Lab: Docker networking

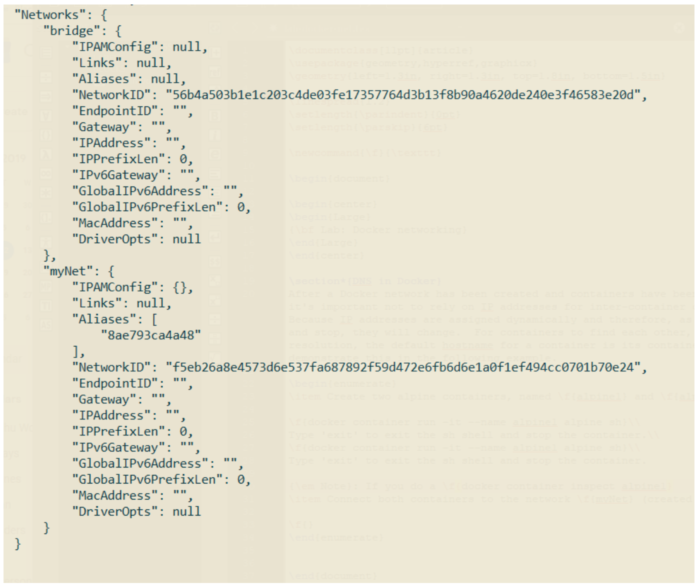
DNS in Docker

After a Docker network has been created and containers have been connected to it, it's important not to rely on IP addresses for inter-container communication. Because IP addresses are assigned dynamically and therefore, as the containers start and stop, they will change. For containers to find each other, Docker uses DNS name resolution, the default hostname for a container is its container name. We demonstrate this in the following example.

1. Create two alpine containers, named `alpine1` and `alpine2`.

   `docker container run -it --name alpine1 alpine sh`
   Type 'exit' to exit the sh shell and stop the container.
   `docker container run -it --name alpine2 alpine sh`
   Type 'exit' to exit the sh shell and stop the container.

   Note: If you do a `docker container inspect alpine1`, you'll see

```
"Networks": {
    "bridge": {
        "IPAMConfig": null,
        "Links": null,
        "Aliases": null,
        "NetworkID": "56b4a503b1e1c203c4de03fe17357764d3b13f8b90a4620de240e3f46583e20d",
        "EndpointID": "",
        "Gateway": "",
        "IPAddress": "",
        "IPPrefixLen": 0,
        "IPv6Gateway": "",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "MacAddress": "",
        "DriverOpts": null
    },
    "myNet": {
        "IPAMConfig": {},
        "Links": null,
        "Aliases": [
            "8ae793ca4a48"
        ],
        "NetworkID": "f5eb26a8e4573d6e537fa687892f59d472e6fb6d6e1a0f1ef494cc0701b70e24",
        "EndpointID": "",
        "Gateway": "",
        "IPAddress": "",
        "IPPrefixLen": 0,
        "IPv6Gateway": "",
        "GlobalIPv6Address": "",
        "GlobalIPv6PrefixLen": 0,
        "MacAddress": "",
        "DriverOpts": null
    }
}
```

   There is no IP address assigned to the container on either of the networks it's connected to. That's because it is not running. When it starts running, it will be

dynamically assigned an IP address, which may be different from the last time it was running.

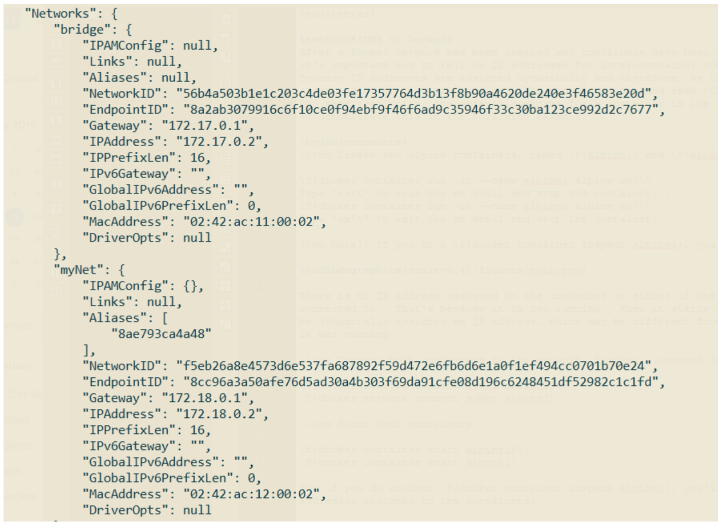
2. Connect both containers to the network `myNet` (created in the last lab).

