Lab: Introduction to Docker - Part 2

More on containers

1. List all containers created (including those not currently running)
   `docker container ls -a`
2. List all container commands (this is very helpful)
   `docker container --help`
3. To remove containers, use either the ID (suppose there are three with IDs starting with '00', 'c6', '63')
   `docker container rm 00 c6 63`

   or use name
   `docker container rm mynginx`

   The same goes for stopping or starting containers.

Let's look more closely at what happens when you run the following command
`docker container run --publish 8080:80 --detach --name mywebhost nginx`

- Look for the image locally. If in local cache, use it.
- If the image is not already in the local cache, look in remote image repository (default is Docker Hub).
- Downloads latest version.
  (Next time a container is run from it, won't need to download, will use local one.)
- Creates new container based on that image.
- Assign an IP address inside a Docker virtual network.
- The option `--publish 8080:80` (notice the left port number is 8080) tells the container to open the port 8080 on the host machine (your machine) and all traffic to port 8080 will be forwarded to port 80 in the container.

  The format for publishing ports, after `--publish` (or `-p` for short), is `host-port:container-port`.

- The `--detach` (or `-d` for short) says run the container in detached mode in the background.

A few more things you can do:

- You can stop a container, then start it again
  `docker container stop mywebhost`
  `docker container start mywebhost`
- Show all the processes inside a container
  `docker container top mywebhost`
- List details of a container's configuration
  `docker container inspect mywebhost`
- Show logs of container
  `docker container logs mywebhost`
- Show all containers' CPU usage live stream
  `docker container stats`

Use a shell inside a container

You can start a new container interactively and get a shell inside that container. For example, run a container interactively from Ubuntu Linux image:

```
docker container -it --name myubuntu ubuntu
```

A few things to note here:

- The `-i` means *interactive*, it tells the container to keep the session open to receive terminal input.
- The `-t` means *tty*, simulates a real terminal, just as SSH does.
- Every image has a `CMD` in its configuration file called `Docker file`, this `CMD` is the default command that gets executed whenever a container is run from it. Because the ubuntu image's default `CMD` is bash (the Bash shell), so the above command

lands us right in Ubuntu bash right away. You can do anything here that you can do in a Ubuntu Linux bash shell.

It should be pointed out that the Ubuntu image is a very minimal distribution of Ubuntu (this is different from `.iso` for a VM). This is generally true for all the different Linux distribution images. But you can install any packages that you need.

So in Ubuntu bash, you can use `apt-get install` to install packages.

Let's install something, while you are still in bash:

```
apt-get update
apt-get install -y curl
```

Try out `curl` that you just installed:

```
curl https://uoit.ca
```

You can exit out of the bash shell by typing `exit`

When you exit out of bash, the container is stopped automatically.

You can start the container again in interactive mode:

```
docker start -ai myubuntu
```

The `i` is for interactive, the `a` is for attach.

Note: This container now has `curl` installed, can be used whenever it's started again.

Start a shell in a running container

If a container is already running, you can jump into it and start a shell inside by using the `docker container exec` command. This is useful for troubleshooting and changing configuration for a container that's already running.

For example, suppose you first run a ubuntu container (in detached interactive mode):

```
docker container run -dit --name newubuntu ubuntu
```

Now that `newubuntu` is running, you can start a shell inside with:

```
docker container exec -it newubuntu bash
```

Now you are in bash in the `newubuntu` container, try typing`ls` to see the directories. Type `exit` to exit.

Notice that this container is still running in the background. To stop it, type `docker`

`container stop newubuntu`.

More on `CMD`

Run `docker container run --help`, scroll up to the top, you'll see
`Usage: docker container run [OPTIONS] IMAGE [COMMAND] [ARG...]`
followed by all the options.

The `[COMMAND] [ARG...]` after the image is the optional **command and arguments** that tells the container what to run. The image has a default command `CMD` (we'll see later how to configure this). If you want to run something else other than the default, you can provide it here when you run the container.

Try the following:

1. Run two containers from nginx image. First one, name it `proxy1`, run in detached mode. Second one, name it `proxy2`, run in interactive mode and run bash instead of the default command. Do `ls`, then exit bash.

2. Show the list of running containers. Why is only `proxy1` running?
   Because you exited bash in `proxy2` and so that container stopped — a container only runs as long as the command it runs on start-up runs.

3. What is the default command for nginx image? (This can be seen in the listing of containers.)

Images and containers

An image is the bundle of executables, source code and libraries for an application. A container is just a process that results from executing that application.

You can list the images that you have locally stored, also shows the sizes of images:
`docker image ls`

To remove an image, say the nginx image:
`docker image rm nginx`

You can pull an image without creating and starting a container from it.
`docker pull alpine`

This will download the alpine Linux image which is a very tiny version of Linux, only about 4MB. The idea is that you can then install packages on a need-to-use basis.

