Docker Kubernetes Lab

Release 0.1

Peng Xiao

Mar 12, 2021

Contents

This handbook contains some docker and kubernetes lab tutorials. It will be useful if you are learning docker or kubernetes now. The labs in this tutorial are all well documented, include the required environments, steps, detailed input and output.

Warning: This is just a lab guide, not a documentation for docker or kubernetes, please go to their online documentation sites for more details about what docker or kubernetes is and how does it work.

Table of Contents

1.1 Lab Environment Quick Setup

Please install vagrant before using vagrant files to quick start.

Download link: https://www.vagrantup.com/downloads.html

For what vagrant is and how to use it with virtualbox and vmware fusion, please reference https://www.vagrantup.com/docs/

And please install git if you don't have one on your machine(https://git-scm.com/)

1.1.1 Vagrant with one node docker engine

we will use vagrant to create one linux virtual machine and install docker automatically.

```
$ git clone https://github.com/xiaopeng163/docker-k8s-lab
$ cd docker-k8s-lab/lab/docker/single-node
```

There are two kinds of Linux, one is Ubuntu18.04, and one is CentOS7, please chose one, for example

```
$ git clone https://github.com/xiaopeng163/docker-k8s-lab
$ cd docker-k8s-lab/lab/docker/single-node
$ cd vagrant-centos7
$ vagrant up
```

`vagrant up` will take some time to create a virtual machine, after finished, you can use `vagrant ssh` ssh into this machine. like

```
$ vagrant status
Current machine states:

docker-host               running (virtualbox)

The VM is running. To stop this VM, you can run `vagrant halt` to
shut it down forcefully, or you can run `vagrant suspend` to simply
suspend the virtual machine. In either case, to restart it again,
simply run `vagrant up`.
$ vagrant ssh
Last login: Wed Jan 24 14:53:38 2018 from 10.0.2.2
[vagrant@docker-host ~]$ docker version
Client:
```

```
Version:      18.01.0-ce
API version:  1.35
Go version:   go1.9.2
Git commit:   03596f5
Built:        Wed Jan 10 20:07:19 2018
OS/Arch:      linux/amd64
Experimental: false
Orchestrator: swarm

Server:
Engine:
  Version:    18.01.0-ce
  API version:        1.35 (minimum version 1.12)
  Go version: go1.9.2
  Git commit: 03596f5
  Built:      Wed Jan 10 20:10:58 2018
  OS/Arch:    linux/amd64
  Experimental:       false
```

1.1.2 Vagrant with two node docker engine

```
$ git clone https://github.com/xiaopeng163/docker-k8s-lab
$ cd docker-k8s-lab/lab/docker/multi-node/vagrant
$ vagrant up
Bringing machine 'docker-node1' up with 'virtualbox' provider...
Bringing machine 'docker-node2' up with 'virtualbox' provider...
==> docker-node1: Importing base box 'ubuntu/bionic64'...
==> docker-node1: Matching MAC address for NAT networking...
==> docker-node1: Checking if box 'ubuntu/bionic64' is up to date...
......
```

The first time you run `vagrant up` will take some time to finished creating the virtual machine, and the time will depend on your network connection situation.

It will create two ubuntu 18.04 VMs based on the base box from the internet, and provision them.

We can use `vagrant ssh` to access each node:

```
$ vagrant status
Current machine states:

docker-node1              running (virtualbox)
docker-node2              running (virtualbox)

This environment represents multiple VMs. The VMs are all listed
above with their current state. For more information about a specific
VM, run `vagrant status NAME`.
$ vagrant ssh docker-node1
Welcome to Ubuntu 18.04 LTS (GNU/Linux 4.4.0-51-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud
```

```
0 packages can be updated.
0 updates are security updates.


Last login: Mon Dec  5 05:46:16 2016 from 10.0.2.2
ubuntu@docker-node1:~$ docker run -d --name test2 hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
c04b14da8d14: Pull complete
Digest: sha256:0256e8a36e2070f7bf2d0b0763dbabdd67798512411de4cdcf9431a1feb60fd9
Status: Downloaded newer image for hello-world:latest
52af64b1a65e3270cd525095974d70538fa9cf382a16123972312b72e858f57e
ubuntu@docker-node1:~$
```

You can play with docker now ~~

If you want to recovery your environment, just:

```
$ vagrant halt
$ vagrant destroy
$ vagrant up
```

1.2 Docker

1.2.1 Docker Engine Basic

When people say "Docker" they typically mean Docker Engine, the client-server application made up of the Docker daemon, a REST API that specifies interfaces for interacting with the daemon, and a command line interface (CLI) client that talks to the daemon (through the REST API wrapper). Docker Engine accepts docker commands from the CLI, such as docker run <image>, docker ps to list running containers, docker images to list images, and so on [1].

By default, the docker engine and command line interface will be installed together in the same host.

Note: Because docker's quick development, and docker's compatibility issue [4], we recommand you chose the verion > 1.10.0. And all the labs in this handbook, I use version 1.11.x and 1.12.x

---

Install Docker Engine on Linux

Host information:

```
$ cat /etc/redhat-release
CentOS Linux release 7.2.1511 (Core)
$ uname -a
Linux ip-172-31-43-155 3.10.0-327.28.2.el7.x86_64 #1 SMP Wed Aug 3 11:11:39 UTC 2016␣
↪x86_64 x86_64 x86_64 GNU/Linux
```

Install with scripts [2]:

1. Log into your machine as a user with sudo or root privileges. Make sure your existing packages are up-to-date.

[1] https://docs.docker.com/machine/overview/
[4] https://success.docker.com/Policies/Compatibility_Matrix
[2] https://docs.docker.com/engine/installation/linux/centos/

```
$ sudo yum update
```

2. Run the Docker installation script.

```
$ curl -fsSL https://get.docker.com/ | sh
```

This script adds the docker.repo repository and installs Docker.

3. Enable the service.

```
$ sudo systemctl enable docker.service
```

4. Start the Docker daemon.

```
$ sudo systemctl start docker
```

5. Verify docker is installed correctly by running a test image in a container.

```
$ sudo docker run --rm hello-world
```

Install Docker Engine on Mac

For the requirements and how to install `Docker Toolbox` on Mac, please go the reference link [5].

Install Docker Engine on Windows

For the requirements and how to install `Docker Toolbox` on Windows, please go to the reference link [6].

Docker Version

```
$ sudo docker version
Client:
 Version:      1.11.2
 API version:  1.23
 Go version:   go1.5.4
 Git commit:   b9f10c9
 Built:        Wed Jun  1 21:23:11 2016
 OS/Arch:      linux/amd64

Server:
 Version:      1.11.2
 API version:  1.23
 Go version:   go1.5.4
 Git commit:   b9f10c9
 Built:        Wed Jun  1 21:23:11 2016
 OS/Arch:      linux/amd64
```

Because there may have backwards incompatibilities if the versions of the client and server are different. We recommand that you should use the same version for client and server.

[5] https://docs.docker.com/engine/installation/mac/
[6] https://docs.docker.com/engine/installation/windows/

Docker without sudo

Because the docker daemon always runs as the root user, so it needs sudo or root to run some docker commands, like: docker command need sudo

```
$ docker images
Cannot connect to the Docker daemon. Is the docker daemon running on this host?
$ sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
hello-world         latest              c54a2cc56cbb        4 months ago        1.848␣
↪kB
```

But you can add your current user to docker group [3].

```
$ sudo groupadd docker
groupadd: group 'docker' already exists
$ sudo gpasswd -a ${USER} docker
Adding user centos to group docker
$ sudo service docker restart
Redirecting to /bin/systemctl restart  docker.service
```

Then logout current user, and login again. You can use docker command from your current user without sudo now.

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
hello-world         latest              c54a2cc56cbb        4 months ago        1.848␣
↪kB
```

Reference

1.2.2 Docker Machine on LocalHost

On macOS and Windows, docker machine is installed along with other Docker products when you install the Docker Toolbox. For example if you are using Mac:

```
$ docker-machine -v
docker-machine version 0.9.0, build 15fd4c7
```

If you are using other OS and want to install docker machine, please go to https://docs.docker.com/machine/install-machine/ for more details.

For what is docker machine and what docker machine can do, please go to https://docs.docker.com/machine/overview/

Create a machine

Docker Machine is a tool for provisioning and managing your Dockerized hosts (hosts with Docker Engine on them). Typically, you install Docker Machine on your local system. Docker Machine has its own command line client docker-machine and the Docker Engine client, docker. You can use Machine to install Docker Engine on one or more virtual systems. These virtual systems can be local (as when you use Machine to install and run Docker Engine in VirtualBox on Mac or Windows) or remote (as when you use Machine to provision Dockerized hosts on cloud providers). The Dockerized hosts themselves can be thought of, and are sometimes referred to as, managed "machines" [1].

For this lab, we will use docker machine on Mac system, and create a docker host with virtualbox driver.

[3] http://askubuntu.com/questions/477551/how-can-i-use-docker-without-sudo
[1] https://docs.docker.com/machine/overview/

Before we start, we can use `ls` command to check if there is any machine already in our host.

```
$ docker-machine ls
NAME    ACTIVE    DRIVER    STATE    URL    SWARM    DOCKER    ERRORS
```

Then create a machine called `default`.

```
$ docker-machine create -d virtualbox default
Running pre-create checks...
Creating machine...
(default) Copying /Users/penxiao/.docker/machine/cache/boot2docker.iso to /Users/
↪penxiao/.docker/machine/machines/default/boot2docker.iso...
(default) Creating VirtualBox VM...
(default) Creating SSH key...
(default) Starting the VM...
(default) Check network to re-create if needed...
(default) Waiting for an IP...
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with boot2docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on this virtual␣
↪machine, run: docker-machine env default
$ docker-machine ls
NAME      ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER ␣
↪   ERRORS
default   -        virtualbox   Running   tcp://192.168.99.100:2376           v1.12.3
```

How to use the docker host

There are two ways to access the docker host

- ssh into the docker host directly, then paly with docker inside
- use docker client on localhost (outside the docker host) to access the docker engine inside the docker host.

1. SSH into the docker host

```
$ docker-machine ssh default
                        ##         .
                  ## ## ##        ==
               ## ## ## ## ##    ===
           /"""""""""""""""""\___/ ===
      ~~~ {~~ ~~~~ ~~~ ~~~~ ~~~ ~ /  ===- ~~~
           \_______ o           __/
             \     \         __/
              \____\_______/
_                 _   ____     _            _
| |__   ___   ___ | |_|___ \ __| | ___   ___| | _____ _ __
| '_ \ / _ \ / _ \| __| __) / _` |/ _ \ / __| |/ / _ \ '__|
```

```
| |_) | (_) | (_) | |_ / __/ (_| | (_) | (__|   <  __/ |
|_.__/ \___/ \___/ \__|_____\__,_|\___/ \___|_|\_\___|_|
Boot2Docker version 1.12.3, build HEAD : 7fc7575 - Thu Oct 27 17:23:17 UTC 2016
Docker version 1.12.3, build 6b644ec
docker@default:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
 ↪STATUS              PORTS               NAMES
docker@default:~$
docker@default:~$ docker run --rm hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
c04b14da8d14: Pull complete
Digest: sha256:0256e8a36e2070f7bf2d0b0763dbabdd67798512411de4cdcf9431a1feb60fd9
Status: Downloaded newer image for hello-world:latest
```

2. docker client connect with remote docker engine

Get the environment commands for your new VM.

```
$ docker-machine env default
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
export DOCKER_CERT_PATH="/Users/penxiao/.docker/machine/machines/default"
export DOCKER_MACHINE_NAME="default"
# Run this command to configure your
```

Connect your docker client CLI to the new machine.

Before and after we run `eval "$(docker-machine env default)"` on localhost:

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu              14.04               aae2b63c4946        5 days ago          188 MB
mongo               2.6                 1999482cb0a5        6 weeks ago         391 MB
python              2.7                 6b494b5f019c        3 months ago        676.1
 ↪MB
tutum/nginx         latest              a2e9b71ed366        8 months ago        206.1
 ↪MB
$ eval "$(docker-machine env default)"
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
hello-world         latest              c54a2cc56cbb        5 months ago        1.848
 ↪kB
```

This sets environment variables for the current shell that the Docker client will read which specify the TLS settings. You need to do this each time you open a new shell or restart your machine. You can now run Docker commands on this host.

Reference

1.2.3 Docker Machine with Amazon AWS

Sign up for AWS and configure credentials [1]

Get AWS Access Key ID and Secret Access Key from `IAM`. Please reference AWS documentation. Then chose a Region and Available Zone, in this lab, we chose `region=us-west-1` which means North California, and Avaiable zone is `a`, please create a subnet in this zone [2].

Create a docker machine

```
  ~ docker-machine create --driver amazonec2 --amazonec2-region us-west-1 \
                          --amazonec2-zone a --amazonec2-vpc-id vpc-32c73756 \
                          --amazonec2-subnet-id subnet-16c84872 \
                          --amazonec2-ami ami-1b17257b \
                          --amazonec2-access-key $AWS_ACCESS_KEY_ID \
                          --amazonec2-secret-key $AWS_SECRET_ACCESS_KEY \
                          aws-swarm-manager
Running pre-create checks...
Creating machine...
(aws-swarm-manager) Launching instance...
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with ubuntu(upstart)...
Installing Docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on this virtual␣
↪machine, run: docker-machine env aws-swarm-manager
  ~ docker-machine ls
NAME                ACTIVE   DRIVER      STATE     URL                         SWARM␣
↪ DOCKER    ERRORS
aws-swarm-manager   -        amazonec2   Running   tcp://54.183.145.111:2376␣
↪ v17.10.0-ce
  ~
```

Please pay attention to `amazonec2-ami`, please chose a `Ubuntu 16:04`.

After created, We can use `docker-machine ssh` to access the host.

```
  ~ docker-machine ssh aws-swarm-manager

Welcome to Ubuntu 16.04.3 LTS (GNU/Linux 4.4.0-1038-aws x86_64)

* Documentation:  https://help.ubuntu.com
* Management:     https://landscape.canonical.com
* Support:        https://ubuntu.com/advantage

  Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud

4 packages can be updated.
1 update is a security update.
```

[1] https://docs.docker.com/machine/examples/aws/#/step-1-sign-up-for-aws-and-configure-credentials
[2] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/get-set-up-for-amazon-ec2.html

```
ubuntu@aws-swarm-manager:~$ sudo docker version
Client:
 Version:      17.10.0-ce
 API version:  1.33
 Go version:   go1.8.3
 Git commit:   f4ffd25
 Built:        Tue Oct 17 19:04:16 2017
 OS/Arch:      linux/amd64

Server:
 Version:      17.10.0-ce
 API version:  1.33 (minimum version 1.12)
 Go version:   go1.8.3
 Git commit:   f4ffd25
 Built:        Tue Oct 17 19:02:56 2017
 OS/Arch:      linux/amd64
 Experimental: false
ubuntu@aws-swarm-manager:~$
```

You can also use `docker-machine ip` to get the ip address of the docker host.

docker local client connect with remote aws docker host

Set the docker environment in local host.

```
  ~ docker-machine env aws-swarm-manager
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://xx.xx.xx.xx:2376"
export DOCKER_CERT_PATH="/Users/penxiao/.docker/machine/machines/aws-swarm-manager"
export DOCKER_MACHINE_NAME="aws-swarm-manager"
# Run this command to configure your shell:
# eval $(docker-machine env aws-swarm-manager)
  ~ eval $(docker-machine env aws-swarm-manager)
  ~ docker version
Client:
 Version:      1.12.3
 API version:  1.24
 Go version:   go1.6.3
 Git commit:   6b644ec
 Built:        Thu Oct 27 00:09:21 2016
 OS/Arch:      darwin/amd64
 Experimental: true

Server:
 Version:      17.10.0-ce
 API version:  1.33
 Go version:   go1.8.3
 Git commit:   f4ffd25
 Built:        Tue Oct 17 19:02:56 2017
 OS/Arch:      linux/amd64
  ~
```

Reference

1.2.4 Docker Command Line Step by Step

Docker Images

Docker images can be pulled from the docker hub, or build from `Dockerfile`.

docker pull

`docker pull` will pull a docker image from image registry, it's docker hub by default.

```
$ docker pull ubuntu:14.04
14.04: Pulling from library/ubuntu

04cf3f0e25b6: Pull complete
d5b45e963ba0: Pull complete
a5c78fda4e14: Pull complete
193d4969ca79: Pull complete
d709551f9630: Pull complete
Digest: sha256:edb984703bd3e8981ff541a5b9297ca1b81fde6e6e8094d86e390a38ebc30b4d
Status: Downloaded newer image for ubuntu:14.04
```

If the image has already on you host.

```
$ docker pull ubuntu:14.04
14.04: Pulling from library/ubuntu

Digest: sha256:edb984703bd3e8981ff541a5b9297ca1b81fde6e6e8094d86e390a38ebc30b4d
Status: Image is up to date for ubuntu:14.04
```

docker build

Create a `Dockerfile` in current folder.

```
$ more Dockerfile
FROM        ubuntu:14.04
MAINTAINER  xiaoquwl@gmail.com
RUN         apt-get update && apt-get install -y redis-server
EXPOSE      6379
ENTRYPOINT  ["/usr/bin/redis-server"]
```

Use `docker build` to create a image.

```
$ docker build -t xiaopeng163/redis:0.1 .
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
xiaopeng163/redis   0.1                 ccbca61a8ed4        7 seconds ago       212.4␣
↪MB
ubuntu              14.04               3f755ca42730        2 days ago          187.9␣
↪MB
```

docker history

```
$ docker history xiaopeng163/redis:0.1
IMAGE               CREATED             CREATED BY                                      ␣
↪  SIZE                COMMENT
ccbca61a8ed4        2 minutes ago       /bin/sh -c #(nop) ENTRYPOINT ["/usr/bin/redis␣
↪  0 B
13d13c016420        2 minutes ago       /bin/sh -c #(nop) EXPOSE 6379/tcp               ␣
↪  0 B
c2675d891098        2 minutes ago       /bin/sh -c apt-get update && apt-get install ␣
↪  24.42 MB
c3035660ff0c        2 minutes ago       /bin/sh -c #(nop) MAINTAINER xiaoquwl@gmail.c␣
↪  0 B
3f755ca42730        2 days ago          /bin/sh -c #(nop)  CMD ["/bin/bash"]            ␣
↪  0 B
<missing>           2 days ago          /bin/sh -c mkdir -p /run/systemd && echo 'doc␣
↪  7 B
<missing>           2 days ago          /bin/sh -c sed -i 's/^#\s*\(deb.*universe\)$/␣
↪  1.895 kB
<missing>           2 days ago          /bin/sh -c rm -rf /var/lib/apt/lists/*          ␣
↪  0 B
<missing>           2 days ago          /bin/sh -c set -xe   && echo '#!/bin/sh' > /u␣
↪  194.6 kB
<missing>           2 days ago          /bin/sh -c #(nop) ADD file:b2236d49147fe14d8d␣
↪  187.7 MB
```

docker images

`docker images` will list all avaiable images on your local host.