   ```
   docker network connect myNet alpine1
   docker network connect myNet alpine2
   ```

3. Start both containers.

   ```
   docker container start alpine1
   docker container start alpine1
   ```

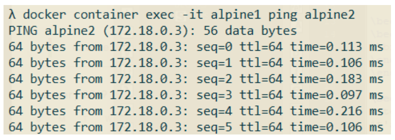
   Now if you do another `docker container inspect alpine1`, you'll see IP addresses assigned to the containers.

   ```
    "Networks": {
        "bridge": {
            "IPAMConfig": null,
            "Links": null,
            "Aliases": null,
            "NetworkID": "56b4a503b1e1c203c4de03fe17357764d3b13f8b90a4620de240e3f46583e20d",
            "EndpointID": "8a2ab3079916c6f10ce0f94ebf9f46f6ad9c35946f33c30ba123ce992d2c7677",
            "Gateway": "172.17.0.1",
            "IPAddress": "172.17.0.2",
            "IPPrefixLen": 16,
            "IPv6Gateway": "",
            "GlobalIPv6Address": "",
            "GlobalIPv6PrefixLen": 0,
            "MacAddress": "02:42:ac:11:00:02",
            "DriverOpts": null
        },
        "myNet": {
            "IPAMConfig": {},
            "Links": null,
            "Aliases": [
                "8ae793ca4a48"
            ],
            "NetworkID": "f5eb26a8e4573d6e537fa687892f59d472e6fb6d6e1a0f1ef494cc0701b70e24",
            "EndpointID": "8cc96a3a50afe76d5ad30a4b303f69da91cfe08d196c6248451df52982c1c1fd",
            "Gateway": "172.18.0.1",
            "IPAddress": "172.18.0.2",
            "IPPrefixLen": 16,
            "IPv6Gateway": "",
            "GlobalIPv6Address": "",
            "GlobalIPv6PrefixLen": 0,
            "MacAddress": "02:42:ac:12:00:02",
            "DriverOpts": null
   ```

4. Container `alpine1` can now `ping` container `alpine2` using its hostname (i.e., its container name) instead of IP address. This can always be done even if IP addresses change as containers stop and restart.

   ```
   docker container exec -it alpine1 ping alpine2
   ```

   ```
   λ docker container exec -it alpine1 ping alpine2
   PING alpine2 (172.18.0.3): 56 data bytes
   64 bytes from 172.18.0.3: seq=0 ttl=64 time=0.113 ms
   64 bytes from 172.18.0.3: seq=1 ttl=64 time=0.106 ms
   64 bytes from 172.18.0.3: seq=2 ttl=64 time=0.183 ms
   64 bytes from 172.18.0.3: seq=3 ttl=64 time=0.097 ms
   64 bytes from 172.18.0.3: seq=4 ttl=64 time=0.216 ms
   64 bytes from 172.18.0.3: seq=5 ttl=64 time=0.106 ms
   ```

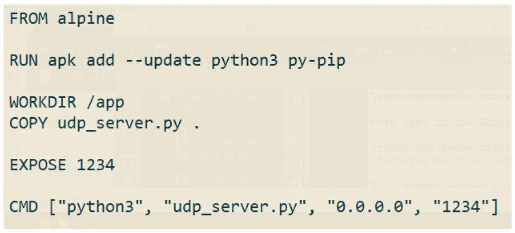
Note: When you type `docker network ls`, you'll see a list of the networks created. The first one is the default bridge network (called 'bridge'), and unlike all other networks you create, it does **not** have DNS built in. So it's better to create new networks for your applications to easily use the DNS feature.

Using Dockerfile to build images

In order to build your own images for your applications (or 'dockerize' your applications), you need to create a **Dockerfile**. An image is built off of the Dockerfile. Dockerfiles have their own syntax and required content.

We use the example of the `udp_client.py` and `udp_server.py` from Lab 3. In that application, we have a server program and a client program. We are now going to create an image for the server and an image for the client. This way, we can run different containers for the server and client (we can run multiple containers from the client image too) and they can communicate with each other with each having their own IP address in a Docker virtual network.

To build the server and client images, we need to write a Dockerfile for each image. Below is the Dockerfile for the server, we will explain every line in the following.



```
FROM alpine

RUN apk add --update python3 py-pip

WORKDIR /app
COPY udp_server.py .

EXPOSE 1234

CMD ["python3", "udp_server.py", "0.0.0.0", "1234"]
```

1. The first stanza (having only one line) is the FROM command which is required and must be in every Dockerfile. It's usually a minimal distribution (e.g., `debian` or `alpine`, these minimal distributions are much smaller than the traditional distributions). Here, we have `alpine`. So we are building the image from the base image of alpine, and adding to it.

