Flavio Toffalini Follow

A Simple Network of Docker Containers

11 minute read

Published: July 08, 2019

The goal of this article is to illustrate a basic Docker configuration that simulates a network of virtual devices. For sake of simplicity, each Docker container runs the same application, however, it is possible to run containers with different applications. Having a large network helps studying the behavior of distributed algorithms without handling physical or virtual machines. In addition, I also discuss a simple way to logging in a scalable manner.

I faced this problem in a recent research project where I needed to simulate a large number of IoT devices. The application developed, basically, implements a Distributed Hash Table. This configuration enabled me to measure large scale properties, such as logarithmic research and network traffic. All of these without running out of grants :)

In this article, I will use a dummy network application. I however plan to release the full open source code, along with the research paper, after acceptance.

TL;TR;
To make the work easier for the laziest, I made a small repository with all the following examples at this link.

To sum up, what you will find in this article is:

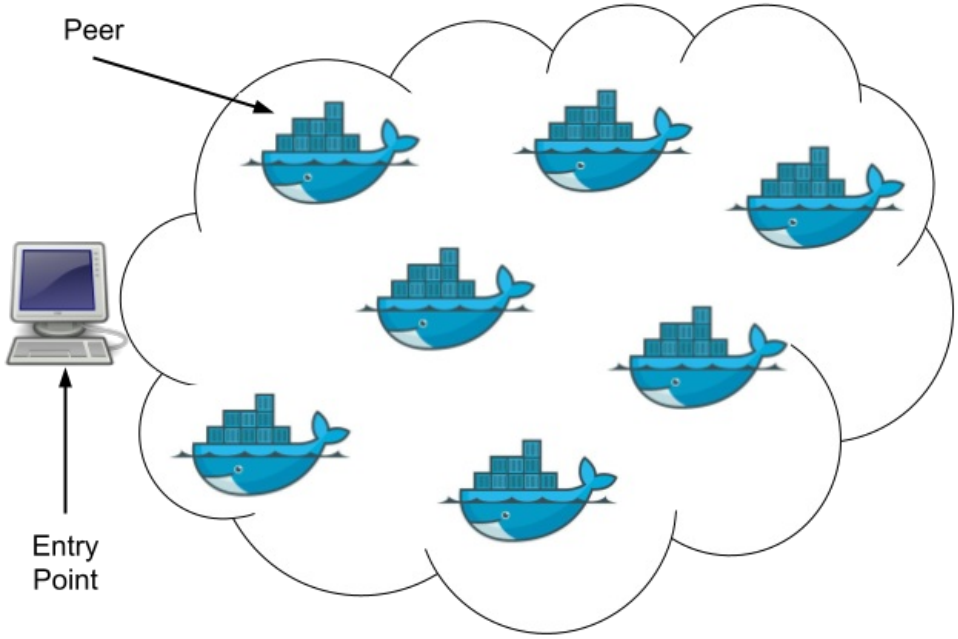
- Basic Docker container configuration.
- Practical tips for handling networks of Docker containers.
- Defining and managing a network of Docker containers, in particular:
  - How to start/stop the network in a controlled way.
  - Collect logs in a scalable fashion.

The rest of the post is organized as follows:

Overview

First thing first, let me describe what we are going to do.
Our network is composed by a number X of devices, called **peers**. The peers communicate with a central node, called **entry point**. Technically speaking, the entry point is implemented by our PC. The peers are, instead, implemented by docker

containers.
The network can be depicted as follows.



The purpose of this network is simple but still important. The entry point listens to communications from the peers. Meanwhile, the peers contact the entry point. During the execution, both peers and entry point write a local log file (i.e., in the respective container). Then, I will collect their log after an experiment session.

Preliminaries

Before going to the meat, we need some preliminary installation and configuration.

Configuration

All the following scripts/configuration files can stay in the same folder. Therefore, I suggest you create a workspace folder such as:

```
mk dockernetwork
cd dockernetwork
```

From this point ahead, I assume you run the commands in the folder `dockernetwork`.