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu              14.04               aae2b63c4946        12 hours ago        187.9␣
↪MB
```

docker rmi

Remove docker images.

```
$ docker rmi aae2b63c4946
Untagged: ubuntu:14.04
Deleted: sha256:aae2b63c49461fcae4962e4a8043f66acf8e3af7e62f5ebceb70b181d8ca01e0
Deleted: sha256:50a2a0443efd0936b13eebb86f52b85551ad7883e093ba0b5bad14fec6ccf2ee
Deleted: sha256:9f0ca687b5937f9ac2c9675065b2daf1a6592e8a1e96bce9de46e94f70fbf418
Deleted: sha256:6e85e9fb34e94d299bb156252c89dfb4dcec65deca5e2471f7e8ba206eba8f8d
Deleted: sha256:cc4264e967e293d5cc16e5def86a0b3160b7a3d09e7a458f781326cd2cecedb1
Deleted: sha256:3181634137c4df95685d73bfbc029c47f6b37eb8a80e74f82e01cd746d0b4b66
```

Docker Containers

Start a container in interactive mode

```
$ docker run -i --name test3  ubuntu:14.04
pwd
/
ls -l
total 20
drwxr-xr-x.   2 root root 4096 Nov 30 08:51 bin
drwxr-xr-x.   2 root root    6 Apr 10  2014 boot
drwxr-xr-x.   5 root root  360 Nov 30 09:00 dev
drwxr-xr-x.   1 root root   62 Nov 30 09:00 etc
drwxr-xr-x.   2 root root    6 Apr 10  2014 home
drwxr-xr-x.  12 root root 4096 Nov 30 08:51 lib
drwxr-xr-x.   2 root root   33 Nov 30 08:51 lib64
drwxr-xr-x.   2 root root    6 Nov 23 01:30 media
drwxr-xr-x.   2 root root    6 Apr 10  2014 mnt
drwxr-xr-x.   2 root root    6 Nov 23 01:30 opt
dr-xr-xr-x. 131 root root    0 Nov 30 09:00 proc
drwx------.   2 root root   35 Nov 30 08:51 root
drwxr-xr-x.   8 root root 4096 Nov 29 20:04 run
drwxr-xr-x.   2 root root 4096 Nov 30 08:51 sbin
drwxr-xr-x.   2 root root    6 Nov 23 01:30 srv
dr-xr-xr-x.  13 root root    0 Sep  4 08:43 sys
drwxrwxrwt.   2 root root    6 Nov 23 01:32 tmp
drwxr-xr-x.  10 root root   97 Nov 30 08:51 usr
drwxr-xr-x.  11 root root 4096 Nov 30 08:51 var

ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:04
          inet addr:172.17.0.4  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:648 (648.0 B)  TX bytes:648 (648.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

exit
$
```

Start a container in background

Start a container in background using `xiaopeng163/redis:0.1` image, and the name of the container is `demo`. Through `docker ps` we can see all running Containers

```
$ docker run -d --name demo xiaopeng163/redis:0.1
4791db4ff0ef5a1ad9ff7c405bd7705d95779b2e9209967ffbef66cbaee80f3a
```

```
$ docker ps
CONTAINER ID        IMAGE                    COMMAND                  CREATED
↪    STATUS              PORTS                NAMES
4791db4ff0ef        xiaopeng163/redis:0.1    "/usr/bin/redis-serve"   5 seconds ago
↪    Up 4 seconds        6379/tcp             demo
```

stop/remove containers

Sometime, we want to manage multiple containers each time, like `start`, `stop`, `rm`.

Firstly, we can use `--filter` to filter out the containers we want to manage.

```
$ docker ps -a --filter "status=exited"
CONTAINER ID        IMAGE               COMMAND                  CREATED
↪STATUS                      PORTS               NAMES
c05d6d379459        centos:7            "/bin/bash -c 'while "   3 days ago
↪Exited (137) 11 hours ago                       test3
8975cb01d142        centos:7            "/bin/bash -c 'while "   5 days ago
↪Exited (137) 3 days ago                         test2
```

Secondly, we can use `-q` option to list only containers ids

```
$ docker ps -aq --filter "status=exited"
c05d6d379459
8975cb01d142
```

At last, we can batch processing these containers, like remove them all or start them all:

```
$ docker rm $(docker ps -aq --filter "status=exited")
c05d6d379459
8975cb01d142
```

1.2.5 Build a Base Image from Scratch

we will build a `hello world` base image from Scratch.

System Environment

Docker running on centos 7 and the version

```
$ docker version
Client:
Version:      17.12.0-ce
API version:  1.35
Go version: go1.9.2
Git commit: c97c6d6
Built:      Wed Dec 27 20:10:14 2017
OS/Arch:    linux/amd64

Server:
Engine:
Version:      17.12.0-ce
API version:  1.35 (minimum version 1.12)
Go version: go1.9.2
```

```
Git commit: c97c6d6
Built:      Wed Dec 27 20:12:46 2017
OS/Arch:    linux/amd64
Experimental:       false
```

install requirements:

```
$ sudo yum install -y gcc glibc-static
```

Create a Hello world

create a `hello.c` and save

```
$ pwd
/home/vagrant/hello-world
[vagrant@localhost hello-world]$ more hello.c
#include<stdio.h>

int main()
{
printf("hello docker\n");
}
[vagrant@localhost hello-world]$
```

Compile the `hello.c` source file to an binary file, and run it.

```
$ gcc -o hello -static  hello.c
$ ls
Dockerfile  hello  hello.c
$ ./hello
hello docker
```

Build Docker image

Create a Dockerfile like this:

```
$ more Dockerfile
FROM scratch
ADD hello /
CMD ["/hello"]
```

build image through:

```
$ docker build -t xiaopeng163/hello-world .
$ docker image ls
REPOSITORY                  TAG                 IMAGE ID            CREATED
 ↪    SIZE
xiaopeng163/hello-world     latest              78d57d4588e3        4 seconds ago
 ↪    844kB
```

Run the hello world container

```
$ docker run xiaopeng163/hello-world
hello docker
```

Done!

1.2.6 Docker Network Overview

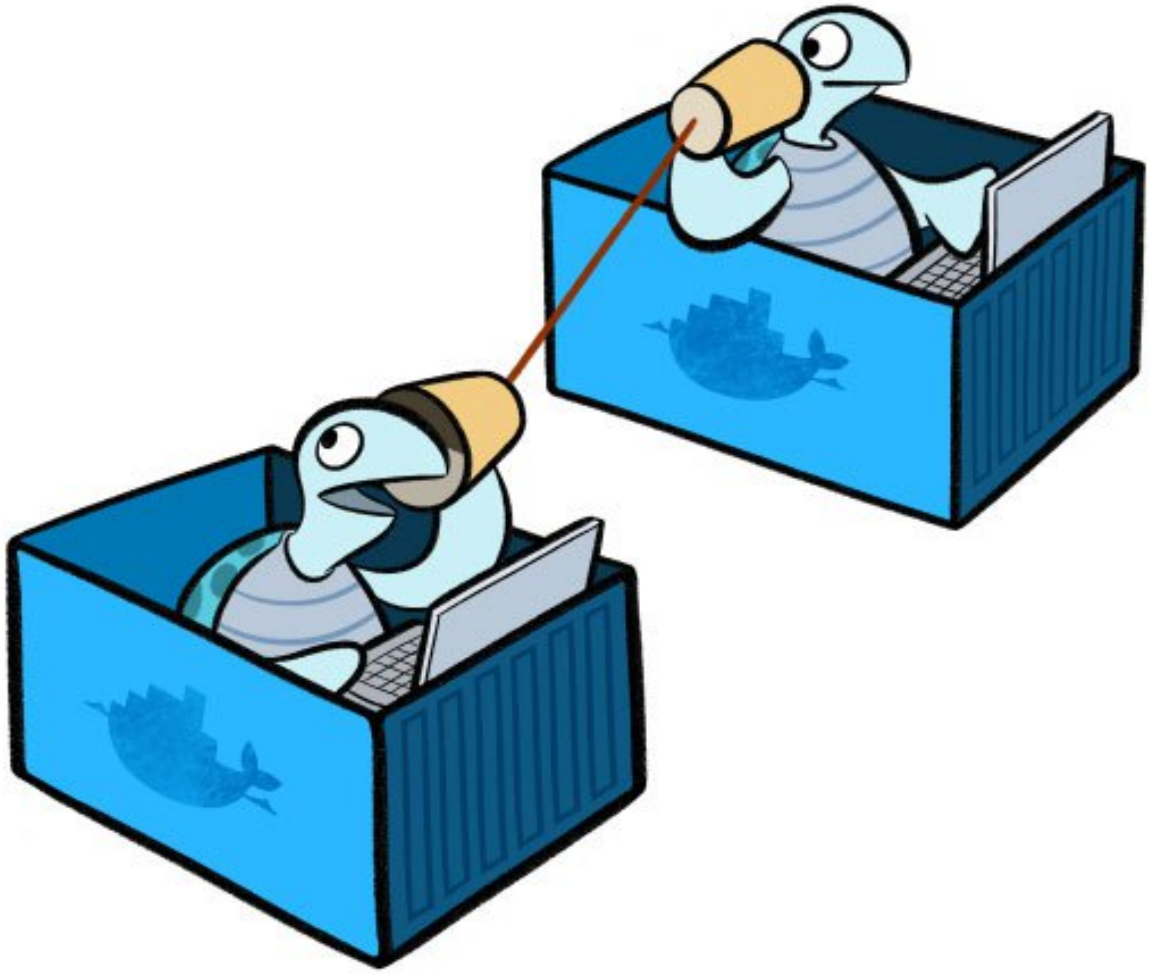


Image reference from [1]

[1] https://blog.docker.com/2015/04/docker-networking-takes-a-step-in-the-right-direction-2/

When you install Docker, it creates three networks automatically. You can list these networks using the docker network ls command:

```
$ docker network ls
NETWORK ID          NAME                DRIVER
32b93b141bae        bridge              bridge
c363d9a92877        host                host
88077db743a8        none                null
```

Reference

1.2.7 Linux Network Namespace Introduction

In this tutorial, we will learn what is Linux network namespace and how to use it.

Docker uses many Linux namespace technologies for isolation, there are user namespace, process namespace, etc. For network isolation docker uses Linux network namespace technology, each docker container has its own network namespace, which means it has its own IP address, routing table, etc.

First, let's see how to create and check a network namespace. The lab environment we used today is a docker host which is created by docker-machine tool on Amazon AWS.

Create and List Network Namespace

Use `ip netns add <network namespace name>` to create a network namespace, and `ip netns list` to list all network namepaces on the host.

```
ubuntu@docker-host-aws:~$ sudo ip netns add test1
ubuntu@docker-host-aws:~$ ip netns list
test1
ubuntu@docker-host-aws:~$
```

Delete Network Namespace

Use `ip netns delete <network namespace name>` to delete a network namespace.

```
ubuntu@docker-host-aws:~$ sudo ip netns delete test1
ubuntu@docker-host-aws:~$ ip netns list
ubuntu@docker-host-aws:~$
```

Execute CMD within Network Namespace

How to check interfaces in a particular network namespace, we can use command `ip netns exec <network namespace name> <command>` like:

```
ubuntu@docker-host-aws:~$ sudo ip netns exec test1 ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
ubuntu@docker-host-aws:~$
```

`ip a` will list all ip interfaces within this `test1` network namespaces. From the output we can see that the `lo` inteface is `DOWN`, we can run a command to let it up.

```
ubuntu@docker-host-aws:~$ sudo ip netns exec test1 ip link
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
ubuntu@docker-host-aws:~$ sudo ip netns exec test1 ip link set dev lo up
ubuntu@docker-host-aws:~$ sudo ip netns exec test1 ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT␣
↪group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

The status of lo became `UNKNOWN`, please ignore that and go on.

Add Interface to a Network Namespace

We will create a virtual interface pair, it has two virtual interfaces which are connected by a virtual cable

```
ubuntu@docker-host-aws:~$ sudo ip link add veth-a type veth peer name veth-b
ubuntu@docker-host-aws:~$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT␣
↪group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc pfifo_fast state UP mode␣
↪DEFAULT group default qlen 1000
    link/ether 02:30:c1:3e:63:3a brd ff:ff:ff:ff:ff:ff
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN␣
↪mode DEFAULT group default
    link/ether 02:42:a7:88:bd:32 brd ff:ff:ff:ff:ff:ff
27: veth-b: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group␣
↪default qlen 1000
    link/ether 52:58:31:ef:0b:98 brd ff:ff:ff:ff:ff:ff
28: veth-a: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group␣
↪default qlen 1000
    link/ether 3e:89:92:ac:ef:10 brd ff:ff:ff:ff:ff:ff
ubuntu@docker-host-aws:~$
```

All these two interfaces are located on localhost default network namespace. what we will do is move one of them to `test1` network namespace, we can do this through:

```
ubuntu@docker-host-aws:~$ sudo ip link set veth-b netns test1
ubuntu@docker-host-aws:~$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT␣
↪group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc pfifo_fast state UP mode␣
↪DEFAULT group default qlen 1000
    link/ether 02:30:c1:3e:63:3a brd ff:ff:ff:ff:ff:ff
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN␣
↪mode DEFAULT group default
    link/ether 02:42:a7:88:bd:32 brd ff:ff:ff:ff:ff:ff
28: veth-a: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group␣
↪default qlen 1000
    link/ether 3e:89:92:ac:ef:10 brd ff:ff:ff:ff:ff:ff
ubuntu@docker-host-aws:~$ sudo ip netns exec test1 ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT␣
↪group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
27: veth-b: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group␣
↪default qlen 1000
```

```
    link/ether 52:58:31:ef:0b:98 brd ff:ff:ff:ff:ff:ff
ubuntu@docker-host-aws:~$
```

Now, the interface `veth-b` is in network namespace `test1`.

Assign IP address to veth interface

In the localhost to set `veth-a`

```
ubuntu@docker-host-aws:~$ sudo ip addr add 192.168.1.1/24 dev veth-a
ubuntu@docker-host-aws:~$ sudo ip link set veth-a up
ubuntu@docker-host-aws:~$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT␣
↪group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc pfifo_fast state UP mode␣
↪DEFAULT group default qlen 1000
    link/ether 02:30:c1:3e:63:3a brd ff:ff:ff:ff:ff:ff
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN␣
↪mode DEFAULT group default
    link/ether 02:42:a7:88:bd:32 brd ff:ff:ff:ff:ff:ff
28: veth-a: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast state DOWN␣
↪mode DEFAULT group default qlen 1000
    link/ether 3e:89:92:ac:ef:10 brd ff:ff:ff:ff:ff:ff
```

`veth-a` has an IP address, but its status is DOWN. Now let's set `veth-b` in `test1`.

```
ubuntu@docker-host-aws:~$ sudo ip netns exec test1 ip addr add 192.168.1.2/24 dev␣
↪veth-b
ubuntu@docker-host-aws:~$ sudo ip netns exec test1 ip link set dev veth-b up
ubuntu@docker-host-aws:~$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT␣
↪group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc pfifo_fast state UP mode␣
↪DEFAULT group default qlen 1000
    link/ether 02:30:c1:3e:63:3a brd ff:ff:ff:ff:ff:ff
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN␣
↪mode DEFAULT group default
    link/ether 02:42:a7:88:bd:32 brd ff:ff:ff:ff:ff:ff
28: veth-a: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode␣
↪DEFAULT group default qlen 1000
    link/ether 3e:89:92:ac:ef:10 brd ff:ff:ff:ff:ff:ff
ubuntu@docker-host-aws:~$ sudo ip netns exec test1 ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT␣
↪group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
27: veth-b: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode␣
↪DEFAULT group default qlen 1000
    link/ether 52:58:31:ef:0b:98 brd ff:ff:ff:ff:ff:ff
```

After configured `veth-b` and up it, both `veth-a` and `veth-b` are UP. Now we can use `ping` to check their connectivity.

```
ubuntu@docker-host-aws:~$ ping 192.168.1.2
PING 192.168.1.2 (192.168.1.2) 56(84) bytes of data.
64 bytes from 192.168.1.2: icmp_seq=1 ttl=64 time=0.047 ms
```

```
64 bytes from 192.168.1.2: icmp_seq=2 ttl=64 time=0.046 ms
64 bytes from 192.168.1.2: icmp_seq=3 ttl=64 time=0.052 ms
^C
--- 192.168.1.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.046/0.048/0.052/0.006 ms
ubuntu@docker-host-aws:~$
```

Please go to http://www.opencloudblog.com/?p=66 to learn more.

1.2.8 Bridge Networking Deep Dive

The bridge network represents the docker0 network present in all Docker installations. Unless you specify otherwise with the `docker run --network=<NETWORK> option`, the Docker daemon connects containers to this network by default.

There are four important concepts about bridged networking:

- Docker0 Bridge
- Network Namespace
- Veth Pair
- External Communication

Docker0 bridge

Docker version for this lab:

```
$ docker version
Client:
 Version:      1.11.2
 API version:  1.23
 Go version:   go1.5.4
 Git commit:   b9f10c9
 Built:        Wed Jun  1 21:23:11 2016
 OS/Arch:      linux/amd64

Server:
 Version:      1.11.2
 API version:  1.23
 Go version:   go1.5.4
 Git commit:   b9f10c9
 Built:        Wed Jun  1 21:23:11 2016
 OS/Arch:      linux/amd64
```

Through `docker network` command we can get more details about the docker0 bridge, and from the output, we can see there is no Container connected with the bridge now.

```
$ docker network ls
NETWORK ID          NAME                DRIVER
32b93b141bae        bridge              bridge
c363d9a92877        host                host
88077db743a8        none                null
```

```
$ docker network inspect 32b93b141bae
[
    {
        "Name": "bridge",
        "Id": "32b93b141baeeac8bbf01382ec594c23515719c0d13febd8583553d70b4ecdba",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Internal": false,
        "Containers": {},
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        },
        "Labels": {}
    }
]
```

You can also see this bridge as a part of a host's network stack by using the ifconfig/ip command on the host.

```
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc pfifo_fast state UP mode␣
↪DEFAULT qlen 1000
    link/ether 06:95:4a:1f:08:7f brd ff:ff:ff:ff:ff:ff
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN␣
↪mode DEFAULT
    link/ether 02:42:d6:23:e6:18 brd ff:ff:ff:ff:ff:ff
```

Because there are no containers running, the bridge `docker0` status is down.