`docker image ls`
will show you how small it is, compared to other images.

If you try to run bash in Alpine, it doesn't work because it doesn't have bash installed.
`docker container run -it alpine bash`

You can start just a shell in Alpine and use `apk` to install packages.
`docker container run -it alpine sh`

Inside Alpine shell, you can install bash:
`apk add bash`

Task

1. List all the images you have locally.
2. List all the containers (including those not running right now).
3. Remove all the containers. List all containers (this should show empty list).
4. Create and run a new container from the `nginx` image, call it `webhost`, in detached mode (running in the background), traffic to port 8008 on host machine is to be forwarded to port 80 in container. Verify this works, by typing `localhost:8008` in browser, should see the nginx page.
5. Stop `webhost` container. Then start it again.
6. Create and run a new container from the `ubuntu` image, in interactive mode and start a shell inside.

After you are done, make sure to stop all running containers.

Docker networks

When a container is started, it is connected to a private *virtual* Docker network. Each of these virtual networks routes through a NAT firewall on the host machine. This NAT is a Docker daemon that makes sure your containers can connect to the outside network or Internet through the network interface on the host.

Containers on the same virtual network can talk to each other easily. You can create new virtual networks.

There is much much more that can be customized and configured for containers and virtual networks. We will only use what we need as we go along.

Container's IP address

A container's IP address is *not* the IP address of the host machine.

In command shell:
`ipconfig`
will show the list of network connections. If you are connected to campus LAN, you will see something like: Ethernet adapter Ethernet 3, and it has its own IP address. If you are connected to wifi, you'll see an IP address for Wireless LAN. These are IP addresses for the host machine.

Now start a container:
`docker container run -d -p 8080:80 --name proxy nginx`

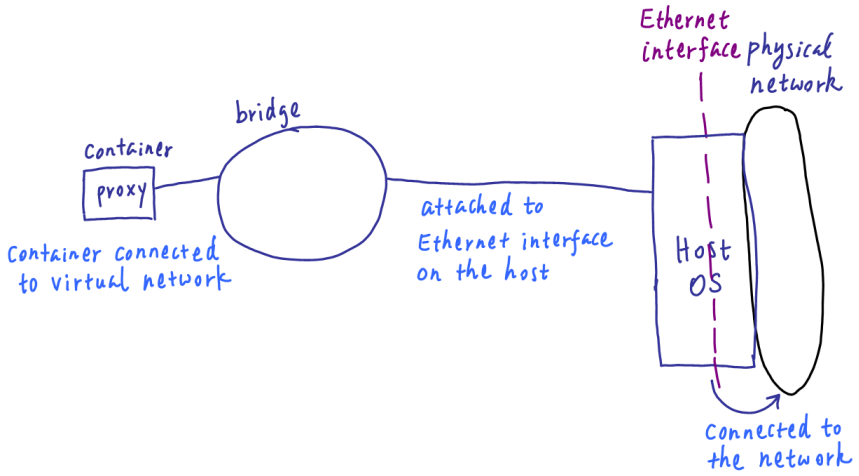
To find the IP address of the container:
`docker container inspect --format '{{.NetworkSettings.IPAddress}}' proxy`
will show the IP address of this container, it is *different* from the host IP address.

Verify the port mapping:
`docker container port proxy`

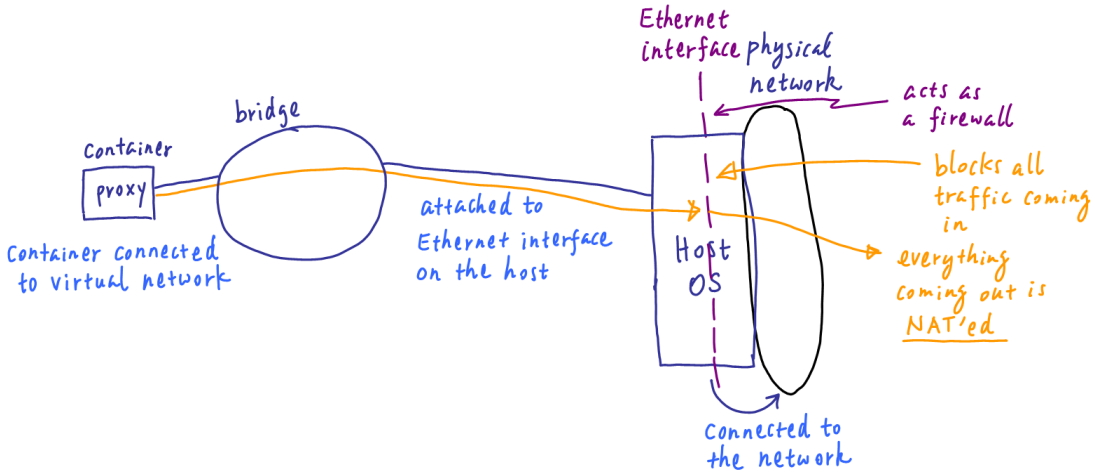
You can also do
`docker container inspect proxy`
to see all the information about this container, including the network it's connected to, its IP address, any port mappings.