   Each of these stanzas (blocks) is a layer in the docker image you are building, and upper layers are built on top of lower layers, so the ***order*** of them really matters.

2. The second stanza is a RUN command. It executes shell commands inside this container as it's building it.

   It's used to install software (not in the minimal distribution) with a package repository, or unzip or edit some files inside the container.

   It's also used to run any shell scripts that was copied into the container earlier in the Dockerfile, or any commands available at that point in the Dockerfile.

   Here, we have the alpine linux distribution, so we can use its `apk` command to install software that is not included in the minimal distribution. We install `python` here so we can run our server/client programs.

3. The third stanza sets the working directory in the container and copies the program file `udp_server.py` to this working directory inside the container. From this point on, you can run this program inside the container.

4. The fourth stanza is the EXPOSE command. By default, no TCP or UDP ports are open inside a container. To expose or open a port to the virtual network, it must be done explicitly by the EXPOSE command. Here, we expose port 1234. You still need to use -p to open and forward these ports on the host machine.

5. The last stanza is the CMD, it's required and it's the last command that is run whenever a new container is created off of this image and whenever a container is restarted after being stopped.

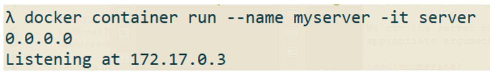
   Here the CMD runs the server program that we had copied into the container earlier, with the appropriate arguments.

Run server and client in separate Docker containers

First, we build the server image and create and start a server container. Note that we create a Dockerfile with the name `Dockerfile-server`. If you are not using the default Dockerfile name, you have to specify it using the `-f` option when you build the image. (Don't forget the '.' at the end of the docker build command, that tells Docker that the file(s) needed for building the image are in the current directory. Also, make sure all the files are in the same directory.)

```
docker build -t server -f Dockerfile-server .
docker container run --name myserver -it server
```

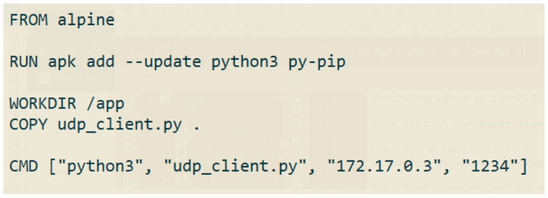
A container called myserver will run and print its IP address, suppose it's 172.17.0.2 (you may have a different one when you run this).



```
λ docker container run --name myserver -it server
0.0.0.0
Listening at 172.17.0.3
```

We also create a Dockerfile for the client image, called `Dockerfile-client`, and put 172.17.0.2 (or whatever IP address you get) in one of the CMD arguments.

The client Dockerfile, called `Dockerfile-client`, is shown below.



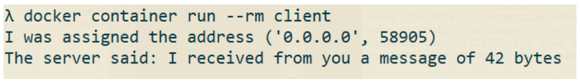
```
FROM alpine

RUN apk add --update python3 py-pip

WORKDIR /app
COPY udp_client.py .

CMD ["python3", "udp_client.py", "172.17.0.3", "1234"]
```

We build the client image and now can run multiple client containers.

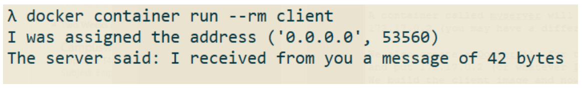
```
docker build -t client -f Dockerfile-client .
```

Below is an example of running two client containers:

```
docker container run --rm client
```

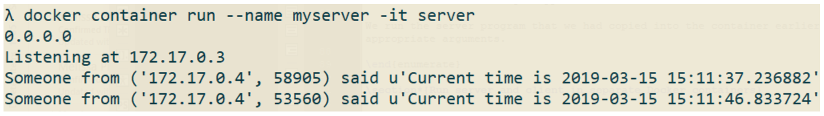


```
λ docker container run --rm client
I was assigned the address ('0.0.0.0', 58905)
The server said: I received from you a message of 42 bytes
```



```
λ docker container run --rm client
I was assigned the address ('0.0.0.0', 53560)
The server said: I received from you a message of 42 bytes
```

The server side after connections from the two clients:



```
λ docker container run --name myserver -it server
0.0.0.0
Listening at 172.17.0.3
Someone from ('172.17.0.4', 58905) said u'Current time is 2019-03-15 15:11:37.236882'
Someone from ('172.17.0.4', 53560) said u'Current time is 2019-03-15 15:11:46.833724'
```

Stop the server, say by `Ctrl-C`. Afterwards, the server can be restarted:

```
docker container start -ai myserver
```

Task

It's inconvenient and not very practical that the IP address of the server is hardcoded into the Dockerfile for buliding the client image — this means that in order to build the client image, the IP address of the server must known beforehand and more seriously, every time server IP address changes, the client image must be rebuilt.