You can also use `brctl` command to get bridge docker0 information

```
$ brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.0242d623e618       no              veth6a5ae6f
```

Note: If you can't find `brctl` command, you can install it. For CentOS, please use `sudo yum install bridge-utils`. For Ubuntu, please use `apt-get install bridge-utils`

Veth Pair

Now we create and run a centos7 container:

```
$ docker run -d --name test1 centos:7 /bin/bash -c "while true; do sleep 3600; done"
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
 ↪STATUS              PORTS               NAMES
4fea95f2e979        centos:7            "/bin/bash -c 'while "   6 minutes ago
 ↪Up 6 minutes                           test1
```

After that we can check the ip interface in the docker host.

```
$ ip li
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc pfifo_fast state UP mode
 ↪DEFAULT qlen 1000
    link/ether 06:95:4a:1f:08:7f brd ff:ff:ff:ff:ff:ff
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode
 ↪DEFAULT
    link/ether 02:42:d6:23:e6:18 brd ff:ff:ff:ff:ff:ff
15: vethae2abb8@if14: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master
 ↪docker0 state UP mode DEFAULT
    link/ether e6:97:43:5c:33:a6 brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

The bridge `docker0` is up, and there is a veth pair created, one is in localhost, and another is in container's network namspace.

Network Namespace

If we add a new network namespace from command line.

```
$ sudo ip netns add demo
$ ip netns list
demo
$ ls /var/run/netns
demo
$ sudo ip netns exec demo ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

But from the command `ip netns list`, we can't get the container's network namespace. The reason is because docker deleted all containers network namespaces information from `/var/run/netns`.

We can get all docker container network namespace from `/var/run/docker/netns`.

```
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED
 ↪STATUS              PORTS               NAMES
4fea95f2e979        centos:7            "/bin/bash -c 'while "   2 hours ago
 ↪Up About an hour                       test1
$ sudo ls -l /var/run/docker/netns
total 0
-rw-r--r--. 1 root root 0 Nov 28 05:51 572d8e7abcb2
```

How to get the detail information (like veth) about the container network namespace?

First we should get the pid of this container process, and get all namespaces about this container.

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
 ↪STATUS              PORTS               NAMES
4fea95f2e979        centos:7            "/bin/bash -c 'while "   2 hours ago
 ↪Up 2 hours                             test1
$ docker inspect --format '{{.State.Pid}}' 4f
3090
$ sudo ls -l /proc/3090/ns
total 0
lrwxrwxrwx. 1 root root 0 Nov 28 05:52 ipc -> ipc:[4026532156]
lrwxrwxrwx. 1 root root 0 Nov 28 05:52 mnt -> mnt:[4026532154]
lrwxrwxrwx. 1 root root 0 Nov 28 05:51 net -> net:[4026532159]
lrwxrwxrwx. 1 root root 0 Nov 28 05:52 pid -> pid:[4026532157]
lrwxrwxrwx. 1 root root 0 Nov 28 08:02 user -> user:[4026531837]
lrwxrwxrwx. 1 root root 0 Nov 28 05:52 uts -> uts:[4026532155]
```

Then restore the network namespace:

```
$ sudo ln -s /proc/3090/ns/net /var/run/netns/3090
$ ip netns list
3090
demo
$ sudo ip netns exec 3090 ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
26: eth0@if27: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode
 ↪DEFAULT
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

After all is done, please remove `/var/run/netns/3090`.

External Communication

All containers connected with bridge `docker0` can communicate with the external network or other containers which connected with the same bridge.

Let's start two containers:

```
$ docker run -d --name test2 centos:7 /bin/bash -c "while true; do sleep 3600; done"
8975cb01d142271d463ec8dac43ea7586f509735d4648203319d28d46365af2f
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
 ↪STATUS              PORTS               NAMES
8975cb01d142        centos:7            "/bin/bash -c 'while "   4 seconds ago
 ↪Up 4 seconds                           test2
4fea95f2e979        centos:7            "/bin/bash -c 'while "   27 hours ago
 ↪Up 26 hours                            test1
```

And from the bridge `docker0`, we can see two interfaces connected.

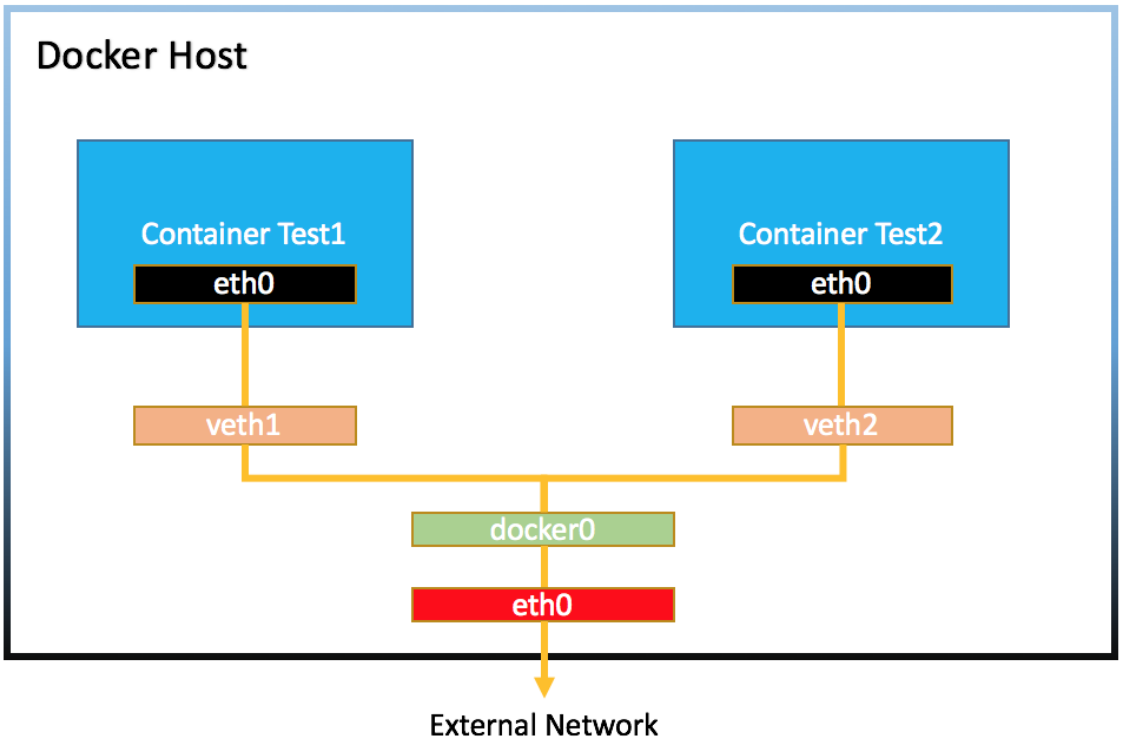
```
$ brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.0242d623e618       no              veth6a5ae6f
                                                        vethc16e6c8
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT
```

```
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc pfifo_fast state UP mode␣
↪DEFAULT qlen 1000
    link/ether 06:95:4a:1f:08:7f brd ff:ff:ff:ff:ff:ff
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode␣
↪DEFAULT
    link/ether 02:42:d6:23:e6:18 brd ff:ff:ff:ff:ff:ff
27: veth6a5ae6f@if26: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master␣
↪docker0 state UP mode DEFAULT
    link/ether 02:7d:eb:4e:85:99 brd ff:ff:ff:ff:ff:ff link-netnsid 0
31: vethc16e6c8@if30: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master␣
↪docker0 state UP mode DEFAULT
    link/ether d2:9f:2e:ca:22:a5 brd ff:ff:ff:ff:ff:ff link-netnsid 1
```

The two containers can be reached by each other

```
$  docker inspect --format '{{.NetworkSettings.IPAddress}}' test1
172.17.0.2
$  docker inspect --format '{{.NetworkSettings.IPAddress}}' test2
172.17.0.3
$ docker exec test1 bash -c 'ping 172.17.0.3'
PING 172.17.0.3 (172.17.0.3) 56(84) bytes of data.
64 bytes from 172.17.0.3: icmp_seq=1 ttl=64 time=0.051 ms
64 bytes from 172.17.0.3: icmp_seq=2 ttl=64 time=0.058 ms
64 bytes from 172.17.0.3: icmp_seq=3 ttl=64 time=0.053 ms
^C
```

The basic network would be like below:



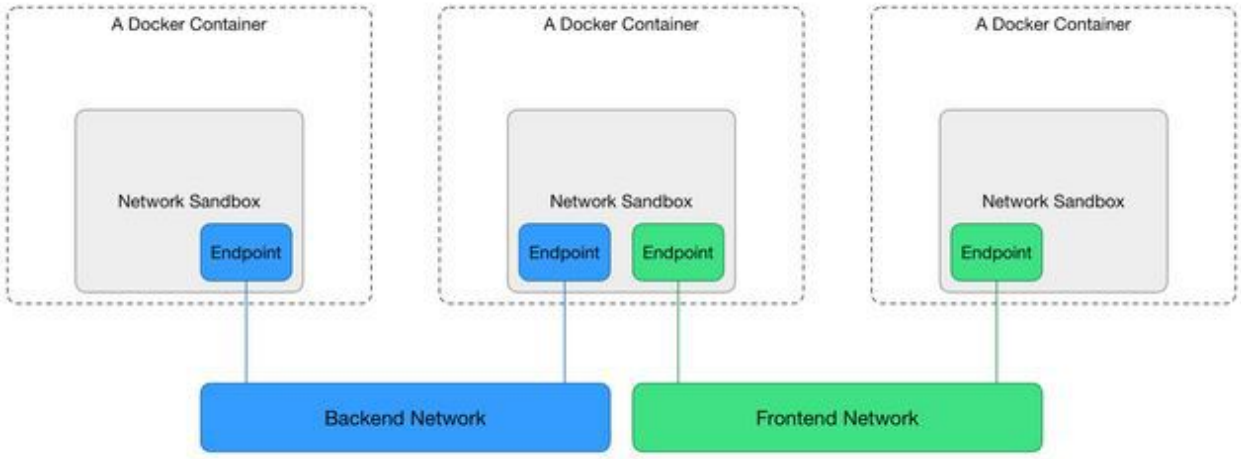
CNM

To understand how container get its ip address, you should understand what is CNM (Container Network Model) [2].

Libnetwork implements Container Network Model (CNM) which formalizes the steps required to provide networking for containers while providing an abstraction that can be used to support multiple network drivers.

During the Network and Endpoints lifecycle, the CNM model controls the IP address assignment for network and endpoint interfaces via the IPAM driver(s) [1].

When creating the bridge `docker0`, libnetwork will do some request to IPAM driver, something like network gateway, address pool. When creating a container, in the network sandbox, and endpoint was created, libnetwork will request an IPv4 address from the IPv4 pool and assign it to the endpoint interface IPv4 address.



NAT

Container in bridge network mode can access the external network through `NAT` which configured by `iptables`.

Inside the container:

```
# ping www.google.com
PING www.google.com (172.217.27.100) 56(84) bytes of data.
64 bytes from sin11s04-in-f4.1e100.net (172.217.27.100): icmp_seq=1 ttl=61 time=99.0␣
↪ms
64 bytes from sin11s04-in-f4.1e100.net (172.217.27.100): icmp_seq=2 ttl=61 time=108 ms
64 bytes from sin11s04-in-f4.1e100.net (172.217.27.100): icmp_seq=3 ttl=61 time=110 ms
^C
--- www.google.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 99.073/106.064/110.400/4.990 ms
```

From the docker host, we can see:

```
$ sudo iptables --list -t nat
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  anywhere             anywhere             ADDRTYPE match dst-type␣
↪LOCAL
```

[2] https://github.com/docker/libnetwork/blob/master/docs/design.md
[1] https://github.com/docker/libnetwork/blob/master/docs/ipam.md

```
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  anywhere            !loopback/8           ADDRTYPE match dst-type␣
↪LOCAL

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  172.17.0.0/16  anywhere

Chain DOCKER (2 references)
target     prot opt source               destination
RETURN     all  --  anywhere             anywhere
```

For NAT with iptables, you can reference [3] [4]

Reference

1.2.9 Container Port Mapping in Bridge networking

Through *Bridge Networking Deep Dive* we know that by default Docker containers can make connections to the outside world, but the outside world cannot connect to containers. Each outgoing connection will appear to originate from one of the host machine's own IP addresses thanks to an iptables masquerading rule on the host machine that the Docker server creates when it starts: [1]

```
ubuntu@docker-node1:~$ sudo iptables -t nat -L -n
...
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  172.17.0.0/16        0.0.0.0/0
...
ubuntu@docker-node1:~$ ifconfig docker0
docker0   Link encap:Ethernet  HWaddr 02:42:58:22:4c:30
          inet addr:172.17.0.1  Bcast:0.0.0.0  Mask:255.255.0.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

ubuntu@docker-node1:~$
```

The Docker server creates a `masquerade` rule that let containers connect to IP addresses in the outside world.

Bind Container port to the host

Start a nginx container which export port 80 and 443. we can access the port from inside of the docker host.

[3] http://www.karlrupp.net/en/computer/nat_tutorial
[4] https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/4/html/Security_Guide/s1-firewall-ipt-fwd.html
[1] https://docs.docker.com/engine/userguide/networking/default_network/binding/

```
ubuntu@docker-node1:~$ sudo docker run -d  --name demo nginx
ubuntu@docker-node1:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
 ↪STATUS              PORTS               NAMES
b5e53067e12f        nginx               "nginx -g 'daemon off"   8 minutes ago
 ↪Up 8 minutes        80/tcp, 443/tcp     demo
ubuntu@docker-node1:~$ sudo docker inspect --format {{.NetworkSettings.IPAddress}}
 ↪demo
172.17.0.2
ubuntu@docker-node1:~$ curl 172.17.0.2
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

If we want to access the nginx web from outside of the docker host, we must bind the port to docker host like this:

```
ubuntu@docker-node1:~$ sudo docker run -d  -p 80 --name demo nginx
0fb783dcd5b3010c0ef47e4c929dfe0c9eac8ddec2e5e0470df5529bfd4cb64e
ubuntu@docker-node1:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
 ↪STATUS              PORTS                                NAMES
0fb783dcd5b3        nginx               "nginx -g 'daemon off"   5 seconds ago
 ↪Up 5 seconds        443/tcp, 0.0.0.0:32768->80/tcp   demo
ubuntu@docker-node1:~$ curl 192.168.205.10:32768
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
```

```
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
ubuntu@docker-node1:~$ ifconfig enp0s8
enp0s8    Link encap:Ethernet  HWaddr 08:00:27:7a:ac:d2
          inet addr:192.168.205.10  Bcast:192.168.205.255  Mask:255.255.255.0
          inet6 addr: fe80::a00:27ff:fe7a:acd2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:648 (648.0 B)

ubuntu@docker-node1:~$
```

If we want to point out which port on host want to bind:

```
ubuntu@docker-node1:~$ sudo docker run -d  -p 80:80 --name demo1 nginx
4f548139a4be6574e3f9718f99a05e5174bdfb62d229ea656d35a979b5b0507d
ubuntu@docker-node1:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
 ↪STATUS              PORTS                                NAMES
4f548139a4be        nginx               "nginx -g 'daemon off"   5 seconds ago
 ↪Up 4 seconds        0.0.0.0:80->80/tcp, 443/tcp          demo1
0fb783dcd5b3        nginx               "nginx -g 'daemon off"   2 minutes ago
 ↪Up 2 minutes        443/tcp, 0.0.0.0:32768->80/tcp       demo
ubuntu@docker-node1:~$
```

What happened

It's iptables

```
ubuntu@docker-node1:~$ sudo iptables -t nat -L -n
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0            0.0.0.0/0            ADDRTYPE match dst-type
 ↪LOCAL

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0           !127.0.0.0/8          ADDRTYPE match dst-type
 ↪LOCAL
```

```
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  172.17.0.0/16        0.0.0.0/0
MASQUERADE  tcp  --  172.17.0.2           172.17.0.2           tcp dpt:80
MASQUERADE  tcp  --  172.17.0.3           172.17.0.3           tcp dpt:80

Chain DOCKER (2 references)
target     prot opt source               destination
RETURN     all  --  0.0.0.0/0            0.0.0.0/0
DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0            tcp dpt:32768 to:172.17.
↪0.2:80
DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0            tcp dpt:80 to:172.17.0.
↪3:80
ubuntu@docker-node1:~$

ubuntu@docker-node1:~$ sudo iptables -t nat -nvxL
Chain PREROUTING (policy ACCEPT 0 packets, 0 bytes)
    pkts      bytes target     prot opt in     out     source               ␣
↪destination
       1       44 DOCKER     all  --  *      *       0.0.0.0/0            0.0.0.0/0  ␣
↪          ADDRTYPE match dst-type LOCAL

Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
    pkts      bytes target     prot opt in     out     source               ␣
↪destination

Chain OUTPUT (policy ACCEPT 0 packets, 0 bytes)
    pkts      bytes target     prot opt in     out     source               ␣
↪destination
       4      240 DOCKER     all  --  *      *       0.0.0.0/0           !127.0.0.0/8␣
↪          ADDRTYPE match dst-type LOCAL

Chain POSTROUTING (policy ACCEPT 2 packets, 120 bytes)
    pkts      bytes target     prot opt in     out     source               ␣
↪destination
       0        0 MASQUERADE  all  --  *      !docker0  172.17.0.0/16        0.0.0.0/0
       0        0 MASQUERADE  tcp  --  *      *       172.17.0.2           172.17.0.2␣
↪           tcp dpt:80
       0        0 MASQUERADE  tcp  --  *      *       172.17.0.3           172.17.0.3␣
↪           tcp dpt:80

Chain DOCKER (2 references)
    pkts      bytes target     prot opt in     out     source               ␣
↪destination
       0        0 RETURN     all  --  docker0 *       0.0.0.0/0            0.0.0.0/0
       1       60 DNAT       tcp  --  !docker0 *       0.0.0.0/0            0.0.0.0/0␣
↪           tcp dpt:32768 to:172.17.0.2:80
       2      120 DNAT       tcp  --  !docker0 *       0.0.0.0/0            0.0.0.0/0␣
↪           tcp dpt:80 to:172.17.0.3:80
ubuntu@docker-node1:~$
```

References

1.2.10 Customize the docker0 bridge

The default docker0 bridge has some default configuration [1].

```
ubuntu@docker-node1:~$ docker network list
NETWORK ID          NAME                DRIVER              SCOPE
83a58f039549        bridge              bridge              local
0f93d7177516        host                host                local
68721ff2f526        none                null                local
ubuntu@docker-node1:~$
ubuntu@docker-node1:~$
ubuntu@docker-node1:~$ docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "83a58f039549470e3374c6631ef721b927e92917af1d21b464dd59551025ac22",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Internal": false,
        "Containers": {
            "13866c4e5bf2c73385883090ccd0b64ca6ff177d61174f4499210b8a17a7def1": {
                "Name": "test1",
                "EndpointID":
↪"99fea9853df1fb5fbed3f927b3d2b00544188aa7913a8c0f4cb9f9a40639d789",
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        },
        "Labels": {}
    }
]
ubuntu@docker-node1:~$
```

What we want to do is to change the default IPAM dirver's configuration, IP address, netmask and IP allocation range.

[1] https://docs.docker.com/engine/userguide/networking/default_network/custom-docker0/

References

1.2.11 Create a new bridge network and connect with container

Lab Environments

We use the docker hosts created by docker-machine on Amazon AWS.