How do Docker networks move packets in and out

Your host machine is connected to the physical network. See figure below. There is an Ethernet interface in the host for Docker. When a container is started, it connects to a virtual network called `bridge`. This virtual network is connected to the Ethernet interface.



This Ethernet interface acts like a NAT firewall. It blocks all traffic coming in from the external network. Everything that comes out of the containers and Docker virtual networks is NAT'ed.



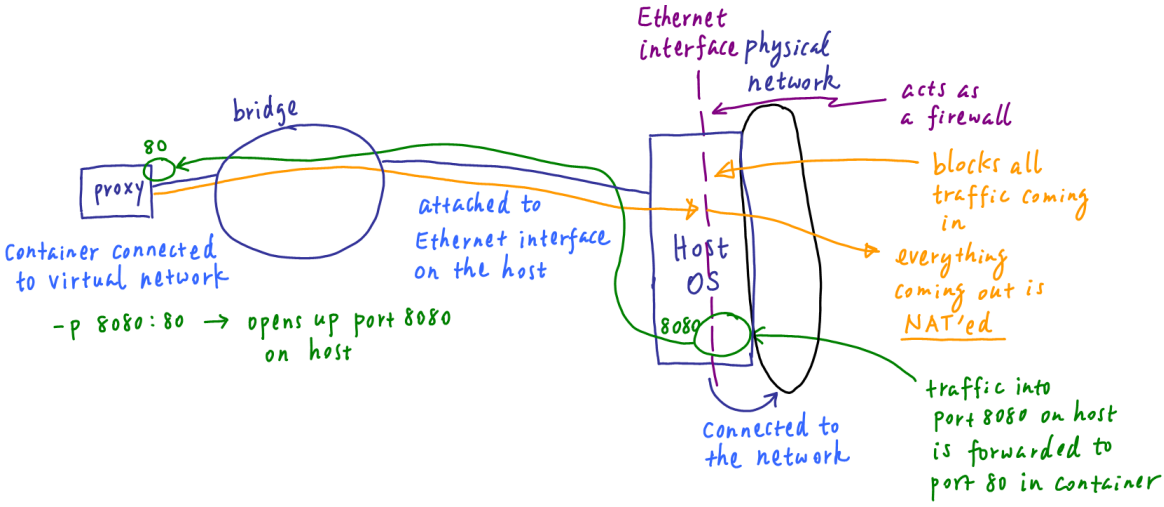
This is why we need to expose a port explicitly when we run a container, using the

`--publish` or `-p` option.

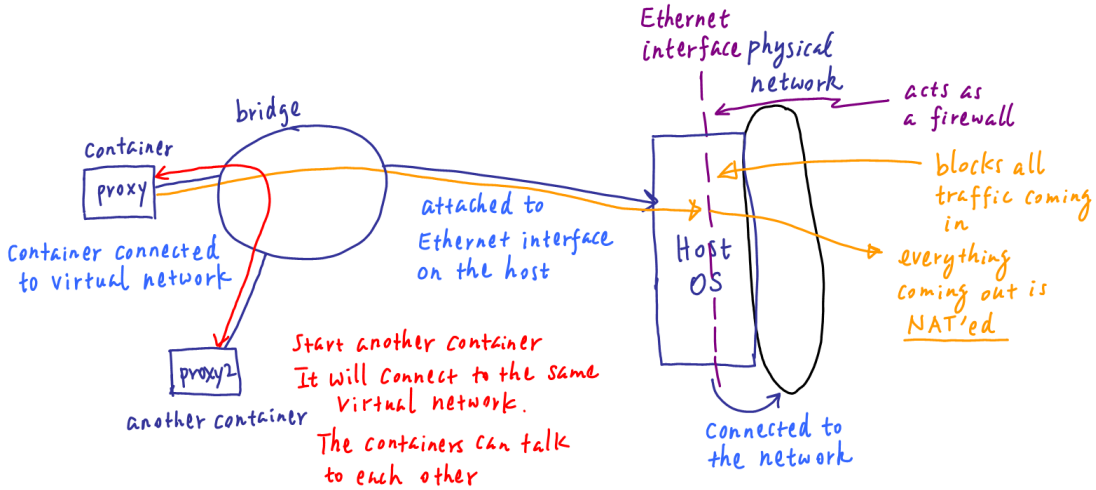
Suppose we run a container:

```
docker container -d -p 8080:80 --name proxy nginx
```

This opens up port 8080 on the host, and all traffic to 8080 on the host is directed to port 80 in the container.

