Lab Exercise: Networking Containers

Preamble

This lab exercise demonstrates how it is possible to control a container's networking capabilities, including the forwarding of a service port to the host, and how a container's service can be consumed from the Docker host, a sibling container, or from the container itself. This will be performed on a bridge network.

It also demonstrates how to establish a multi-host overlay network, and illustrates inter-host container service consumption. The lab exercise requires multiple Docker hosts, which can be achieved with Oracle Virtualbox VMs.

Bridge Network

Step 1 – create a new user-defined bridge network

List the currently available Docker networks registered with your local Docker daemon.

```
$ docker network ls
```

You should see output similar to the following:

```
NETWORK ID          NAME                DRIVER
a35b69b5ed3e        none                null
76a226c6cad7        host                host
913227fccf96        bridge              bridge
```

Using the docker network client command, create a new bridge network called 'my_bridge', with a subnet of `172.20.0.0/16` and a gateway of `172.20.0.254`.

```
$ docker network create --subnet 172.20.0.0/16 --gateway 172.20.0.254 \
> my_bridge
```

Use the `docker network ls` sub-command to confirm it has been created, and then use the `docker network inspect` sub-command to check it has been configured correctly.

Step 2 – forward a container's port

Start a new container based on the Ubuntu library image, connected to the 'my_bridge' network, give it the name 'netcat', and specify a port to forward to the host:

```
$ docker run -it --rm --name netcat--net my_bridge -p :80 ubuntu:latest \
> /bin/bash
```

Check which host port the container's port 80 has been forwarded to using the `docker inspect` command:

```
$ docker inspect -f \
> '{{(index (index .NetworkSettings.Ports "80/tcp") 0).HostPort}}' \
> netcat
32769
```

At the `bash` command shell inside the container, start the `netcat` utility listening on port 80:

```
$ nc -k -l 80
```

Start a new `bash` command shell on the host, and invoke the `netcat` utility, using the port number retrieved with the `docker inspect` command:

```
$ nc $(hostname) 32769
```

Type some text in the `bash` command shell on the host, and verify that it echoed within the container (remember to hit the `<ENTER>` key).

Terminate the `netcat` utility running on the host by typing `CTL-C`, but leave `netcat` running in the container.

Step 3 – connect from another container

Start a new container based on the Ubuntu library image, with a `bash` command shell as the container's process:

```
$ docker run -it --rm ubuntu:latest /bin/bash
```

From within the container, invoke the `netcat` utility, using the same port number you used in the previous step, using the hostname associated with your host:

```
$ nc <hostname> 32769
```

Type some text in the `bash` command shell, and verify that it is echoed within the container whose port has been forwarded to the host. Notice that the container was able to lookup the IP address of the host via the embedded DNS server provided by the Docker daemon.

Terminate the `netcat` utility running in the container by typing `CTL-C`, and exit the container (not the container with the port forwarded).

Step 4 – connect from the same container

Finally, we're going to use the `netcat` utility to make the container consume its own service.

From a `bash` command shell on the host, execute an interactive `bash` command shell in our container with the port forwarded, using the `docker exec` command:

```
$ docker exec -it netcat /bin/bash
```

In the new command shell, use the `netcat` utility to connect to the socket on the host (substituting `<hostname>` for the actual hostname, and `<host port no.>` for the actual port number):

```
$ nc <hostname> <host port no.>
```

Type a message and verify that it is echoed within the container that we have been monitoring in the previous steps.

Terminate the `netcat` utility running in both containers by typing `CTL-C`, and exit both containers. Use the `docker network rm` sub-command to remove the 'my_bridge' network from the list of networks available to connect to.

Overlay Network

To demonstrate the use of overlay networks in Docker, we will use a variation of Dj Walker-Morgan's goredchat application, a simple chat application that uses the Redis database engine to register chat users, and for routing chat messages. We'll create a Redis instance, and two client sessions using goredchat (all running in containers), but on different Docker hosts and connected via an overlay network.

Step 1 – establsih a key/value store

The first thing we need to do is establish the key/value store that overlay networking requires. We will use Hashicorp's Consul key/value store, and the easiest way to do this is to run it in a container on a dedicated VM. To simplify things, we'll use Docker Machine to create the VM. If `docker-machine` is not already installed locally, then follow these instructions.

Create the VM:

```
$ docker-machine create -d virtualbox kv-store
```

Point your Docker client at the Docker host running on the kv-store VM:

```
$ eval $(docker-machine env kv-store)
```

Check this has taken effect by issuing the following command, which should return the result shown:

```
$ docker info | grep Name
Name: kv-store
```

The next step is to start a container running Consul, and we'll use the `progrium/consul` image from the Docker Hub. The image needs some ports forwarded to its VM host, and can be started with the following command:

```
$ docker run -it -d --restart unless-stopped -p 8400:8400 -p 8500:8500 \
> -p 8600:53/udp -h consul progrium/consul -server -bootstrap -ui-dir /ui
```

Check that Consul is up and running by determining the IP address of the kv-store VM, and then entering the following URL into a web browser; `<ip>:8500/ui`:

```
$ docker-machine ip kv-store
192.168.99.100
```

If all is well, you will see the user-interface for the Consul application.