```
$ docker-machine ls
NAME              ACTIVE   DRIVER      STATE     URL                         SWARM
↪DOCKER    ERRORS
docker-host-aws   -        amazonec2   Running   tcp://52.53.176.55:2376           v1.
↪13.0
(docker-k8s-lab)  docker-k8s-lab git:(master)  docker ssh docker-host-aws
docker: 'ssh' is not a docker command.
See 'docker --help'
$ docker-machine ssh docker-host-aws
ubuntu@docker-host-aws:~$ docker version
Client:
 Version:      1.13.0
 API version:  1.25
 Go version:   go1.7.3
 Git commit:   49bf474
 Built:        Tue Jan 17 09:50:17 2017
 OS/Arch:      linux/amd64

Server:
 Version:      1.13.0
 API version:  1.25 (minimum version 1.12)
 Go version:   go1.7.3
 Git commit:   49bf474
 Built:        Tue Jan 17 09:50:17 2017
 OS/Arch:      linux/amd64
 Experimental: false
ubuntu@docker-host-aws:~$
```

Create a new Bridge Network

Use `docker network create -d bridge NETWORK_NAME` command to create a new bridge network [1].

```
ubuntu@docker-host-aws:~$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
326ddef352c5        bridge              bridge              local
28cc7c021812        demo                bridge              local
1ca18e6b4867        host                host                local
e9530f1fb046        none                null                local
ubuntu@docker-host-aws:~$ docker network rm demo
demo
ubuntu@docker-host-aws:~$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
326ddef352c5        bridge              bridge              local
1ca18e6b4867        host                host                local
e9530f1fb046        none                null                local
ubuntu@docker-host-aws:~$ docker network create -d bridge my-bridge
```

[1] https://docs.docker.com/engine/reference/commandline/network_create/

```
e0fc5f7ff50e97787a7b13064f12806232dcc88bafa9c2eb07cec5e81cefd886
ubuntu@docker-host-aws:~$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
326ddef352c5        bridge              bridge              local
1ca18e6b4867        host                host                local
e0fc5f7ff50e        my-bridge           bridge              local
e9530f1fb046        none                null                local
ubuntu@docker-host-aws:~$
ubuntu@docker-host-aws:~$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc pfifo_fast state UP group␣
↪default qlen 1000
    link/ether 02:30:c1:3e:63:3a brd ff:ff:ff:ff:ff:ff
    inet 172.31.29.93/20 brd 172.31.31.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::30:c1ff:fe3e:633a/64 scope link
       valid_lft forever preferred_lft forever
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN␣
↪group default
    link/ether 02:42:a7:88:bd:32 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:a7ff:fe88:bd32/64 scope link
       valid_lft forever preferred_lft forever
56: br-e0fc5f7ff50e: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state␣
↪DOWN group default
    link/ether 02:42:c0:80:09:3c brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.1/16 scope global br-e0fc5f7ff50e
       valid_lft forever preferred_lft forever
ubuntu@docker-host-aws:~$ brctl show
bridge name bridge id   STP enabled interfaces
br-e0fc5f7ff50e   8000.0242c080093c no
docker0   8000.0242a788bd32 no
ubuntu@docker-host-aws:~$
```

Create a Container connected with new Bridge

Create a container connected with the `my-bridge` network.

```
$ docker run -d --name test1 --network my-bridge busybox sh -c "while true;do sleep␣
↪3600;done"
$ docker exec -it test1 sh
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
57: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:12:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.2/16 scope global eth0
```

```
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe12:2/64 scope link
       valid_lft forever preferred_lft forever

ubuntu@docker-host-aws:~$ brctl show
bridge name bridge id   STP enabled interfaces
br-e0fc5f7ff50e   8000.0242c080093c no    veth2f36f74
docker0   8000.0242a788bd32 no
ubuntu@docker-host-aws:~$
```

The new container will connect with the `my-bridge`.

Change a Container's network

Create two containers which connect with the default `docker0` bridge.

```
ubuntu@docker-host-aws:~$ docker run -d --name test1  busybox sh -c "while true;do␣
↪sleep 3600;done"
73624dd5373b594526d73a1d6fb68a32b92c1ed75e84575f32e4e0f2e1d8d356
ubuntu@docker-host-aws:~$ docker run -d --name test2  busybox sh -c "while true;do␣
↪sleep 3600;done"
33498192d489832a8534fb516029be7fbaf0b58e665d3e4922147857ffbbc10b
```

Create a new bridge network

```
ubuntu@docker-host-aws:~$ docker network create -d bridge demo-bridge
be9309ebb3b3fc18c3d43b0fef7c82fe348ce7bf841e281934deccf6bd6e51eb
```

Use `docker network connect demo-bridge test1` command to connect container `test1` to bridge `demo-bridge`.

```
ubuntu@docker-host-aws:~$ docker network connect demo-bridge test1
ubuntu@docker-host-aws:~$ brctl show
bridge name bridge id   STP enabled interfaces
br-be9309ebb3b3   8000.02423906b898 no    vethec7dc1d
docker0   8000.0242a788bd32 no    veth3238a5d
              veth7b516dd
ubuntu@docker-host-aws:~$ docker network inspect demo-bridge
[
    {
        "Name": "demo-bridge",
        "Id": "be9309ebb3b3fc18c3d43b0fef7c82fe348ce7bf841e281934deccf6bd6e51eb",
        "Created": "2017-02-23T06:16:28.251575297Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
                }
            ]
        },
        "Internal": false,
```

```
        "Attachable": false,
        "Containers": {
            "73624dd5373b594526d73a1d6fb68a32b92c1ed75e84575f32e4e0f2e1d8d356": {
                "Name": "test1",
                "EndpointID":
↪"b766bfcc7fc851620b63931f114f5b81b5e072c7ffd64d8f1c99d9828810f17a",
                "MacAddress": "02:42:ac:12:00:02",
                "IPv4Address": "172.18.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {}
    }
]
```

Now the container `test1` has connected with the default `docker0` bridge and `demo-bridge`. we can do them same action to connect container `test2` to `demo-bridge` network. After that:

```
ubuntu@docker-host-aws:~$ brctl show
bridge name bridge id   STP enabled interfaces
br-be9309ebb3b3   8000.02423906b898 no    veth67bd1b0
                vethec7dc1d
docker0   8000.0242a788bd32 no    veth3238a5d
                veth7b516dd
ubuntu@docker-host-aws:~$ docker network inspect demo-bridge
[
    {
        "Name": "demo-bridge",
        "Id": "be9309ebb3b3fc18c3d43b0fef7c82fe348ce7bf841e281934deccf6bd6e51eb",
        "Created": "2017-02-23T06:16:28.251575297Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Containers": {
            "33498192d489832a8534fb516029be7fbaf0b58e665d3e4922147857ffbbc10b": {
                "Name": "test2",
                "EndpointID":
↪"26d6bdc1c1c0459ba49718e07d6983a9dda1a1a96db3f1beedcbc5ea54abd163",
                "MacAddress": "02:42:ac:12:00:03",
                "IPv4Address": "172.18.0.3/16",
                "IPv6Address": ""
            },
            "73624dd5373b594526d73a1d6fb68a32b92c1ed75e84575f32e4e0f2e1d8d356": {
                "Name": "test1",
                "EndpointID":
↪"b766bfcc7fc851620b63931f114f5b81b5e072c7ffd64d8f1c99d9828810f17a",
```

```
                "MacAddress": "02:42:ac:12:00:02",
                "IPv4Address": "172.18.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {}
    }
]
```

Now, if we go into `test1`, we can ping `test2` directly by container name:

```
ubuntu@docker-host-aws:~$ docker exec -it test1 sh
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
78: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:2/64 scope link
       valid_lft forever preferred_lft forever
83: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:12:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.2/16 scope global eth1
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe12:2/64 scope link
       valid_lft forever preferred_lft forever
/ # ping test2
PING test2 (172.18.0.3): 56 data bytes
64 bytes from 172.18.0.3: seq=0 ttl=64 time=0.095 ms
64 bytes from 172.18.0.3: seq=1 ttl=64 time=0.077 ms
^C
--- test2 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.077/0.086/0.095 ms
```

Also, we can use `docker network disconnect demo-bridge test1` to disconnect container `test1` from network `demo-bridge`.

Reference

1.2.12 Host Network Deep Dive

In host network mode, the container and the host will be in the same network namespace.

Docker version for this lab:

```
$ docker version
Client:
 Version:      1.11.2
 API version:  1.23
 Go version:   go1.5.4
```

```
 Git commit:   b9f10c9
 Built:        Wed Jun  1 21:23:11 2016
 OS/Arch:      linux/amd64

Server:
 Version:      1.11.2
 API version:  1.23
 Go version:   go1.5.4
 Git commit:   b9f10c9
 Built:        Wed Jun  1 21:23:11 2016
 OS/Arch:      linux/amd64
 docker
```

Start a container in host network mode with `--net=host`.

```
$ docker run -d --name test3 --net=host centos:7 /bin/bash -c "while true; do sleep␣
↪3600; done"
c05d6d379459a651dbd6a98606328236063c541842db5e456767c219e2c52716
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc pfifo_fast state UP mode␣
↪DEFAULT qlen 1000
    link/ether 06:95:4a:1f:08:7f brd ff:ff:ff:ff:ff:ff
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN␣
↪mode DEFAULT
    link/ether 02:42:d6:23:e6:18 brd ff:ff:ff:ff:ff:ff
$ docker network inspect host
[
    {
        "Name": "host",
        "Id": "c363d9a92877e78cb33e7e5dd7884babfd6d05ae2100162fca21f756fe340b79",
        "Scope": "local",
        "Driver": "host",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": []
        },
        "Internal": false,
        "Containers": {
            "c05d6d379459a651dbd6a98606328236063c541842db5e456767c219e2c52716": {
                "Name": "test3",
                "EndpointID":
↪"929c58100f6e4356eadccbe2f44bf1ce4056776359426683125 9d012cd76e4d6",
                "MacAddress": "",
                "IPv4Address": "",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {}
    }
]
```

Unlike bridge network mode, there is no veth pair. Go to the inside of the container.

```
$ docker exec -it test3 bash
# yum install net-tools -y
# ifconfig
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 0.0.0.0
        inet6 fe80::42:d6ff:fe23:e618  prefixlen 64  scopeid 0x20<link>
        ether 02:42:d6:23:e6:18  txqueuelen 0  (Ethernet)
        RX packets 6624  bytes 359995 (351.5 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 11019  bytes 16432384 (15.6 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 9001
        inet 172.31.43.155  netmask 255.255.240.0  broadcast 172.31.47.255
        inet6 fe80::495:4aff:fe1f:87f  prefixlen 64  scopeid 0x20<link>
        ether 06:95:4a:1f:08:7f  txqueuelen 1000  (Ethernet)
        RX packets 1982838  bytes 765628507 (730.1 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 2689881  bytes 330857410 (315.5 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 0  (Local Loopback)
        RX packets 6349  bytes 8535636 (8.1 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 6349  bytes 8535636 (8.1 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
# ping www.google.com
PING www.google.com (172.217.3.196) 56(84) bytes of data.
64 bytes from sea15s12-in-f196.1e100.net (172.217.3.196): icmp_seq=1 ttl=43 time=7.34␣
↪ms
64 bytes from sea15s12-in-f4.1e100.net (172.217.3.196): icmp_seq=2 ttl=43 time=7.35 ms
^C
--- www.google.com ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 7.342/7.346/7.350/0.004 ms
```

The container has the same ip/mac address as the host. we see that when using host mode networking, the container effectively inherits the IP address from its host. This mode is faster than the bridge mode (because there is no routing overhead), but it exposes the container directly to the public network, with all its security implications [1].

Reference

1.2.13 Multi-Host Overlay Networking with Etcd

Docker has a build-in overlay networking driver, and it is used by default when docker running in swarm mode [1].

Note: The Docker Overlay driver has existed since Docker Engine 1.9, and an external K/V store was required to manage state for the network. Docker Engine 1.12 integrated the control plane state into Docker Engine so that an external store is no longer required. 1.12 also introduced several new features including encryption and service load

[1] https://www.oreilly.com/learning/what-is-docker-networking
[1] https://docs.docker.com/engine/swarm/swarm-mode/

balancing. Networking features that are introduced require a Docker Engine version that supports them, and using these features with older versions of Docker Engine is not supported.

This lab we will not run docker in swarm mode, but use docker engine with external key-value store to do multi-host overlay networking.

We chose etcd [2] as our external key-value store. You can trade etcd cluster as the management plane in this multi-host networking.

For data plane, The Docker overlay network encapsulates container traffic in a VXLAN header which allows the traffic to traverse the physical Layer 2 or Layer 3 network.

Note: VXLAN has been a part of the Linux kernel since version 3.7, and Docker uses the native VXLAN features of the kernel to create overlay networks. The Docker overlay datapath is entirely in kernel space. This results in fewer context switches, less CPU overhead, and a low-latency, direct traffic path between applications and the physical NIC.

Prepare Environment

Create a `etcd` two node cluster [3]. On docker-node1:

```
ubuntu@docker-node1:~$ wget https://github.com/coreos/etcd/releases/download/v3.0.12/
↪etcd-v3.0.12-linux-amd64.tar.gz
ubuntu@docker-node1:~$ tar zxvf etcd-v3.0.12-linux-amd64.tar.gz
ubuntu@docker-node1:~$ cd etcd-v3.0.12-linux-amd64
ubuntu@docker-node1:~$ nohup ./etcd --name docker-node1 --initial-advertise-peer-urls␣
↪http://192.168.205.10:2380 \
--listen-peer-urls http://192.168.205.10:2380 \
--listen-client-urls http://192.168.205.10:2379,http://127.0.0.1:2379 \
--advertise-client-urls http://192.168.205.10:2379 \
--initial-cluster-token etcd-cluster \
--initial-cluster docker-node1=http://192.168.205.10:2380,docker-node2=http://192.168.
↪205.11:2380 \
--initial-cluster-state new&
```

On docker-node2, start etcd and check cluster status through cmd `./etcdctl cluster-health`.

```
ubuntu@docker-node2:~$ wget https://github.com/coreos/etcd/releases/download/v3.0.12/
↪etcd-v3.0.12-linux-amd64.tar.gz
ubuntu@docker-node2:~$ tar zxvf etcd-v3.0.12-linux-amd64.tar.gz
ubuntu@docker-node2:~$ cd etcd-v3.0.12-linux-amd64/
ubuntu@docker-node2:~$ nohup ./etcd --name docker-node2 --initial-advertise-peer-urls␣
↪http://192.168.205.11:2380 \
--listen-peer-urls http://192.168.205.11:2380 \
--listen-client-urls http://192.168.205.11:2379,http://127.0.0.1:2379 \
--advertise-client-urls http://192.168.205.11:2379 \
--initial-cluster-token etcd-cluster \
--initial-cluster docker-node1=http://192.168.205.10:2380,docker-node2=http://192.168.
↪205.11:2380 \
--initial-cluster-state new&
ubuntu@docker-node2:~/etcd-v3.0.12-linux-amd64$ ./etcdctl cluster-health
member 21eca106efe4caee is healthy: got healthy result from http://192.168.205.10:2379
member 8614974c83d1cc6d is healthy: got healthy result from http://192.168.205.11:2379
cluster is healthy
```

[2] https://github.com/coreos/etcd
[3] https://coreos.com/etcd/docs/latest/op-guide/clustering.html

Restart docker engine with cluster configuration

on docker-node1

if docker version < 17.09

```
ubuntu@docker-node1:~$ sudo service docker stop
ubuntu@docker-node1:~$ sudo /usr/bin/docker daemon -H tcp://0.0.0.0:2375 -H unix:///
↪var/run/docker.sock --cluster-store=etcd://192.168.205.10:2379 --cluster-
↪advertise=192.168.205.10:2375
```

if docker version >= 17.09

```
ubuntu@docker-node1:~$ sudo service docker stop
ubuntu@docker-node1:~$ sudo /usr/bin/dockerd  -H tcp://0.0.0.0:2375 -H unix:///var/
↪run/docker.sock --cluster-store=etcd://192.168.205.10:2379 --cluster-advertise=192.
↪168.205.10:2375
```

On docker-node2

```
ubuntu@docker-node2:~$ sudo service docker stop
ubuntu@docker-node2:~$ sudo /usr/bin/docker daemon -H tcp://0.0.0.0:2375 -H unix:///
↪var/run/docker.sock --cluster-store=etcd://192.168.205.11:2379 --cluster-
↪advertise=192.168.205.11:2375
```

Create Overlay Network

On docker-node1, we create a new network whose driver is overlay.

```
ubuntu@docker-node1:~$ sudo docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
0e7bef3f143a        bridge              bridge              local
a5c7daf62325        host                host                local
3198cae88ab4        none                null                local
ubuntu@docker-node1:~$ sudo docker network create -d overlay demo
3d430f3338a2c3496e9edeccc880f0a7affa06522b4249497ef6c4cd6571eaa9
ubuntu@docker-node1:~$ sudo docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
0e7bef3f143a        bridge              bridge              local
3d430f3338a2        demo                overlay             global
a5c7daf62325        host                host                local
3198cae88ab4        none                null                local
ubuntu@docker-node1:~$ sudo docker network inspect demo
[
    {
        "Name": "demo",
        "Id": "3d430f3338a2c3496e9edeccc880f0a7affa06522b4249497ef6c4cd6571eaa9",
        "Scope": "global",
        "Driver": "overlay",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
```

```
                {
                    "Subnet": "10.0.0.0/24",
                    "Gateway": "10.0.0.1/24"
                }
            ]
        },
        "Internal": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
```

On docker-node2, we can see the demo network is added automatically.