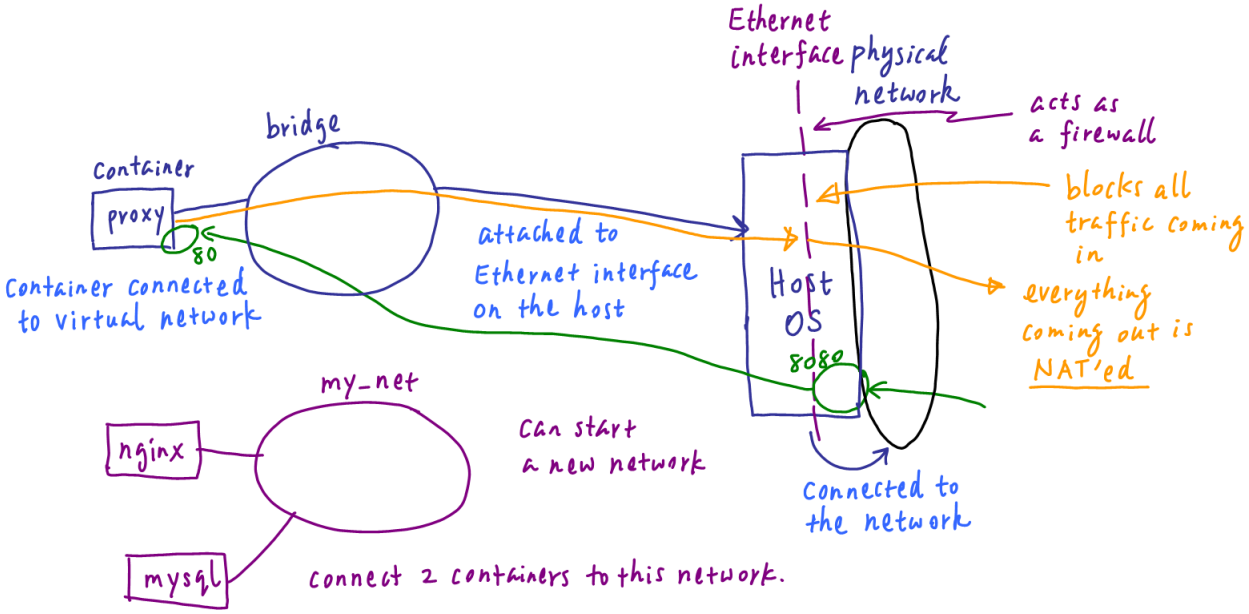


You can run another container and it will connect to the same virtual network. These two containers will each have their IP address on this network, and they will be able to talk to each other.



You can also create new networks and connect other containers to them. In the figure below, a new virtual network called `my_net` is created, and two containers are connected to it. These two containers will have an Ethernet interface that receives an IP address assigned by DHCP on this network.

Note: Since port 8080 is already used by a container (`proxy`), other containers will not be allowed to use this port on the host to receive packets from outside network.



Virtual networks

Let's go over the commands that deal with virtual networks.

To show the list of virtual networks that have been created:

```
docker network ls
```

You'll see the `bridge` network, it's the default Docker virtual network.

Information about a network can be found by:

```
docker network inspect bridge
```

It'll show a list of containers connected to this network, with their IP addresses. It also

shows the subnet used.

Create a new virtual network, call it `myNet`:
`docker network create myNet`

Now you can run a container and connect to this new network:
`docker container run -d --name newproxy --network myNet`

Connect a container created earlier called `proxy` to an existing network:
`docker network connect myNet proxy`

Docker security

The idea is that you should develop the front-end and back-end of your application as containers that connect to the same Docker network. This way, their communication stays on the virtual network which is completely hidden from the external network, protected by NAT. Their communication remains perfectly secure. All their externally exposed ports are closed by default. A port can only be opened by explicit instruction.

Task

1. For the `proxy` container that was run above, find the virtual network it's connected to.
2. Find its IP address.
3. Find its port mapping.
4. After creating the new virtual network above called `myNet`, connect a ubuntu container (name it, say, `ubuntu`) to it.
5. Show the container(s)that are connected to `myNet`.
6. For the `ubuntu` container, use `inspect` to look at its IP address on the network it's connected to.
7. Use `docker network --help` to find out how to disconnect a container from a virtual network. Disconnect the `ubuntu` container from `myNet`.

To Submit

Submit screen shot of the steps of the tasks. You should try to combine multiple steps into one screen shot, as much as possible.