Installation

If you have already installed these things, then, go ahead!
Docker is a quite flexible technology that allows installing so-called containers. In a nutshell, a container is an object that provides all the needs for the application, while it relies on the host OS kernel. In my experiments, I used an Ubuntu 18.04. The application is written in Python and requires Flask, while the network is handled by *Docker compose*.

To install the requirements, just open a terminal and let's start with Docker. I referred to this guide.

```
sudo apt-get install  curl apt-transport-https ca-certificates software-properties-common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
sudo apt update
sudo apt install docker-ce
sudo systemctl status docker
# to avoid further issues, we set ourselves as docker users
sudo groupadd docker # this may return an error
sudo usermod -aG docker $USER
```

For *Docker compose*, I followed this guide.

```
# install curl, one never knows
sudo apt update && sudo apt install curl
sudo curl -L "https://github.com/docker/compose/releases/download/1.24.0/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
```

```
sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose
# restart the shell
docker-compose --version
```

NOTE: afaik, new versions of Docker should allow handling groups of containers like Docker compose does. However, I found much easier this approach.

For Python and Flask, instead, the things are much simpler:

```
sudo apt update
sudo apt install python3 python3-pip
pip3 install flask
```

Entry Point

The entry point is a simple Web server written in Python3 and Flask. Let's have a look to this simple code.

```
#!/usr/bin/python3
import sys
import socket
from flask import Flask, request

app = Flask(__name__)

if len(sys.argv) != 2:
    print("usage: {0} <logfile>".format(sys.argv[0]))
    exit(0)

peerips = sys.argv[1]

@app.route('/add')
def add():
    global peerips
    ip = request.remote_addr
    if ip:
        with open(peerips, 'a') as l:
            l.write(ip + "\n")
        return "done"
    return "miss peer-ip"

app.run(debug=True, port=2222, host='0.0.0.0')
```

Just dump the former code in a file, e.g., `app_pc.py`. It will be useful later.

Also, don't forget.

```
chmod +x ./app_pc.py
```

The scope of the entry point is quite simple. It is a simple Web server that is listening requests at the url `http://XX.XX.XX.XX:2222/add`. At every `GET` request from a peer, the entry point extracts the IP address of the sender and saves it in a local log file.

Docker Container

As stated before, any peer is a Docker container.

To create a Docker container, we first define its image. Basically, create a new file, called `Dockerfile`, and dump inside it the following text.

```
FROM ubuntu:latest

RUN apt-get update -y
RUN apt-get upgrade -y
RUN apt-get install wget -y

CMD wget 172.20.0.2:2222/add; sleep 50; echo "My IP is `hostname -I | awk '{print $1}'`" > local.txt
```

The peer is quite simple. When it boots, it will contact the entry point by using a `wget`. This action sends the peer IP to the entry point, which is then logged.

From the `Dockerfile` previously described, we create a Docker image by throwing:

```
docker build -t "docker-network-peer" .
```

This command creates an image described by the previous `Dockerfile` and names it as `docker-network-peer`.

Explanation
Here, I briefly explain the `Dockerfile`. However, I suggest you read a plethora of better and deeper tutorials that discuss Docker.

The first line indicates the base OS on top of which we build our container. In our case, I start from an Ubuntu.

```
FROM ubuntu:latest
```

When you create a Docker container, the system downloads a minimal image, in our case based on Ubuntu. These OSes contain a minimal file system and utilities (e.g., apt-get). However, they are not usually up-to-date. Therefore, we run some commands to update the system:

```
RUN apt-get update -y
RUN apt-get upgrade -y
```

In the end, we install `wget`.

```
RUN apt-get install wget -y
```

All the previous commands starts with `FROM` and `RUN`. In a nutshell, these two keywords indicate commands that will be used only in the container creation phase. This means that they are executed only once. I still suggest you read a full guide.

The last line, instead, contains the current commands that describe the container behavior. Basically, it is the command that the container executes when it is booted. They are executed like bash commands.

```
CMD wget 172.20.0.2:2222/add; sleep 50; echo "My IP is `hostname -I | awk '{print $1}'`" > local.txt
```

Now, let's have a look at the previous line. I unroll them down for a better understanding.

```
wget 172.20.0.2:2222/add
sleep 50
echo "My IP is `hostname -I | awk '{print $1}'`" > local.txt
```

The first line is simply a `wget` toward the entry point `172.20.0.2:2222/add`. Then, the container sleeps for 50 seconds. Finally, the container finds its own IP and saves it in a local file within the container itself.