```
ubuntu@docker-node2:~$ sudo docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
c9947d4c3669        bridge              bridge              local
3d430f3338a2        demo                overlay             global
fa5168034de1        host                host                local
c2ca34abec2a        none                null                local
```

What happened? It's done through `etcd`. Check etcd key-value on node2

```
ubuntu@docker-node2:~/etcd-v3.0.12-linux-amd64$ ./etcdctl ls /docker
/docker/network
/docker/nodes
ubuntu@docker-node2:~/etcd-v3.0.12-linux-amd64$ ./etcdctl ls /docker/nodes
/docker/nodes/192.168.205.11:2375
/docker/nodes/192.168.205.10:2375
ubuntu@docker-node2:~/etcd-v3.0.12-linux-amd64$ ./etcdctl ls /docker/network/v1.0/
↪network
/docker/network/v1.0/network/
↪3d430f3338a2c3496e9edeccc880f0a7affa06522b4249497ef6c4cd6571eaa9
ubuntu@docker-node2:~/etcd-v3.0.12-linux-amd64$ ./etcdctl get /docker/network/v1.0/
↪network/3d430f3338a2c3496e9edeccc880f0a7affa06522b4249497ef6c4cd6571eaa9 | jq .
{
  "addrSpace": "GlobalDefault",
  "enableIPv6": false,
  "generic": {
    "com.docker.network.enable_ipv6": false,
    "com.docker.network.generic": {}
  },
  "id": "3d430f3338a2c3496e9edeccc880f0a7affa06522b4249497ef6c4cd6571eaa9",
  "inDelete": false,
  "ingress": false,
  "internal": false,
  "ipamOptions": {},
  "ipamType": "default",
  "ipamV4Config": "[{\"PreferredPool\":\"\",\"SubPool\":\"\",\"Gateway\":\"\",\
↪"AuxAddresses\":null}]",
  "ipamV4Info": "[{\"IPAMData\":\"{\\\"AddressSpace\\\":\\\"GlobalDefault\\\",\\\
↪"Gateway\\\":\\\"10.0.0.1/24\\\",\\\"Pool\\\":\\\"10.0.0.0/24\\\"}\",\"PoolID\":\
↪"GlobalDefault/10.0.0.0/24\"}]",
  "labels": {},
  "name": "demo",
  "networkType": "overlay",
  "persist": true,
```

```
  "postIPv6": false,
  "scope": "global"
}
```

The network ID `3d430f3338a2c3496e9edeccc880f0a7affa06522b4249497ef6c4cd6571eaa9` is exactly the ID you see from `docker network ls`. So all the information is synchronized by etcd.

```
ubuntu@docker-node1:~$ sudo docker exec test1 ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
53: eth0@if54: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1450 qdisc noqueue
    link/ether 02:42:0a:00:00:02 brd ff:ff:ff:ff:ff:ff
55: eth1@if56: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:12:00:02 brd ff:ff:ff:ff:ff:ff
```

Start Containers With Overlay Network

On docker-node1:

```
ubuntu@docker-node1:~$ sudo docker run -d --name test1 --net demo busybox sh -c
↪"while true; do sleep 3600; done"
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
56bec22e3559: Pull complete
Digest: sha256:29f5d56d12684887bdfa50dcd29fc31eea4aaf4ad3bec43daf19026a7ce69912
Status: Downloaded newer image for busybox:latest
a95a9466331dd9305f9f3c30e7330b5a41aae64afda78f038fc9e04900fcac54
ubuntu@docker-node1:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
↪STATUS              PORTS               NAMES
a95a9466331d        busybox             "sh -c 'while true; d"   4 seconds ago
↪Up 3 seconds                            test1
ubuntu@docker-node1:~$ sudo docker exec test1 ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:0A:00:00:02
          inet addr:10.0.0.2  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::42:aff:fe00:2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
          RX packets:15 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1206 (1.1 KiB)  TX bytes:648 (648.0 B)

eth1      Link encap:Ethernet  HWaddr 02:42:AC:12:00:02
          inet addr:172.18.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe12:2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:648 (648.0 B)  TX bytes:648 (648.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
```

```
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

ubuntu@docker-node1:~$
```

On docker-node2:

```
ubuntu@docker-node2:~$ sudo docker run -d --name test1 --net demo busybox sh -c
 ↪"while true; do sleep 3600; done"
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
56bec22e3559: Pull complete
Digest: sha256:29f5d56d12684887bdfa50dcd29fc31eea4aaf4ad3bec43daf19026a7ce69912
Status: Downloaded newer image for busybox:latest
fad6dc6538a85d3dcc958e8ed7b1ec3810feee3e454c1d3f4e53ba25429b290b
docker: Error response from daemon: service endpoint with name test1 already exists.
ubuntu@docker-node2:~$ sudo docker run -d --name test2 --net demo busybox sh -c
 ↪"while true; do sleep 3600; done"
9d494a2f66a69e6b861961d0c6af2446265bec9b1d273d7e70d0e46eb2e98d20
```

We can see that if we create a container named test1, it return an error: test1 already exists. The reason is that the two hosts share configurations through etcd.

Through etcd

```
ubuntu@docker-node2:~/etcd-v3.0.12-linux-amd64$ ./etcdctl get /docker/network/v1.0/
 ↪endpoint/3d430f3338a2c3496e9edeccc880f0a7affa06522b4249497ef6c4cd6571eaa9/
 ↪57aec8a581a7f664faad9bae6c48437289b0376512bbfe9a9ecb9d18496b3c61 | jq .
{
  "anonymous": false,
  "disableResolution": false,
  "ep_iface": {
    "addr": "10.0.0.2/24",
    "dstPrefix": "eth",
    "mac": "02:42:0a:00:00:02",
    "routes": null,
    "srcName": "veth9337a4a",
    "v4PoolID": "GlobalDefault/10.0.0.0/24",
    "v6PoolID": ""
  },
  "exposed_ports": [],
  "generic": {
    "com.docker.network.endpoint.exposedports": [],
    "com.docker.network.portmap": []
  },
  "id": "57aec8a581a7f664faad9bae6c48437289b0376512bbfe9a9ecb9d18496b3c61",
  "ingressPorts": null,
  "joinInfo": {
    "StaticRoutes": null,
    "disableGatewayService": false
  },
  "locator": "192.168.205.10",
  "myAliases": [
    "a95a9466331d"
  ],
  "name": "test1",
  "sandbox": "fb8288acaf2169ff12230293dea6ec508387c3fb06ade120ba2c4283b3e88a6b",
```

```
  "svcAliases": null,
  "svcID": "",
  "svcName": "",
  "virtualIP": "<nil>"
}
  ubuntu@docker-node2:~/etcd-v3.0.12-linux-amd64$
```

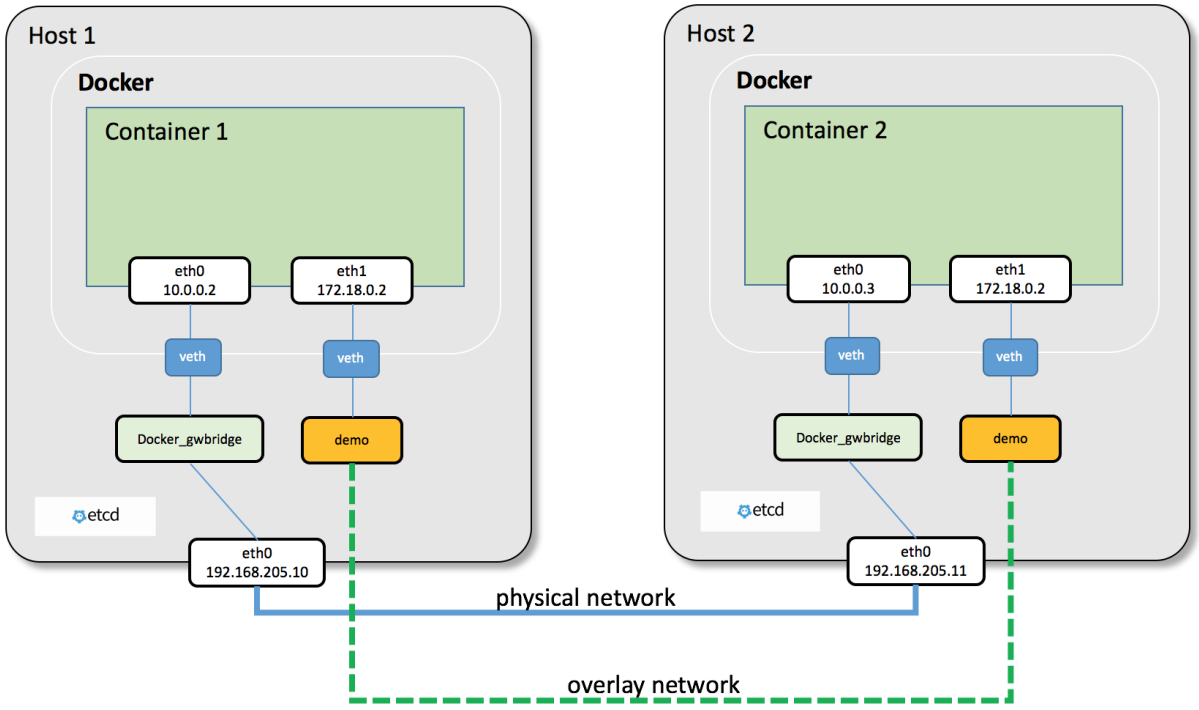
The ip and mac address is container test1.

Let check the connectivity.

```
ubuntu@docker-node2:~$ sudo docker exec -it test2 ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:0A:00:00:03
          inet addr:10.0.0.3  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::42:aff:fe00:3/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
          RX packets:208 errors:0 dropped:0 overruns:0 frame:0
          TX packets:201 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:20008 (19.5 KiB)  TX bytes:19450 (18.9 KiB)

eth1      Link encap:Ethernet  HWaddr 02:42:AC:12:00:02
          inet addr:172.18.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe12:2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:648 (648.0 B)  TX bytes:648 (648.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

  ubuntu@docker-node1:~$ sudo docker exec test1 sh -c "ping 10.0.0.3"
  PING 10.0.0.3 (10.0.0.3): 56 data bytes
  64 bytes from 10.0.0.3: seq=0 ttl=64 time=0.579 ms
  64 bytes from 10.0.0.3: seq=1 ttl=64 time=0.411 ms
  64 bytes from 10.0.0.3: seq=2 ttl=64 time=0.483 ms
  ^C
  ubuntu@docker-node1:~$
```

Analysis [4] [5]

[4] https://github.com/docker/labs/blob/master/networking/concepts/06-overlay-networks.md
[5] https://www.singlestoneconsulting.com/~/media/files/whitepapers/dockernetworking2.pdf



During overlay network creation, Docker Engine creates the network infrastructure required for overlays on each host (Create on one host, and through etcd sync to the other host). A Linux bridge is created per overlay along with its associated VXLAN interfaces. The Docker Engine intelligently instantiates overlay networks on hosts only when a container attached to that network is scheduled on the host. This prevents sprawl of overlay networks where connected containers do not exist.

There are two interfaces in each container, one is for `docker_gwbridge` network, and the other is for `demo` overlay network.

Reference

1.2.14 Multi-Host Overlay Networking with Open vSwitch

Note: Using OVS is not a good choice, because there are many problems need to resolve, like IP management, external routing. So we do not recommand this solution.

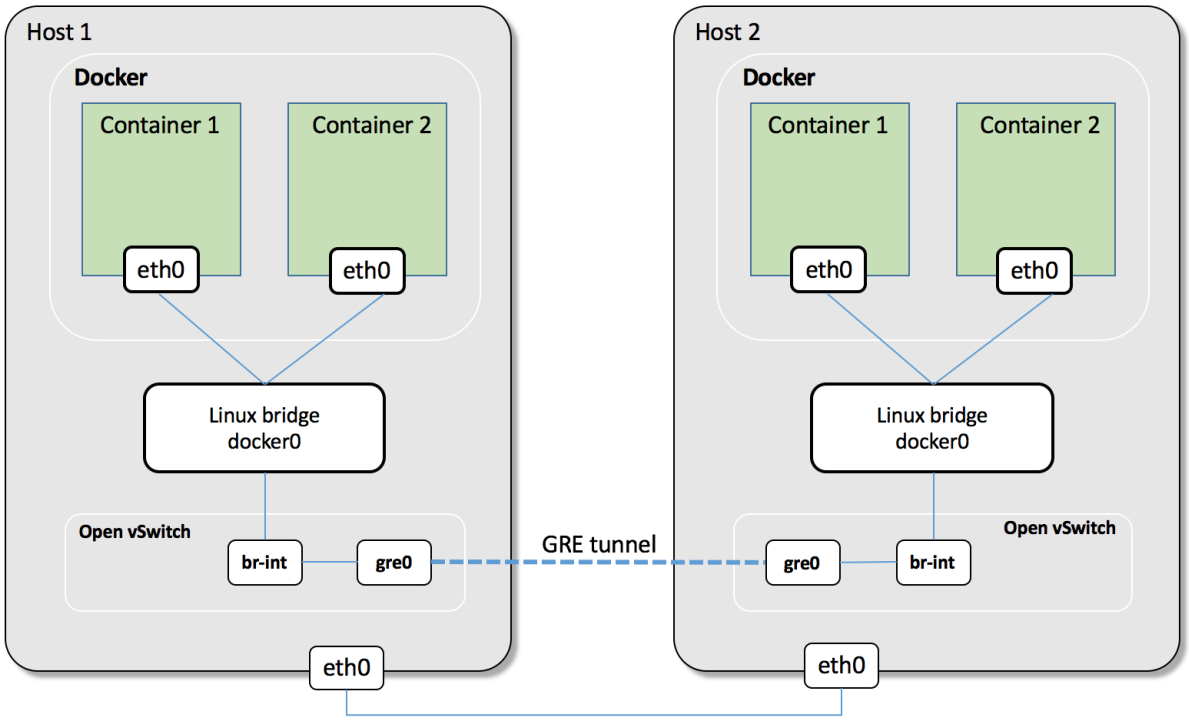
This lab will show multi-host network, let's see how containers in different hosts can communicate with each other.

There are at least two ways connect containers with open vSwitch.

- connect default docker0 with ovs bridge
- connect container with ovs bridge directly through veth pair.

We will chose the first way, becuase it's easier. For the second way, if don't use the default docker0 bridge, we will need to do more work toconnect containers with ovs, such as create network namespace and veth pair manully, attach veth to container, resolve ip address management, NAT, etc.

Topology



containers connect with docker0 bridge

Start a container on host 2

```
ubuntu@docker-node2:~$ docker run -d --name container1 centos:7 /bin/bash -c "while␣
↪true; do sleep 3600; done"
98ddd33b16ed5206615aa6bd8e930b359a877794dffe921ee20f0c4b000a440a
ubuntu@docker-node2:~$
ubuntu@docker-node2:~$ docker inspect --format '{{.NetworkSettings.IPAddress}}'␣
↪container1
172.17.0.2
```

Start two containers on host 1

```
ubuntu@docker-node1:~$ docker run -d --name container1 centos:7 /bin/bash -c "while␣
↪true; do sleep 3600; done"
31109d970148d710c3465af86ec3fb14229c1660640ae56c5b18435286168824
ubuntu@docker-node1:~$ docker run -d --name container2 centos:7 /bin/bash -c "while␣
↪true; do sleep 3600; done"
fdf1cebdd9a5264e18337ea3569a081c59e5e27e2219184557e44921faa63822
ubuntu@docker-node1:~$
ubuntu@docker-node1:~$ docker inspect --format '{{.NetworkSettings.IPAddress}}'␣
↪container1
172.17.0.2
ubuntu@docker-node1:~$ docker inspect --format '{{.NetworkSettings.IPAddress}}'␣
↪container2
172.17.0.3
ubuntu@docker-node1:~$
```

Stop container 1 on host 1, because it has them same IP address as container 1 on host 2

```
ubuntu@docker-node1:~$ docker stop container1
container1
```

container 2 on host 1 can not access container 1 on host 2

```
ubuntu@docker-node1:~$ docker exec -it container2 bash
[root@fdf1cebdd9a5 /]# ping 172.17.0.2
PING 172.17.0.2 (172.17.0.2) 56(84) bytes of data.
^C
--- 172.17.0.2 ping statistics ---
18 packets transmitted, 0 received, 100% packet loss, time 17033ms

[root@fdf1cebdd9a5 /]#
```

Configure OVS

Install OVS:

```
$ sudo apt-get install -y openvswitch-switch openvswitch-common
```

Host 1

Create a ovs bridge and a veth pair

```
ubuntu@docker-node1:~$ sudo ovs-vsctl add-br br-int
ubuntu@docker-node1:~$ sudo ovs-vsctl show
9e5ebe46-02bf-4899-badd-7aa10245afcb
    Bridge br-int
        Port br-int
            Interface br-int
                type: internal
    ovs_version: "2.5.0"
ubuntu@docker-node1:~$
ubuntu@docker-node1:~$ sudo ip link add veth0 type veth peer name veth1
```

Connect veth pair with `dockre0` and ovs bridge `br-int`, set them up.

```
ubuntu@docker-node1:~$ sudo ovs-vsctl add-port br-int veth1
ubuntu@docker-node1:~$ sudo brctl addif docker0 veth0
ubuntu@docker-node1:~$ sudo ip link set veth1 up
ubuntu@docker-node1:~$ sudo ip link set veth0 up
ubuntu@docker-node1:~$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT␣
↪group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode␣
↪DEFAULT group default qlen 1000
    link/ether 02:57:5b:96:48:35 brd ff:ff:ff:ff:ff:ff
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode␣
↪DEFAULT group default qlen 1000
    link/ether 08:00:27:c3:54:4f brd ff:ff:ff:ff:ff:ff
4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode␣
↪DEFAULT group default
```

```
    link/ether 02:42:23:8f:ab:da brd ff:ff:ff:ff:ff:ff
9: ovs-system: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT␣
↪group default qlen 1
    link/ether 46:eb:b5:81:eb:31 brd ff:ff:ff:ff:ff:ff
10: br-int: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group␣
↪default qlen 1
    link/ether 42:a0:96:7b:e9:42 brd ff:ff:ff:ff:ff:ff
11: veth1@veth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master ovs-
↪system state UP mode DEFAULT group default qlen 1000
    link/ether 2a:8a:93:9d:b2:b4 brd ff:ff:ff:ff:ff:ff
12: veth0@veth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master␣
↪docker0 state UP mode DEFAULT group default qlen 1000
    link/ether ae:16:a0:03:12:4e brd ff:ff:ff:ff:ff:ff
```

Host 2

Almost do the same thing on host 2.