Your task is to have the clients connect to the server using its hostname or container name (in the example run above, that was 'myserver').

Recall from the first section of this lab that when new containers are created, the default virtual network they are connected to is the default bridge network, and the default bridge network is the only one that does not have built-in DNS support. However, when you create a new virtual network and connect containers to it, the network provides DNS service and these containers can address each other by their container names.

Modify the client Dockerfile, and do whatever is necessary to run the server and have multiple clients connect to it by the container name of the server, not its IP address. This way, even if the IP address of server container changes in the future, the client image does not need to be built again.

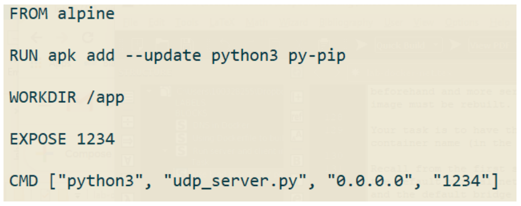
Use bind mounts for persistent data from containers

Once a container's lifetime is over, i.e., it's been removed, the data inside it is gone and cannot be recovered. To allow containers to leave persistent data after they are gone, we can use **bind mounts** to share directories on the host machine with containers.

A bind mount is a mapping from a directory on the host machine to a directory inside the container. Underneath, the two locations are pointing to the same directory and its files.

A container can read from and write to this directory, so bind mounts make it easier to put files into a container (you don't have to resort to COPY commands in the Dockerfile), and they make it possible for containers to put persistent data in that directory which will persist even after the containers are deleted.

The mapping of a bind mount is done at runtime when a container is created. The **syntax** is illustrated in the following example. For the `udp_server` image, instead of passing the program file `udp_server.py` to the container by using the `COPY` command in the Dockerfile, we create a bind mount at the time of creating a container so that the directory where `udp_server.py` resides is shared with the container (mapped from the local directory `/app` inside the container).



```
FROM alpine

RUN apk add --update python3 py-pip

WORKDIR /app

EXPOSE 1234

CMD ["python3", "udp_server.py", "0.0.0.0", "1234"]
```

Note that in the Dockerfile, there is no COPY of the program file.

Create the image first:
`docker build -t udpserver -f Dockerfile-server .`

Put `udp_server.py` in a local directory. Assume that the directory is `C:\code\\docker`, yours will be different.

Run a server container with a bind mount created for the directory `C:\code\\docker` (yours will be different, substitute in your directory here):
`docker container run --rm -it -v //c/code/docker:/app udpserver`

The `-v` option creates a bind mount. Here, the directory `C:\code\\docker` on the host machine is shared with the container and the directory `/app` inside the container is mapped to it.

Task

Develop a distributed logging application and run it as containers.

1. An asynchronous server takes connections from multiple clients.
2. Each client keeps getting lines from command-line input, and sends these messages to the server and the server stores these messages in a file on disk.
3. The first message sent by a client to the server includes the name of this client, e.g., 'John von Neumann'. Server will store every message m from this client as 'John von Neumann: m'
4. Modify the asynchronous server and client programs from Lab 6.
5. Create images for server and client, and run the server and multiple clients as separate containers that are connected to the same virtual network (it could be the one

you created in the previous task or a new one, as long as it's not the default bridge network).

The program files from Lab 6 are provided, `server_asyncio.py`, `client_questions.py`, `helpers.py`. The programs are slightly modified from Lab 6, to have the client ask user if they want to continue and take user input. If user types 'y' (for yes), then proceeds to send questions to server. If not 'y', then terminates.

In addition to modifying the code, there are a number of things you need to do to dockerize the application.

1. Create a Dockerfile for the server image and one for the client image.
2. In the Dockerfiles, do not use `COPY` to copy any files into the container.
3. For the `CMD` in the client Dockerfile, do not use an IP address (just as in the previous task).
4. Create a server image and a client image.
5. Suppose your program files are in the directory `C:\code\\docker` on your machine. Run a container for the server, and multiple client containers. When you call `docker run ...`, use the `-v` option to create a bind mount to share this directory `C:\code\\docker` with the containers. The server container will then store the messages in a file in this directory.

Because the log file with all the messages has been saved in the directory on the host machine mapped from inside the server container, this log file will still be there after all the containers have stopped and been deleted.

To submit

Submit all the files created and modified for the two tasks, including the program files and Dockerfiles. Submit a screen shot of the `docker run` commands that start the server container and multiple client containers for the second task.