settings:

  $ docker container run -itd --network dnsnet --network-alias alpinedns2 --name alpinedns2 alpine:latest

- Use the docker inspect command to verify that the IP addresses of the container instances are from within the subnet (192.168.54.0/24) that was specified:

  $ docker container inspect alpinedns1

Write down the IP address and Gateway address of alpinedns1.

IP Address: Gateway:

- Execute the docker network inspect command in a similar fashion for the alpinedns2 container:

  $ docker container inspect alpinedns2

Write down the IP address and Gateway address of alpinedns2.

IP Address: Gateway:

Are these both containers from the same subnetwork?

- Run the docker exec command to access a shell in the alpinedns1 container:

  $ docker container exec -it alpinedns1 /bin/sh

This should drop you into a root shell inside of the containers.

Once inside the alpinedns1 container, use the ping utility to ping the alpinedns2 container:

/ # ping alpinedns2

Are you able to ping using DNS name?

Similarly, run the docker exec command to gain access to shell in alpinedns2 container and verify the ping command.

How would you prove that containers re using true DNS when communicating with each other?

Clean up your environment by stopping all running containers using the docker stop command.

Use the docker system prune -fa command to clean the remaining stopped containers and networks:

$ docker system prune -fa

Exploring Docker Networks

In this part of the lab, we will look deeper into the various types of Docker network drivers that are supported by default.

- Use the docker network ls command to view the networks available in your Docker environment:

  $ docker network ls

- View the verbose details of these networks using the docker network inspect command, followed by the ID or the name of the network you want to inspect. In this step, you will view the verbose details of the bridge network:

  $ docker network inspect bridge

Some key parameters to note in this output are the Scope, Subnet, and Gateway keywords.

Can we share container network between Docker hosts?

Which interface is this network tied to on your Docker host?

- View the verbose details of the host network using the docker network inspect command:

  $ docker network inspect host

What can you conclude from the output of host network configuration?

Why is subnets, interfaces or other metadata not defined in this network?

- Investigate the none network next. Use the docker network inspect command to view the details of the none network:

  $ docker network inspect none

Why is none network mostly empty?

Based on your above observation, explain the differences between the none and host networks?

- Use the docker run command to start an Alpine Linux container in the host network. Name it hostnet1 to tell it apart from the other containers:

  $ docker container run -itd --network host --name hostnet1 alpine:latest

Docker will start this container in the background using the host network.

- Use the docker inspect command to look at the network configuration of the hostnet1 container you just created:

  $ docker container inspect hostnet1

Based on the output, why there is no IP address of gateway to the container?

Can this container communicate with other containers or the Internet?

- Use docker exec to access a sh shell inside this container, providing the name hostnet1:

  $ docker container exec -it hostnet1 /bin/sh

This should drop you into a root shell inside the hostnet1 container.

- Inside the hostnet1 container, execute the ifconfig command to list which network interfaces are available to it:

  / # ifconfig

What can you conclude about the output of the preceding command?

Why are we seeing the list of network interfaces available inside of the container when we have no network information from the docker inspect command?

- To understand more fully how the shared networking model works in Docker, start a NGINX container in host network mode. The NGINX container automatically exposes port 80, which we previously had to forward to a port on the host machine. Use the docker run command to start a NGINX container on the host machine:

  $ docker container run -itd --network host --name hostnet2 nginx:latest

This command will start a NGINX container in the host networking mode.

Navigate to http://Docker_host_IP using a web browser on the host machine.

You should be able to see the NGINX default web page displayed in your web browser. It should be noted that the docker run command did not explicitly forward or expose any ports to the host machine. Since the container is running in host networking mode, any ports that containers expose by default will be available directly on the host machine.

- Use the docker run command to create another NGINX instance in the host network mode. Call this container hostnet3 to differentiate it from the other two container instances:

  $ docker container run -itd --network host --name hostnet3 nginx:latest

- Now use the docker ls -a command to list all the containers, both in running and stopped status:

  $ docker container ls -a

Based on the preceding output, why is hostnet3 container exited and is currently in a stopped state? Justify your answer.

Clean up your Docker environment for future labs.