A container stops working once the command issued by `CMD` ends its execution. In my paper, I use a software that is based on an infinite loop to avoid a container to stop. If this is the case, we need to stop the container externally.

Network Configuration

Docker-compose can help handling interesting and complex sets of containers. In this setting, we use this tool to run and stop an arbitrary number of peers. One of the most useful thing of docker-compose is the ability to group a lot of boring setting in a simple configuration file called `docker-compose.yml`. In my case, I used docker-compose to set a virtual network called `docker-network-test` and to run/stop the peers.

Save a `docker-compose.yml` file with the following content:

```
version: '2.4'

services:
  peer:
    image: docker-network-peer:latest
    networks:
```

```
    - docker-network-test

networks:
  docker-network-test:
    driver: bridge
    ipam:
      driver: default
      config:
        - subnet: 172.20.0.0/16
          gateway: 172.20.0.2
```

The first part of the configuration file indicates the service itself. You can think of it as the context in which the container will be executed. In this configuration, I need two simple things:

- The container to execute: `docker-network-peer:latest`. The last version of our docker-network-peer that was previously explained.
- The network configuration for the peers: `docker-network-test`.

The network used in my example is described after the line `docker-network-test:`. Docker provides a number of different network configurations. The more suitable for my purpuse is the `bridge` (i.e., `driver: bridge`). This setting creates a `TUN` network interface in the host OS which allows intercommunication with the containers and the physical machine.

The network then requires further tuning:

```
- subnet: 172.20.0.0/16
  gateway: 172.20.0.2
```

These two commands define the space address of the virtual network. The containers will pick an IP from this space. The `gateway`, instead, have two functionalities: 1) it sets a `TUN` interface in the host with the gateway address, and 2) it sets a gateway address in the containers. In this way, the containers can reach the host through the gateway address.

In the next session, we see how to run the entire network.

Start-stop and network control

Open a shell, go to `dockernetwork`, and start the entry point:

```
cd dockernetwork
python3 app_pc.py entrypoint.txt
```

In another shell, go to `dockernetwork`, and launch the peers:

```
cd dockernetwork
docker-compose up --scale peer=10
```

The former command launches 10 peers. Then, wait until everything finishes. Finally, you shall close the entry point by typing `ctrl+c`.

In my example, each peer shuts down itself. If your peer application is meant to run for an undetermined time, you can use a command like that:

```
docker-compose up --scale peer=10; sleep 60; docker-compose stop
```

This command runs 10 peers, sleep for 60 seconds, and then stop itself.

After the execution, we have this situation:

1. each Docker container has a local log file
2. the entry point has its own log file, which is saved in the host.

TIP
When you stop the Docker containers externally, e.g., `docker-compose stop`, you are actually sending a signal to each single container. The applications within the container can be written to handle those signals and stop their execution properly.

Log Collection

To collect and inspect the logs, we can do:

Entry points

This is pretty simple, the entry point saves its log in the host machine. So, you can read it in this way:

```
cat entrypoint.txt
```

Peers

Each peer saves its log in the container, which is basically a file system not normally accessible from the host. For instance, to read the log from the container 1, you can use the following command.

```
docker cp dockernetwork_peer_1:/local.txt ./log1.txt
```

It is possible to automatize the log extraction by using the following script:

```
#!/bin/bash

NDEV=$1

echo "NDEV: $NDEV"

for i in `seq $NDEV`; do
  echo $i
  docker cp dockernetwork_peer_$i:/local.txt ./log\_$i.txt
done;
```

Just save this code in a script called `extractlog.sh`. Then, give some execution permission:

```
chmod +x extractlog.sh
```

This script just copies the local logs of each container to the host in the following way:

```
./extractlog.sh 10
```

Conclusion

This example shows the basic steps to build, start, and stop a network of Docker containers. For simplicity, I collected all the previous scripts in a simple repository.
Don’t hesitate to contact me for any reason!

Tags: docker virtual-network