Step 2 – create three Docker hosts additional Docker hosts

Next, we'll create three different VMs, each running a Docker daemon, which we'll use to host containers that will be connected to the overlay network. Each of the Docker daemons running on these machine needs to be made aware of the KV store, and of each other.

```
$ for i in {1..3}; do \
> docker-machine create -d virtualbox --engine-opt \
> “cluster-store consul://$(docker-machine ip kv-store):8500” \
> “cluster-advertise eth1:2376” \
> host0${i}; done
```

Check that all the VMs are running as expected:

```
$ docker-machine ls -f \
> "table {{.Name}}  \t{{.State}}\t{{.URL}}\t{{.DockerVersion}}"
NAME         STATE     URL                         DOCKER
host01       Running   tcp://192.168.99.104:2376   v1.10.2
host02       Running   tcp://192.168.99.105:2376   v1.10.2
host03       Running   tcp://192.168.99.106:2376   v1.10.2
kv-store     Running   tcp://192.168.99.100:2376   v1.10.2
```

Step 2 – create an overlay network

We now need to run some Docker CLI commands on each of the Docker hosts we have

created. There numerous ways of doing this;

1. Establish an `ssh` session on the VM in question, using the `docker-machine ssh` command
2. Point our local Docker client at the Docker host in question, using the `docker-machine env` command
3. Run one-time commands against the particular Docker host using the `docker-machine config` command

The overlay network can be created using any of the Docker hosts:

```
$ docker $(docker-machine config host01) network create -d overlay \
> my_overlay
0223fc182bd363b95fce819bd33640c7afff8f5b306f0a93a7093b9a254cd9cc
```

Check that the overlay network 'my_overlay' can be seen from each of the Docker hosts:

```
$ docker $(docker-machine config host01) network ls -f name=my_overlay
NETWORK ID          NAME                DRIVER
0223fc182bd3        my_overlay          overlay
```

Step 3 – create a container running the Redis KV database engine

First, we need to start a Redis server running in a container on Docker host `host01`, using the library image found on the Docker Hub registry.

```
$ docker $(docker-machine config host01) run -d --restart unless-stopped \
> --net-alias redis_svr --net my_overlay redis:alpine redis-server \
> --appendonly yes
```

The library Redis image will be pulled from the Docker Hub registry, and the Docker CLI will return the container ID, e.g.

```
4d96bb2bd436bb6a19d5408de026904adc91892db40cfa25a8d71bae0e73bb29
```

The Redis library image exposes port `6379`, which will be accessible to containers running on the 'my_overlay' network. To test whether the `redis_svr` container is listening for connections, we can run the Redis client in an ephemeral container, also on `host01`:

```
$ docker $(docker-machine config host01) run --rm --net my_overlay \
> redis:alpine redis-cli -h redis_svr ping
PONG
```

Notice that Docker's embedded DNS server resolves the `redis_svr` name, which is an alias for the container, and the Redis server responds to the Redis client `ping`, with a `PONG`.

Step 4 – create a container running the goredchat application on host02

Now that we have established that a Redis server container is running on the 'my_overlay' network, we can attempt to consume its service from a container running on a different Docker host.

The goredchat image can be found on the Docker Hub registry, and can be run like a binary, with command line options. Run the following command to determine its usage:

```
$ docker $(docker-machine config host02) run --rm --net my_overlay \
> nbrown/goredchat --help
Usage: /goredchat [-r URL] username
  e.g. /goredchat -r redis://redis_svr:6379 antirez

  If -r URL is not used, the REDIS_URL env must be set instead
```

Now run a container using the `-r` configuration option to address the Redis server:

```
$ docker $(docker-machine config host02) run -it --rm --net my_overlay \
> nbrown/goredchat -r redis://redis_svr:6379 bill

Welcome to goredchat bill! Type /who to see who's online, /exit to exit.
```

Find out who's online by typing `/who`:

```
/who
bill
```

The goredchat client has created a TCP socket connection from the container on `host02` to the Redis server container on `host01` via the 'my_overlay' network.

Step 5 – create a container running the goredchat application on host03

In another bash command shell, we can start another instance of goredchat, this time on host03.

```
$ docker $(docker-machine config host03) run -it --rm --net my_overlay \
> nbrown/goredchat -r redis://redis_svr:6379 brian

Welcome to goredchat brian! Type /who to see who's online, /exit to exit.
```

Notice that in goredchat on `host02`, Bill gets a message that 'brian has joined'. Exchange some messages between Bill and Brian before exiting from the chat with `/exit`.

Stop and remove the Redis container, remove the 'my_overlay' network, stop and remove the Consul container, and then user `docker-machine rm` to remove the VMs.