```
ubuntu@docker-node2:~$ ovs-vsctl add-br br-int
ubuntu@docker-node2:~$ sudo ip link add veth0 type veth peer name veth1
ubuntu@docker-node2:~$ sudo ovs-vsctl add-port br-int veth1
ubuntu@docker-node2:~$ sudo brctl addif docker0 veth0
ubuntu@docker-node2:~$ sudo ip link set veth1 up
ubuntu@docker-node2:~$ sudo ip link set veth0 up
```

GRE tunnel between host 1 and host 2

on host 1

```
ubuntu@docker-node1:~$ sudo ovs-vsctl add-port br-int gre0 -- \
set interface gre0 type=gre options:remote_ip=192.168.205.11
```

on host 2

```
ubuntu@docker-node2:~$ sudo ovs-vsctl add-port br-int gre0 -- \
set interface gre0 type=gre options:remote_ip=192.168.205.10
```

The connection between ovs bridge and docker0 bridge

```
ubuntu@docker-node1:~$ sudo ovs-vsctl show
9e5ebe46-02bf-4899-badd-7aa10245afcb
    Bridge br-int
        Port "veth1"
            Interface "veth1"
        Port br-int
            Interface br-int
                type: internal
        Port "gre0"
            Interface "gre0"
                type: gre
                options: {remote_ip="192.168.205.11"}
    ovs_version: "2.5.0"
ubuntu@docker-node1:~$ brctl show
bridge name bridge id               STP enabled     interfaces
```

```
docker0             8000.0242238fabda       no              veth0
                                                            vethd5c0abe
ubuntu@docker-node1:~$
```

Check GRE tunnel connection

in container1 on host 2 ping container 2 on host 1

```
ubuntu@docker-node2:~$ docker exec -it container1 bash
[root@98ddd33b16ed /]# ping 172.17.0.3
PING 172.17.0.3 (172.17.0.3) 56(84) bytes of data.
64 bytes from 172.17.0.3: icmp_seq=1 ttl=64 time=1.19 ms
64 bytes from 172.17.0.3: icmp_seq=2 ttl=64 time=0.624 ms
64 bytes from 172.17.0.3: icmp_seq=3 ttl=64 time=0.571 ms
^C
--- 172.17.0.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 0.571/0.797/1.198/0.285 ms
[root@98ddd33b16ed /]#
```

At the same time, start `tcpdump` on host 1 and capture packges on the GRE source interface.

```
ubuntu@docker-node1:~$ sudo tcpdump -n -i enp0s8 proto gre
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on enp0s8, link-type EN10MB (Ethernet), capture size 262144 bytes
14:12:17.966149 IP 192.168.205.11 > 192.168.205.10: GREv0, length 102: IP 172.17.0.2 >
 ↪ 172.17.0.3: ICMP echo request, id 23, seq 1, length 64
14:12:17.966843 IP 192.168.205.10 > 192.168.205.11: GREv0, length 102: IP 172.17.0.3 >
 ↪ 172.17.0.2: ICMP echo reply, id 23, seq 1, length 64
14:12:18.967513 IP 192.168.205.11 > 192.168.205.10: GREv0, length 102: IP 172.17.0.2 >
 ↪ 172.17.0.3: ICMP echo request, id 23, seq 2, length 64
14:12:18.967658 IP 192.168.205.10 > 192.168.205.11: GREv0, length 102: IP 172.17.0.3 >
 ↪ 172.17.0.2: ICMP echo reply, id 23, seq 2, length 64
14:12:19.968683 IP 192.168.205.11 > 192.168.205.10: GREv0, length 102: IP 172.17.0.2 >
 ↪ 172.17.0.3: ICMP echo request, id 23, seq 3, length 64
14:12:19.968814 IP 192.168.205.10 > 192.168.205.11: GREv0, length 102: IP 172.17.0.3 >
 ↪ 172.17.0.2: ICMP echo reply, id 23, seq 3, length 64
14:12:22.982906 ARP, Request who-has 192.168.205.11 tell 192.168.205.10, length 28
14:12:22.983262 ARP, Reply 192.168.205.11 is-at 08:00:27:b8:22:30 (oui Unknown),␣
 ↪length 46
```

Improvement

There are some improvements can be done for this lab:

- Create a new docket network instead of using the default docker0 bridge
- docker bridge on host 1 and host 1 have different network ip range for containers

1.2.15 Multi-Host Networking Overlay with Calico

1.2.16 Multi-Host Networking Overlay with Flannel

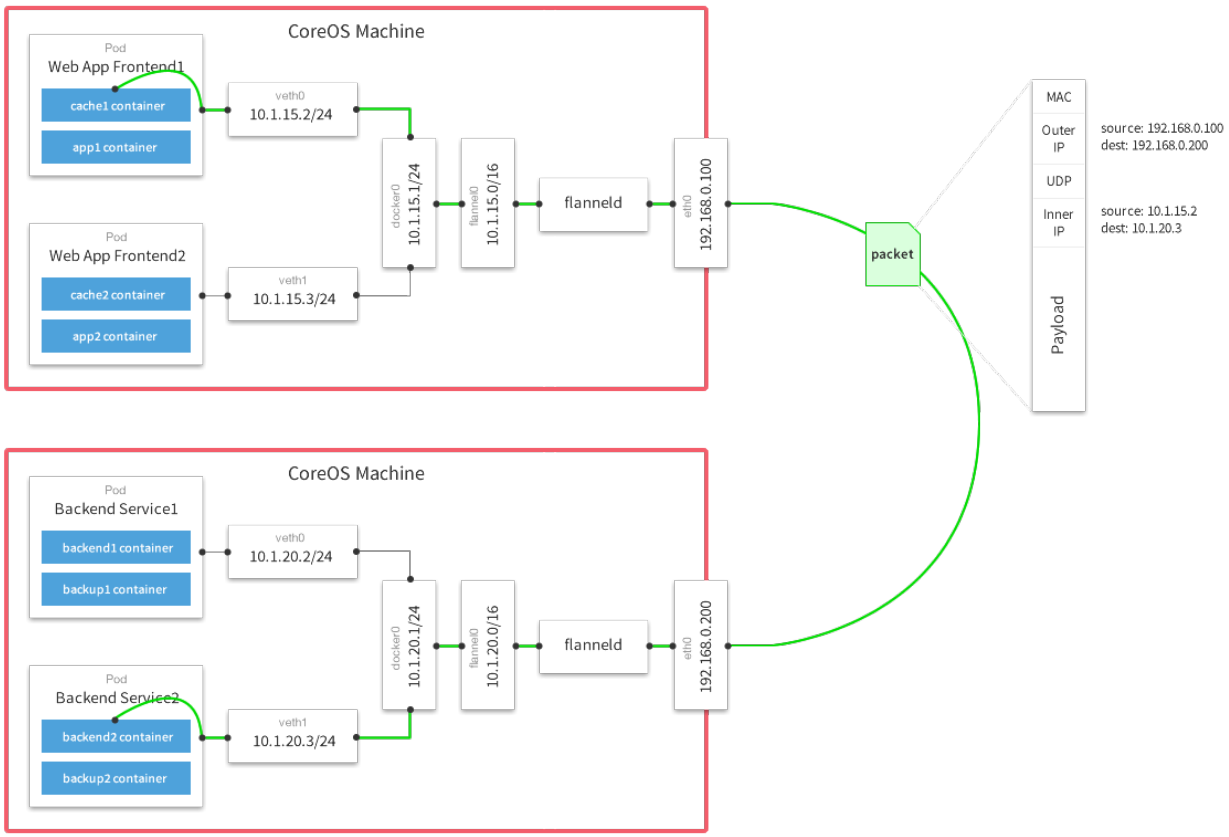
In the Lab *Multi-Host Overlay Networking with Etcd*, we use `etcd` as management plane and docker build-in overlay network as data plane to show how containers in different host connect with each other.

This time we will use `flannel` to do almost the same thing.

`Flannel` is created by CoreOS and it is a network fabric for containers, designed for Kubernetes.

Theory of Operation [1]

flannel runs an agent, `flanneld`, on each host and is responsible for allocating a subnet lease out of a preconfigured address space. flannel uses `etcd` to store the network configuration, allocated subnets, and auxiliary data (such as host's IP). The forwarding of packets is achieved using one of several strategies that are known as backends. The simplest backend is udp and uses a TUN device to encapsulate every IP fragment in a UDP packet, forming an overlay network. The following diagram demonstrates the path a packet takes as it traverses the overlay network:



Lab Environment

Follow *Lab Environment Quick Setup* and setup two nodes of docker host.

[1] https://github.com/coreos/flannel

Hostname	IP	Docker version
docker-node1	192.168.205.10	1.12.1
docker-node2	192.168.205.11	1.12.1

Etcd Cluster Setup

Just follow *Multi-Host Overlay Networking with Etcd* to setup two nodes etcd cluster.

When setup is ready, you should see the etcd cluster status as:

```
ubuntu@docker-node2:~/etcd-v3.0.12-linux-amd64$ ./etcdctl cluster-health
member 21eca106efe4caee is healthy: got healthy result from http://192.168.205.10:2379
member 8614974c83d1cc6d is healthy: got healthy result from http://192.168.205.11:2379
cluster is healthy
```

Install & Configure & Run flannel

Download flannel both on node1 and node2

```
$ wget https://github.com/coreos/flannel/releases/download/v0.6.2/flanneld-amd64 -O␣
↪flanneld && chmod 755 flanneld
```

flannel will read the configuration from etcd `/coreos.com/network/config` by default. We will use `etcdctl` to set our configuration to etcd cluster, the configuration is JSON format like that:

```
ubuntu@docker-node1:~$ cat > flannel-network-config.json
{
    "Network": "10.0.0.0/8",
    "SubnetLen": 20,
    "SubnetMin": "10.10.0.0",
    "SubnetMax": "10.99.0.0",
    "Backend": {
        "Type": "vxlan",
        "VNI": 100,
        "Port": 8472
    }
}
EOF
```

For the configuration keys meaning, please go to https://github.com/coreos/flannel for more information. Set the configuration on host1:

```
ubuntu@docker-node1:~$ cd etcd-v3.0.12-linux-amd64/
ubuntu@docker-node1:~/etcd-v3.0.12-linux-amd64$ ./etcdctl set /coreos.com/network/
↪config < ../flannel-network-config.json
{
    "Network": "10.0.0.0/8",
    "SubnetLen": 20,
    "SubnetMin": "10.10.0.0",
    "SubnetMax": "10.99.0.0",
    "Backend": {
        "Type": "vxlan",
        "VNI": 100,
        "Port": 8472
    }
}
```

Check the configuration on host2:

```
ubuntu@docker-node2:~/etcd-v3.0.12-linux-amd64$ ./etcdctl get /coreos.com/network/
↪config | jq .
{
  "Network": "10.0.0.0/8",
  "SubnetLen": 20,
  "SubnetMin": "10.10.0.0",
  "SubnetMax": "10.99.0.0",
  "Backend": {
    "Type": "vxlan",
    "VNI": 100,
    "Port": 8472
  }
}
```

Start flannel on host1:

```
ubuntu@docker-node1:~$ cd
ubuntu@docker-node1:~$ nohup sudo ./flanneld -iface=192.168.205.10 &
```

After that a new interface `flannel.100` will be list on the host:

```
flannel.100 Link encap:Ethernet  HWaddr 82:53:2e:6a:a9:43
          inet addr:10.15.64.0  Bcast:0.0.0.0  Mask:255.0.0.0
          inet6 addr: fe80::8053:2eff:fe6a:a943/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:8 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Before we start flannel on host2, we can check etcd configuration on host2:

```
ubuntu@docker-node2:~/etcd-v3.0.12-linux-amd64$ ./etcdctl ls /coreos.com/network/
↪subnets
/coreos.com/network/subnets/10.15.64.0-20
ubuntu@docker-node2:~/etcd-v3.0.12-linux-amd64$ ./etcdctl get /coreos.com/network/
↪subnets/10.15.64.0-20 | jq .
{
  "PublicIP": "192.168.205.10",
  "BackendType": "vxlan",
  "BackendData": {
    "VtepMAC": "82:53:2e:6a:a9:43"
  }
}
```

This is the flannel backend information on host1.

Start flannel on host2

```
ubuntu@docker-node2:~$ nohup sudo ./flanneld -iface=192.168.205.11 &
```

Check the etcd configuration

```
ubuntu@docker-node2:~/etcd-v3.0.12-linux-amd64$ ./etcdctl ls /coreos.com/network/
↪subnets/
/coreos.com/network/subnets/10.15.64.0-20
/coreos.com/network/subnets/10.13.48.0-20
```

```
ubuntu@docker-node2:~/etcd-v3.0.12-linux-amd64$ ./etcdctl get /coreos.com/network/
↪subnets/10.13.48.0-20
{"PublicIP":"192.168.205.11","BackendType":"vxlan","BackendData":{"VtepMAC":"9e:e7:65:
↪f3:9d:31"}}
```

This also has a new interface created by flannel `flannel.100`

Restart docker daemon with flannel network

Restart docker daemon with Flannel network configuration, execute commands as follows on node1 and node2:

```
ubuntu@docker-node1:~$ sudo service docker stop
ubuntu@docker-node1:~$ sudo docker ps
Cannot connect to the Docker daemon. Is the docker daemon running on this host?
ubuntu@docker-node1:~$ source /run/flannel/subnet.env
ubuntu@docker-node1:~$ sudo ifconfig docker0 ${FLANNEL_SUBNET}
ubuntu@docker-node1:~$ sudo docker daemon --bip=${FLANNEL_SUBNET} --mtu=${FLANNEL_MTU}
↪ &
```

After restarting, the docker daemon will bind docker0 which has a new address. We can check the new configuration with `sudo docker network inspect bridge`.

Adjust iptables

Starting from Docker 1.13 default iptables policy for FORWARDING is DROP, so to make sure that containers will receive traffic from another hosts we need to adjust it:

On host1:

```
ubuntu@docker-node1:~$ sudo iptables -P FORWARD ACCEPT
```

On host2:

```
ubuntu@docker-node2:~$ sudo iptables -P FORWARD ACCEPT
```

Start Containers

On host1:

```
ubuntu@docker-node1:~$ sudo docker run -d --name test1  busybox sh -c "while true; do␣
↪sleep 3600; done"
ubuntu@docker-node1:~$ sudo docker exec test1 ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:0A:0F:40:02
          inet addr:10.15.64.2  Bcast:0.0.0.0  Mask:255.255.240.0
          inet6 addr: fe80::42:aff:fe0f:4002/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
          RX packets:16 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1296 (1.2 KiB)  TX bytes:648 (648.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
```

```
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Oh host2:

```
ubuntu@docker-node2:~$ sudo docker run -d --name test2  busybox sh -c "while true; do␣
↪sleep 3600; done"
ubuntu@docker-node2:~$ sudo docker exec test2 ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:0A:0D:30:02
          inet addr:10.13.48.2  Bcast:0.0.0.0  Mask:255.255.240.0
          inet6 addr: fe80::42:aff:fe0d:3002/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:648 (648.0 B)  TX bytes:648 (648.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Container test1 on host1 ping container test2 on host2

```
ubuntu@docker-node1:~$ sudo docker exec test1 ping google.com
PING google.com (74.125.68.102): 56 data bytes
64 bytes from 74.125.68.102: seq=0 ttl=61 time=123.295 ms
64 bytes from 74.125.68.102: seq=1 ttl=61 time=127.646 ms
ubuntu@docker-node1:~$ sudo docker exec test1 ping 10.13.48.2
PING 10.13.48.2 (10.13.48.2): 56 data bytes
64 bytes from 10.13.48.2: seq=0 ttl=62 time=1.347 ms
64 bytes from 10.13.48.2: seq=1 ttl=62 time=0.430 ms
```

Through `sudo tcpdump -i enp0s8 -n not port 2380` we can confirm the vxlan tunnel.

```
05:54:43.824182 IP 192.168.205.10.36214 > 192.168.205.11.8472: OTV, flags [I] (0x08),␣
↪overlay 0, instance 100
IP 10.15.64.0 > 10.13.48.2: ICMP echo request, id 9728, seq 462, length 64
05:54:43.880055 IP 192.168.205.10.36214 > 192.168.205.11.8472: OTV, flags [I] (0x08),␣
↪overlay 0, instance 100
IP 10.15.64.0 > 10.13.48.2: ICMP echo request, id 11264, seq 245, length 64
05:54:44.179703 IP 192.168.205.10.36214 > 192.168.205.11.8472: OTV, flags [I] (0x08),␣
↪overlay 0, instance 100
IP 10.15.64.0 > 10.13.48.2: ICMP echo request, id 12288, seq 206, length 64
```

Performance test [2]

[2] http://chunqi.li/2015/10/10/Flannel-for-Docker-Overlay-Network/

Reference

1.2.17 Multi-host networking with Contiv

http://contiv.github.io/documents/tutorials/container-101.html

1.2.18 Docker Compose Networking Deep Dive

Note: We suggest that you should complete the lab *Bridge Networking Deep Dive* firstly before going to this lab.

This lab will use `example-voting-app` as the demo application run by docker-compose, you can find the source code of the project in https://github.com/DaoCloud/example-voting-app

Using Compose is basically a three-step process. [1]

1. Define your app's environment with a Dockerfile so it can be reproduced anywhere.
2. Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment.
3. Lastly, run docker-compose up and Compose will start and run your entire app.

For `example-voting-app`, we already have `Dockerfile` and `docker-compose.yml`, what need to do is `docker-compose up`.

Install Docker Compose

There are many ways to install docker compose [2].

In our one node docker engine lab environment *Lab Environment Quick Setup* we install docker compose as the following way in one docker host.

```
ubuntu@docker-node1:~$ sudo curl -L "https://github.com/docker/compose/releases/
↪download/1.9.0/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-
↪compose
ubuntu@docker-node1:~$ sudo chmod +x /usr/local/bin/docker-compose
ubuntu@docker-node1:~$ docker-compose -v
docker-compose version 1.9.0, build 2585387
```

Start APP

Clone `example-voting-app` repository to docker host, it defined five containers: `voting-app`, `result-app`, `worker`, `redis`, `db`. and two networks: `front-tier`, `back-tier` through `docker-compose.yml`.

```
version: "2"

services:
  voting-app:
    build: ./voting-app/.
    volumes:
     - ./voting-app:/app
```

[1] https://docs.docker.com/compose/overview/
[2] https://docs.docker.com/compose/install/

```
    ports:
      - "5000:80"
    links:
      - redis
    networks:
      - front-tier
      - back-tier

  result-app:
    build: ./result-app/.
    volumes:
      - ./result-app:/app
    ports:
      - "5001:80"
    links:
      - db
    networks:
      - front-tier
      - back-tier

  worker:
    build: ./worker
    links:
      - db
      - redis
    networks:
      - back-tier

  redis:
    image: redis
    ports: ["6379"]
    networks:
      - back-tier

  db:
    image: postgres:9.4
    volumes:
      - "db-data:/var/lib/postgresql/data"
    networks:
      - back-tier

volumes:
  db-data:

networks:
  front-tier:
  back-tier:
```

Then run `docker-compose build` to build required docker images. This will take some time.

```
ubuntu@docker-node1:~$ git clone https://github.com/DaoCloud/example-voting-app
ubuntu@docker-node1:~$ cd example-voting-app/
ubuntu@docker-node1:~/example-voting-app$ sudo docker-compose build


ubuntu@docker-node1:~/example-voting-app$ sudo docker-compose up
Creating network "examplevotingapp_front-tier" with the default driver
Creating network "examplevotingapp_back-tier" with the default driver
```

```
Creating volume "examplevotingapp_db-data" with default driver
....
Creating examplevotingapp_db_1
Creating examplevotingapp_redis_1
Creating examplevotingapp_voting-app_1
Creating examplevotingapp_result-app_1
Creating examplevotingapp_worker_1
Attaching to examplevotingapp_redis_1, examplevotingapp_db_1, examplevotingapp_result-
↪app_1, examplevotingapp_voting-app_1, examplevotingapp_worker_1
...
```

There will be five containers, two bridge networks and seven veth interfaces created.

```
ubuntu@docker-node1:~/example-voting-app$ sudo docker ps
CONTAINER ID        IMAGE                            COMMAND                  CREATED␣
↪         STATUS              PORTS                      NAMES
c9c4e7fe7b6c        examplevotingapp_worker          "/usr/lib/jvm/java-7-"   About an␣
↪hour ago   Up 5 seconds                                  examplevotingapp_worker_1
4213167049aa        examplevotingapp_result-app      "node server.js"         About an␣
↪hour ago   Up 4 seconds        0.0.0.0:5001->80/tcp      examplevotingapp_result-
↪app_1
8711d687bda9        examplevotingapp_voting-app      "python app.py"          About an␣
↪hour ago   Up 5 seconds        0.0.0.0:5000->80/tcp      examplevotingapp_voting-
↪app_1
b7eda251865d        redis                            "docker-entrypoint.sh"   About an␣
↪hour ago   Up 5 seconds        0.0.0.0:32770->6379/tcp   examplevotingapp_redis_1
7d6dbb98ce40        postgres:9.4                     "/docker-entrypoint.s"   About an␣
↪hour ago   Up 5 seconds        5432/tcp                  examplevotingapp_db_1
ubuntu@docker-node1:~/example-voting-app$ sudo docker network ls
NETWORK ID          NAME                             DRIVER              SCOPE
3b5cfe4aafa1        bridge                           bridge              local
69a019d00603        examplevotingapp_back-tier       bridge              local
6ddb07377c35        examplevotingapp_front-tier      bridge              local
b1670e00e2a3        host                             host                local
6006af29f010        none                             null                local
ubuntu@docker-node1:~/example-voting-app$ brctl show
bridge name   bridge id               STP enabled     interfaces
br-69a019d00603               8000.0242c780244f       no              veth2eccb94
                                                      veth374be12
                                                      veth57f50a8
                                                      veth8418ed3
                                                      veth91d724d
br-6ddb07377c35               8000.02421dac7490       no              veth156c0a9
                                                      vethaba6401
```

Through `docker network inspect`, we can know which container connnect with the bridge.

There are two containers connect with docker network `examplevotingapp_front-tier`.

```
ubuntu@docker-node1:~/example-voting-app$ sudo docker network inspect␣
↪examplevotingapp_front-tier
[
    {
        "Name": "examplevotingapp_front-tier",
        "Id": "6ddb07377c354bcf68542592a8c6eb34d334ce8515e64832b3c7bf2af56274ca",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
```

```
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.18.0.0/16",
                    "Gateway": "172.18.0.1/16"
                }
            ]
        },
        "Internal": false,
        "Containers": {
            "4213167049aa7b2cc1b3096333706f2ef0428e78b2847a7c5ddc755f5332505c": {
                "Name": "examplevotingapp_result-app_1",
                "EndpointID":
↪"00c7e1101227ece1535385e8d6fe9210dfcdc3c58d71cedb4e9fad6c949120e3",
                "MacAddress": "02:42:ac:12:00:03",
                "IPv4Address": "172.18.0.3/16",
                "IPv6Address": ""
            },
            "8711d687bda94069ed7d5a7677ca4c7953d384f1ebf83c3bd75ac51b1606ed2f": {
                "Name": "examplevotingapp_voting-app_1",
                "EndpointID":
↪"ffc9905cbfd5332b9ef333bcc7578415977a0044c2ec2055d6760c419513ae5f",
                "MacAddress": "02:42:ac:12:00:02",
                "IPv4Address": "172.18.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {}
    }
]
```

There are five containers connect with docker network `examplevotingapp_back-tier`.

```
ubuntu@docker-node1:~/example-voting-app$ sudo docker network inspect␣
↪examplevotingapp_back-tier
[
    {
        "Name": "examplevotingapp_back-tier",
        "Id": "69a019d00603ca3a06a30ac99fc0a2700dd8cc14ba8b8368de4fe0c26ad4c69d",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.19.0.0/16",
                    "Gateway": "172.19.0.1/16"
                }
            ]
        },
        "Internal": false,
        "Containers": {
            "4213167049aa7b2cc1b3096333706f2ef0428e78b2847a7c5ddc755f5332505c": {
```

```
                "Name": "examplevotingapp_result-app_1",
                "EndpointID":
↪"cb531eb6deb08346d1dbcfa65ea67d43d4c2f244f002b195fc4dadd2adb0b47d",
                "MacAddress": "02:42:ac:13:00:06",
                "IPv4Address": "172.19.0.6/16",
                "IPv6Address": ""
            },
            "7d6dbb98ce408c1837f42fdf743e365cc9b0ee2b7dffd108d97e81b172d43114": {
                "Name": "examplevotingapp_db_1",
                "EndpointID":
↪"67007a454f320d336c13e30e028cd8e85537400b70a880eabdd1f0ed743b7a6a",
                "MacAddress": "02:42:ac:13:00:03",
                "IPv4Address": "172.19.0.3/16",
                "IPv6Address": ""
            },
            "8711d687bda94069ed7d5a7677ca4c7953d384f1ebf83c3bd75ac51b1606ed2f": {
                "Name": "examplevotingapp_voting-app_1",
                "EndpointID":
↪"d414b06b9368d1719a05d527500a06fc714a4efae187df32c1476385ee03ae67",
                "MacAddress": "02:42:ac:13:00:05",
                "IPv4Address": "172.19.0.5/16",
                "IPv6Address": ""
            },
            "b7eda251865d824de90ebe0dfefa3e4aab924d5030ccfb21a55e79f910ff857a": {
                "Name": "examplevotingapp_redis_1",
                "EndpointID":
↪"9acc267d3e6b41da6fe3db040cff964c91037df215a0f2be2155b94be3bb87d0",
                "MacAddress": "02:42:ac:13:00:02",
                "IPv4Address": "172.19.0.2/16",
                "IPv6Address": ""
            },
            "c9c4e7fe7b6c1508f9d9d3a05e8a4e66aa1265f2a5c3d33f363343cd37184e6f": {
                "Name": "examplevotingapp_worker_1",
                "EndpointID":
↪"557e978eaef18a64f24d400727d396431d74cd7e8735f060396e3226f31ab97b",
                "MacAddress": "02:42:ac:13:00:04",
                "IPv4Address": "172.19.0.4/16",
                "IPv6Address": ""
            }
        },
        "Options": {},
        "Labels": {}
    }
]
```

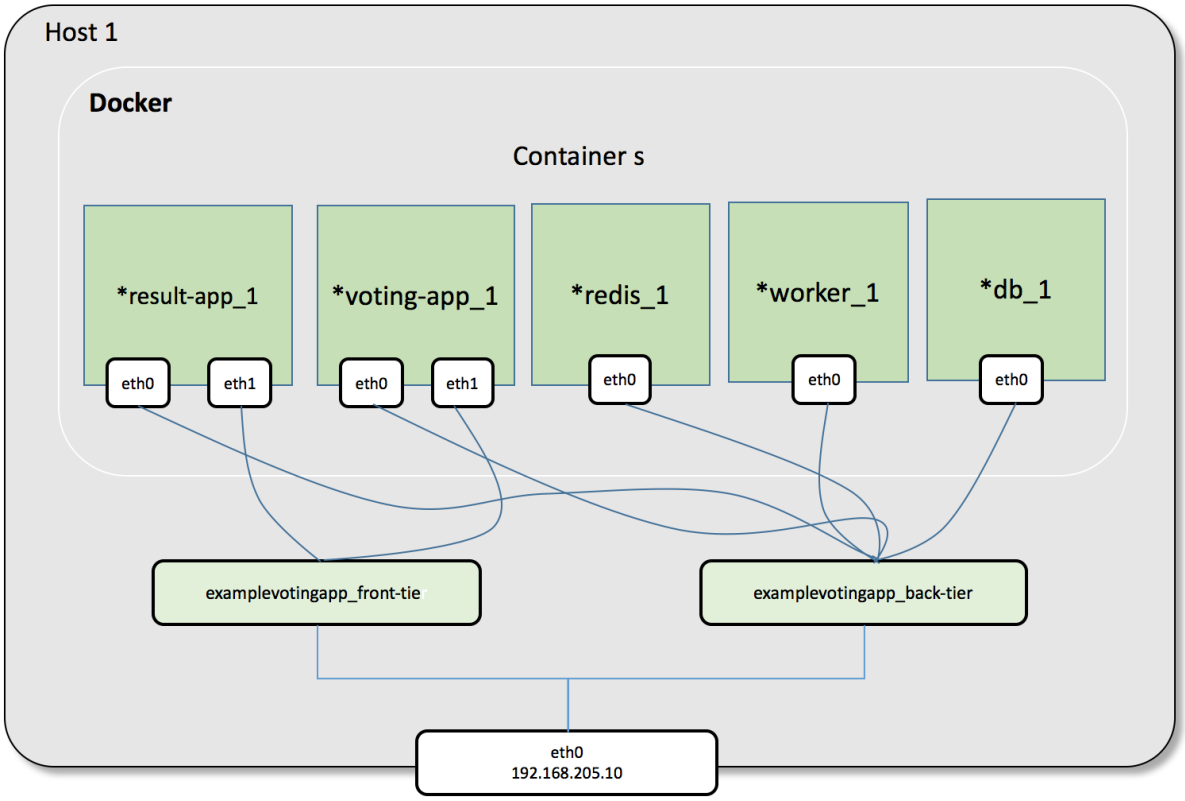
Container information summary:

Container Name	IP Address
examplevotingapp_result-app_1	172.19.0.6/16, 172.18.0.3/16
examplevotingapp_voting-app_1	172.19.0.3/16, 172.18.0.2/16
examplevotingapp_redis_1	172.19.0.2/16
examplevotingapp_worker_1	172.19.0.4/16
examplevotingapp_db_1	172.19.0.3/16

Docker network information summary:

Docker Network Name	Gateway	Subnet	Containers
examplevotingapp_front-tier	172.18.0.1	172.18.0.0/16	examplevotingapp_result-app_1, examplevotingapp_voting-app_1
examplevotingapp_back-tier	172.19.0.1	172.19.0.0/16	examplevotingapp_result-app_1, examplevotingapp_voting-app_1, examplevotingapp_db_1, examplevotingapp_redis_1, examplevotingapp_worker_1

Network Topology



For bridge network connection details, please reference lab *Bridge Networking Deep Dive*

Reference

1.2.19 Docker Compose Load Blancing and Scaling

Please finish *Docker Compose Networking Deep Dive* firstly.

In this lab, we will create a web service, try to scale this service, and add load blancer.

`docker-compose.yml` file, we just use two images.

```
$ more docker-compose.yml
web:
  image: 'jwilder/whoami'
lb:
```

```
  image: 'dockercloud/haproxy:latest'
  links:
    - web
  ports:
    - '80:80'
```

Start and check the service.

```
$ docker-compose up
$ docker-compose up -d
Creating ubuntu_web_1
Creating ubuntu_lb_1
$ docker-compose ps
    Name                   Command               State                    Ports
-------------------------------------------------------------------------------------------
 ↪-------
ubuntu_lb_1    /sbin/tini -- dockercloud- ...   Up      1936/tcp, 443/tcp, 0.0.0.0:80-
 ↪>80/tcp
ubuntu_web_1   /bin/sh -c php-fpm -d vari ...   Up      80/tcp
```

Open the browser and check the hostname.

Scale the web service to 2 and check:

```
$ docker-compose scale web=3
Creating and starting ubuntu_web_2 ... done
Creating and starting ubuntu_web_3 ... done
ubuntu@aws-swarm-manager:~$ docker-compose ps
    Name                   Command               State                    Ports
-------------------------------------------------------------------------------------------
 ↪-------
ubuntu_lb_1    /sbin/tini -- dockercloud- ...   Up      1936/tcp, 443/tcp, 0.0.0.0:80-
 ↪>80/tcp
ubuntu_web_1   /bin/sh -c php-fpm -d vari ...   Up      80/tcp
ubuntu_web_2   /bin/sh -c php-fpm -d vari ...   Up      80/tcp
ubuntu_web_3   /bin/sh -c php-fpm -d vari ...   Up      80/tcp
```

1.2.20 Swarm Mode: Create a Docker Swarm Cluster

Docker swarm mode requires docker engine 1.12 or higher. This lab will need two docker engine host created by docker machine.

Prepare Environment

Create two docker host machines.

```
  ~ docker-machine ls
NAME             ACTIVE   DRIVER       STATE     URL                         SWARM   
 ↪DOCKER    ERRORS
swarm-manager    -        virtualbox   Running   tcp://192.168.99.100:2376           
 ↪v1.12.4
swarm-worker1    -        virtualbox   Running   tcp://192.168.99.101:2376           
 ↪v1.12.4
  ~ docker-machine ip swarm-manager
192.168.99.100
```

```
  ~ docker-machine ip swarm-worker1
192.168.99.101
  ~
```

Create a Swarm Manage node

SSH to swarm-manager host and init a manager node.

```
  ~ docker-machine ssh swarm-manager
docker@swarm-manager:~$ docker swarm init --advertise-addr 192.168.99.100
Swarm initialized: current node (7f2gi8xoz6prs2gi53nqa4wu8) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join \
    --token SWMTKN-1-58lrmtavqlt9v1ejujsfh5o9hf3p804xtn5qhnsriqw4an2vhd-
↪8x1q7q4jpvs1govwmjhnhffo7 \
    192.168.99.100:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the␣
↪instructions.

docker@swarm-manager:~$
```

From command `docker info` we can get the current information about this swarm cluster.

Add one Docker Node to the Swarm cluster

Just run the command generated by `swarm init` last step in the other docker machine host. Please make sure the `swarm-worker1` host can access 192.168.99.100:2377

```
  ~ docker-machine ssh swarm-worker1
docker@swarm-worker1:~$ docker swarm join \
        --token SWMTKN-1-58lrmtavqlt9v1ejujsfh5o9hf3p804xtn5qhnsriqw4an2vhd-
↪8x1q7q4jpvs1govwmjhnhffo7 \
        192.168.99.100:2377
This node joined a swarm as a worker.
docker@swarm-worker1:~$
```

We can check the cluster status on manager node:

```
  ~ docker-machine ssh swarm-manager
Boot2Docker version 1.12.4, build HEAD : d0b8fd8 - Tue Dec 13 18:21:26 UTC 2016
Docker version 1.12.4, build 1564f02
docker@swarm-manager:~$ docker node ls
ID                           HOSTNAME       STATUS  AVAILABILITY  MANAGER STATUS
7f2gi8xoz6prs2gi53nqa4wu8 *  swarm-manager  Ready   Active        Leader
9mm8t4l5stcudn5tx1fweht1d    swarm-worker1  Ready   Active
docker@swarm-manager:~$
```

And there are two networks automatically created on these two hosts:

```
docker@swarm-manager:~$ sudo docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
f773d9bee59f        bridge              bridge              local
```

```
bcc7996ba96b        docker_gwbridge     bridge              local
a2d7040abdd0        host                host                local
01y2wr8jucgf        ingress             overlay             swarm
8fde4990cff2        none                null                local
docker@swarm-manager:~$
docker@swarm-worker1:~$ sudo docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
470f8e1db857        bridge              bridge              local
18bcb76c26b0        docker_gwbridge     bridge              local
1e347b54188e        host                host                local
01y2wr8jucgf        ingress             overlay             swarm
9ba27b95c9ad        none                null                local
docker@swarm-worker1:~$
```

The first is `docker_gwbridge` and the second is `ingress`, one is bridge network, and the other is overlay network.

1.2.21 Docker Swarm: Create and Scale a Service

In this lab we will create a new docker swarm cluster: one manger node and three worker nodes, then create a service and try to scale it.

Create a Swarm Cluster

Based on the lab *Swarm Mode: Create a Docker Swarm Cluster*, create four docker machines and init a swarm cluster.

```
  ~ docker-machine ls
NAME             ACTIVE   DRIVER       STATE     URL                         SWARM
↪DOCKER    ERRORS
swarm-manager    -        virtualbox   Running   tcp://192.168.99.103:2376
↪v1.12.5
swarm-worker1    -        virtualbox   Running   tcp://192.168.99.104:2376
↪v1.12.5
swarm-worker2    -        virtualbox   Running   tcp://192.168.99.105:2376
↪v1.12.5
swarm-worker3    -        virtualbox   Running   tcp://192.168.99.106:2376
↪v1.12.5
  ~

docker@swarm-manager:~$ docker node ls
ID                           HOSTNAME       STATUS  AVAILABILITY  MANAGER STATUS
0skz2g68hb76efq4xknhwsjt9    swarm-worker2  Ready   Active
2q015a61bl879o6adtlb7kxkl    swarm-worker3  Ready   Active
2sph1ezrnr5q9vy0683ah3b90 *  swarm-manager  Ready   Active        Leader
59rzjt0kqbcgw4cz7zsfflk8z    swarm-worker1  Ready   Active
docker@swarm-manager:~$
```

Create a Service

Use `docker service create` command on manager node to create a service

```
docker@swarm-manager:~$ docker service create --name myapp --publish 80:80/tcp nginx
7bb8pgwjky3pg1nfpu44aoyti
docker@swarm-manager:~$ docker service inspect myapp --pretty
ID:           7bb8pgwjky3pg1nfpu44aoyti
```

```
Name:          myapp
Mode:          Replicated
 Replicas:     1
Placement:
UpdateConfig:
 Parallelism: 1
 On failure:  pause
ContainerSpec:
 Image:               nginx
Resources:
Ports:
 Protocol = tcp
 TargetPort = 80
 PublishedPort = 80
docker@swarm-manager:~$
```

Open the web browser, you will see the nginx page http://192.168.99.103/

Scale a Service

We can use `docker service scale` to scale a service.

```
docker@swarm-manager:~$ docker service scale myapp=2
myapp scaled to 2
docker@swarm-manager:~$ docker service inspect myapp --pretty
ID:           7bb8pgwjky3pg1nfpu44aoyti
Name:         myapp
Mode:         Replicated
 Replicas:    2
Placement:
UpdateConfig:
 Parallelism: 1
 On failure:  pause
ContainerSpec:
 Image:               nginx
Resources:
Ports:
 Protocol = tcp
 TargetPort = 80
 PublishedPort = 80
```

In this example, we scale the service to 2 replicas.

1.2.22 Docker Swarm with Load Balancing and Scaling

Create a Swarm Cluster

Reference *Swarm Mode: Create a Docker Swarm Cluster* to create a swarm cluster which has four node (one manger node and three worker node).

```
  ~ docker-machine ls
NAME                  ACTIVE   DRIVER       STATE     URL                         ␣
↪SWARM   DOCKER    ERRORS
local-swarm-manager   -        virtualbox   Running   tcp://192.168.99.100:2376       ␣
↪    v1.12.5
```

```
local-swarm-worker1   -        virtualbox   Running   tcp://192.168.99.101:2376
↪    v1.12.5
local-swarm-worker2   -        virtualbox   Running   tcp://192.168.99.102:2376
↪    v1.12.5
local-swarm-worker3   -        virtualbox   Running   tcp://192.168.99.103:2376
↪    v1.12.5
  ~ docker-machine ssh local-swarm-manager
docker@local-swarm-manager:~$ docker node ls
ID                           HOSTNAME             STATUS  AVAILABILITY  MANAGER STATUS
3oseehppjrgkslxug746bfzvg    local-swarm-worker2  Ready   Active
4wi3zg11lghywrz3c3lph5929    local-swarm-worker3  Ready   Active
64m0c4gyewt7si74idd2lbi16    local-swarm-worker1  Ready   Active
9r994lgqivf2dr0v02np63co3 *  local-swarm-manager  Ready   Active        Leader
docker@local-swarm-manager:~$
```

Create a Service

Create a service with cmd `docker service create`.

```
docker@local-swarm-manager:~$ docker service create --replicas 1 --name helloworld --
↪publish 80:8000 jwilder/whoami
docker@local-swarm-manager:~$ docker service ls
ID            NAME        REPLICAS  IMAGE           COMMAND
4issxzw4mknz  helloworld  1/1       jwilder/whoami
docker@local-swarm-manager:~$ docker service ps helloworld
ID                         NAME          IMAGE           NODE                DESIRED
↪STATE  CURRENT STATE            ERROR
4m3bbm16oqqw0tafznii7cell  helloworld.2  jwilder/whoami  local-swarm-worker2  Running
↪       Running 8 minutes ago
docker@local-swarm-manager:~$
```

We use docker image `jwilder/whoami` [1] which is a simple HTTP docker service that return it's container ID. It will export port 8000 by default, we use `--publish 80:8000` to publish its http port to 80.

It will return the container host name when we use curl to access the service like:

```
docker@local-swarm-manager:~$ curl 127.0.0.1
I\'m 6075d1ad668c
docker@local-swarm-manager:~$
```

Scale a Service

Use command `docker service scale` to scale a service.

```
docker@local-swarm-manager:~$ docker service ps helloworld
ID                         NAME              IMAGE           NODE
↪DESIRED STATE  CURRENT STATE               ERROR
9azr7sushz03hmequqw24o9kf  helloworld.1      jwilder/whoami  local-swarm-worker3
↪Running        Preparing about a minute ago
4m3bbm16oqqw0tafznii7cell  helloworld.2      jwilder/whoami  local-swarm-worker2
↪Running        Running 10 minutes ago
eoiym8q7gqpwg1o6k0oys9bod  helloworld.3      jwilder/whoami  local-swarm-worker1
↪Running        Running 59 seconds ago
2klxh8c8m3m8jctmqclnj8awg  helloworld.4      jwilder/whoami  local-swarm-manager
↪Running        Running 59 seconds ago
```

[1] https://github.com/jwilder/whoami

```
dopnnfmpfqgvhwvel42vl2yw5  helloworld.5      jwilder/whoami  local-swarm-worker3
↪Running        Preparing about a minute ago
docker@local-swarm-manager:~$ docker service ls
ID            NAME        REPLICAS  IMAGE           COMMAND
4issxzw4mknz  helloworld  3/5       jwilder/whoami
```

There are four helloworld replicas, and two of them are preparing because it need download the docker image.

We can use `curl` to test it again.

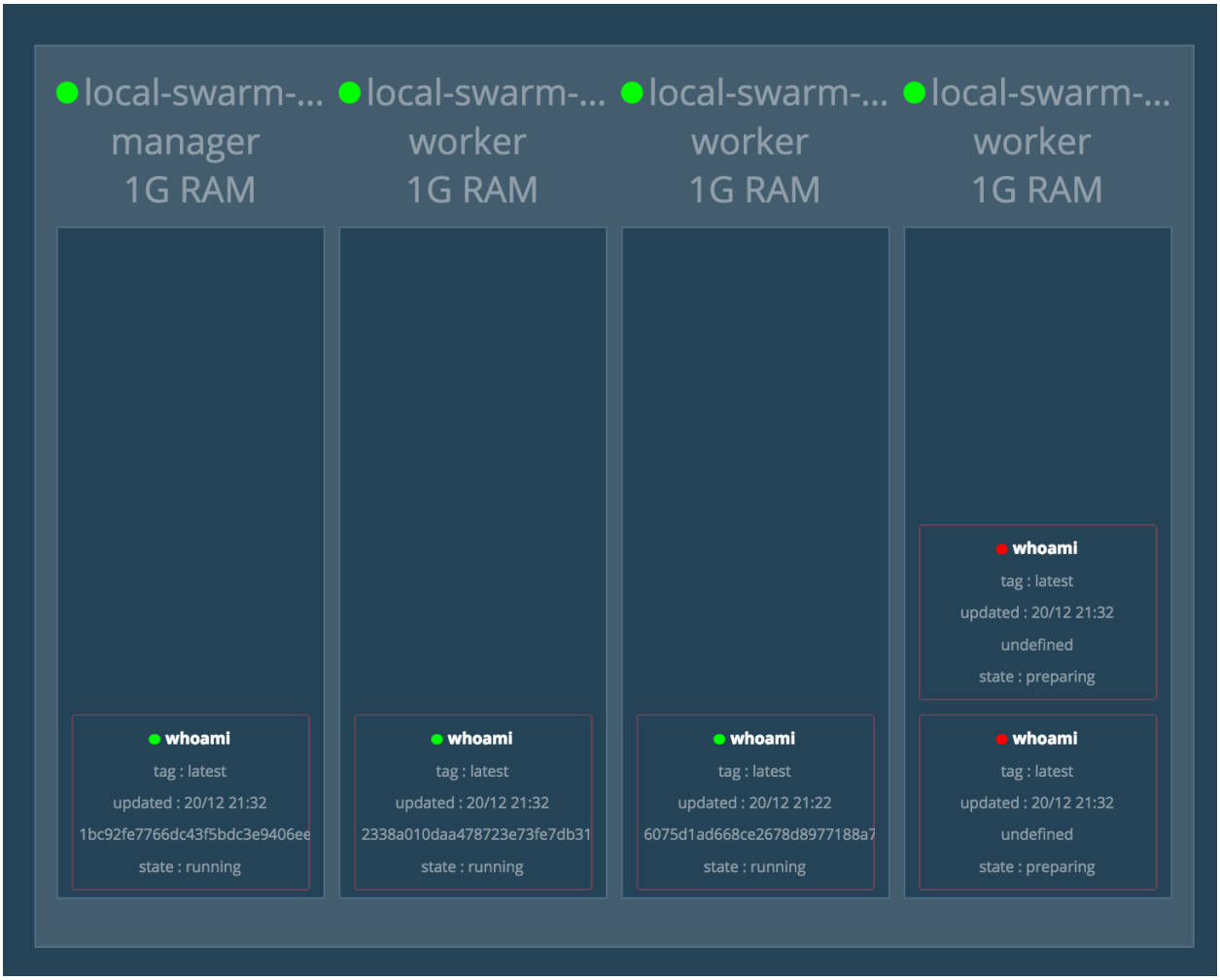
```
docker@local-swarm-manager:~$ for i in `seq 4`; do curl 127.0.0.1; done
I\'m 2338a010daa4
I\'m 1bc92fe7766d
I\'m 6075d1ad668c
I\'m 2338a010daa4
docker@local-swarm-manager:~$
```

it's load balancing!

Visualization Swarm Cluster

There is a visualizer for Docker Swarm Mode using the Docker Remote API, Node.JS, and D3 [2]. Start it on the manager node, then through web browser, we can get the picture like:

[2] https://github.com/ManoMarks/docker-swarm-visualizer



Reference

1.2.23 Docker Swarm Topology Deep Dive

1.3 Kubernetes

1.3.1 Create a Kubernetes Cluster on AWS

In this tutorial, we will create a Kubernetes Cluster on AWS different A-Zone, and will reference this https://kubernetes.io/docs/admin/multiple-zones/

Please make sure you have installed `awscli` (https://aws.amazon.com/cli/)

Create the cluster

```
curl -sS https://get.k8s.io | MULTIZONE=true KUBERNETES_PROVIDER=aws KUBE_AWS_ZONE=us-
↪west-2a NUM_NODES=1 bash
```

This command will create a k8s cluster which include one master node and one worker node.

Add more nodes to the cluster

```
KUBE_USE_EXISTING_MASTER=true MULTIZONE=true KUBERNETES_PROVIDER=aws KUBE_AWS_ZONE=us-
↪west-2b NUM_NODES=2 KUBE_SUBNET_CIDR=172.20.1.0/24 MASTER_INTERNAL_IP=172.20.0.9␣
↪kubernetes/cluster/kube-up.sh
```

This will create two worker nodes in another zone `us-west-2b`.

Check our cluster

```
  ~ kubectl get nodes --show-labels
NAME                                         STATUS    AGE       LABELS
ip-172-20-0-157.us-west-2.compute.internal   Ready     1h        beta.kubernetes.io/
↪arch=amd64,beta.kubernetes.io/instance-type=t2.micro,beta.kubernetes.io/os=linux,
↪failure-domain.beta.kubernetes.io/region=us-west-2,failure-domain.beta.kubernetes.
↪io/zone=us-west-2a,kubernetes.io/hostname=ip-172-20-0-157.us-west-2.compute.internal
ip-172-20-1-145.us-west-2.compute.internal   Ready     1h        beta.kubernetes.io/
↪arch=amd64,beta.kubernetes.io/instance-type=t2.micro,beta.kubernetes.io/os=linux,
↪failure-domain.beta.kubernetes.io/region=us-west-2,failure-domain.beta.kubernetes.
↪io/zone=us-west-2b,kubernetes.io/hostname=ip-172-20-1-145.us-west-2.compute.internal
ip-172-20-1-194.us-west-2.compute.internal   Ready     1h        beta.kubernetes.io/
↪arch=amd64,beta.kubernetes.io/instance-type=t2.micro,beta.kubernetes.io/os=linux,
↪failure-domain.beta.kubernetes.io/region=us-west-2,failure-domain.beta.kubernetes.
↪io/zone=us-west-2b,kubernetes.io/hostname=ip-172-20-1-194.us-west-2.compute.internal
  ~
```

If you want to know what happened during these shell command, please go to https://medium.com/@canthefason/kube-up-i-know-what-you-did-on-aws-93e728d3f56a#.r3ynj2ooe

1.3.2 Create a Kubernetes Cluster on AWS with Tectonic

Please check the Youtube

https://www.youtube.com/watch?v=wwho8DsN5iU&list=PLfQqWeOCIH4AF-4IUpHZaEdlQOkkVt-0D&index=12

1.3.3 Get Start with minikube

1.3.4 Get Started with Kubeadm

We will create a three nodes kubernetes cluster with `kubeadm`.

Prepare three vagrant hosts

```
$ git clone https://github.com/xiaopeng163/docker-k8s-lab
$ cd docker-k8s-lab/lab/k8s/multi-node/vagrant
$ vagrant up
$ vagrant status
Current machine states:

k8s-master                running (virtualbox)
k8s-worker1               running (virtualbox)
k8s-worker2               running (virtualbox)
```

`docker` `kubelet` `kubeadm` `kubectl` `kubernetes-cni` are already installed on each host.

Initialize master node

Use `kubeadm init` command to initialize the master node just like `docker swarm`.

```
ubuntu@k8s-master:~$ sudo kubeadm init --api-advertise-addresses=192.168.205.10
[kubeadm] WARNING: kubeadm is in alpha, please do not use it for production clusters.
[preflight] Running pre-flight checks
[init] Using Kubernetes version: v1.5.1
[tokens] Generated token: "af6b44.f383a4116ef0d028"
[certificates] Generated Certificate Authority key and certificate.
[certificates] Generated API Server key and certificate
[certificates] Generated Service Account signing keys
[certificates] Created keys and certificates in "/etc/kubernetes/pki"
[kubeconfig] Wrote KubeConfig file to disk: "/etc/kubernetes/kubelet.conf"
[kubeconfig] Wrote KubeConfig file to disk: "/etc/kubernetes/admin.conf"
[apiclient] Created API client, waiting for the control plane to become ready
[apiclient] All control plane components are healthy after 61.784561 seconds
[apiclient] Waiting for at least one node to register and become ready
[apiclient] First node is ready after 3.004480 seconds
[apiclient] Creating a test deployment
[apiclient] Test deployment succeeded
[token-discovery] Created the kube-discovery deployment, waiting for it to become␣
↪ready
[token-discovery] kube-discovery is ready after 21.503085 seconds
[addons] Created essential addon: kube-proxy
[addons] Created essential addon: kube-dns

Your Kubernetes master has initialized successfully!

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
    http://kubernetes.io/docs/admin/addons/

You can now join any number of machines by running the following on each node:

kubeadm join --token=af6b44.f383a4116ef0d028 192.168.205.10
```

Join worker nodes

Run `kubeadm join` on each worker node to join the kubernetes cluster.

```
ubuntu@k8s-worker1:~$ kubeadm join --token=af6b44.f383a4116ef0d028 192.168.205.10
ubuntu@k8s-worker2:~$ kubeadm join --token=af6b44.f383a4116ef0d028 192.168.205.10
```

Use `kubectl get nodes` to check the cluster information.

```
ubuntu@k8s-master:~$ kubectl get nodes
NAME          STATUS         AGE
k8s-master    Ready,master   10m
k8s-worker1   Ready          1m
k8s-worker2   Ready          3s
```

1.3.5 Kubernetes Architecture Step by Step

We will have a overview of k8s architecture through this lab step by step.

Prepare Lab Enviroment

We will install kubernetes with Vagrant & CoreOS reference by https://coreos.com/kubernetes/docs/latest/kubernetes-on-vagrant.html.

```
  vagrant git:(master) vagrant status
Current machine states:

e1                        running (virtualbox)
c1                        running (virtualbox)
w1                        running (virtualbox)
w2                        running (virtualbox)
w3                        running (virtualbox)

This environment represents multiple VMs. The VMs are all listed
above with their current state. For more information about a specific
VM, run `vagrant status NAME`.
```

One etcd node, one controller node and three worker nodes.

Kubectl version and cluster information

```
  vagrant git:(master) kubectl version
Client Version: version.Info{Major:"1", Minor:"5", GitVersion:"v1.5.1", GitCommit:
↪"82450d03cb057bab0950214ef122b67c83fb11df", GitTreeState:"clean", BuildDate:"2016-
↪12-14T00:57:05Z", GoVersion:"go1.7.4", Compiler:"gc", Platform:"darwin/amd64"}
Server Version: version.Info{Major:"1", Minor:"5", GitVersion:"v1.5.1+coreos.0",␣
↪GitCommit:"cc65f5321f9230bf9a3fa171155c1213d6e3480e", GitTreeState:"clean",␣
↪BuildDate:"2016-12-14T04:08:28Z", GoVersion:"go1.7.4", Compiler:"gc", Platform:
↪"linux/amd64"}
  vagrant git:(master)
  vagrant git:(master) kubectl get nodes
NAME           STATUS                     AGE
172.17.4.101   Ready,SchedulingDisabled   32m
172.17.4.201   Ready                      32m
172.17.4.202   Ready                      32m
172.17.4.203   Ready                      32m
  vagrant git:(master)
  kubernetes-101 git:(master)  kubectl cluster-info
Kubernetes master is running at https://172.17.4.101:443
Heapster is running at https://172.17.4.101:443/api/v1/proxy/namespaces/kube-system/
↪services/heapster
KubeDNS is running at https://172.17.4.101:443/api/v1/proxy/namespaces/kube-system/
↪services/kube-dns
kubernetes-dashboard is running at https://172.17.4.101:443/api/v1/proxy/namespaces/
↪kube-system/services/kubernetes-dashboard

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
  kubernetes-101 git:(master)
```

Get the application we will deploy from github:

```
$ git clone https://github.com/xiaopeng163/kubernetes-101
```

This application is a simple python flask web app with a redis server as backend.

Create Pods

Use cmd `kubectl create` to create a pod through a yml file. Firstly, create a redis server pod.

```
  kubernetes-101 git:(master)  cd Kubernetes
  Kubernetes git:(master)  ls
db-pod.yml  db-svc.yml  set.sh      web-pod.yml web-rc.yml  web-svc.yml
  Kubernetes git:(master)
  Kubernetes git:(master)  kubectl create -f db-pod.yml
pod "redis" created
  Kubernetes git:(master)  kubectl get pods -o wide
NAME      READY     STATUS    RESTARTS   AGE       IP           NODE
redis     1/1       Running   0          1m        10.2.26.2   172.17.4.201
```

It created a pod which running redis, and the pod is on node `w1`. We can SSH to this node and check the exactly container created by kubernetes.

```
  vagrant git:(master) vagrant ssh w1
CoreOS alpha (1164.1.0)
Last login: Mon Jan  9 06:33:50 2017 from 10.0.2.2
core@w1 ~ $ docker ps
CONTAINER ID    IMAGE           COMMAND                  CREATED          STATUS
↪   PORTS    NAMES
7df09a520c43    redis:latest    "docker-entrypoint.sh"   19 minutes ago   Up 19
↪minutes           k8s_redis.afd331f6_redis_default_b6c27624-d632-11e6-b809-
↪0800274503e1_fb526620
```

Next, create a web server pod.

```
  Kubernetes git:(master)  kubectl create -f web-pod.yml
pod "web" created
  Kubernetes git:(master)  kubectl get pods -o wide
NAME      READY     STATUS    RESTARTS   AGE       IP           NODE
redis     1/1       Running   0          2h        10.2.26.2   172.17.4.201
web       1/1       Running   0          6m        10.2.14.6   172.17.4.203
  Kubernetes git:(master)
```

The web pod is running on node `w3`.

Create Services

Now we have two pods, but they do not know each other. If you SSH to the `w3` node which `web` located on, and access the flask web, it will return a error.

```
core@w3 ~ $ curl 10.2.14.6:5000
.....
.....
ConnectionError: Error -2 connecting to redis:6379. Name or service not known.

-->
core@w3 ~ $
```

The reason is the `web` pod can not resolve the `redis` name. We need to create a service.

```
  Kubernetes git:(master)  kubectl create -f db-svc.yml
service "redis" created
  Kubernetes git:(master)  kubectl get svc
NAME         CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
kubernetes   10.3.0.1     <none>        443/TCP    3h
redis        10.3.0.201   <none>        6379/TCP   42s
```

After that, go to `w3` and access the flask web again, it works!

```
core@w3 ~ $ curl 10.2.14.6:5000
Hello Container World! I have been seen 1 times.
core@w3 ~ $ curl 10.2.14.6:5000
Hello Container World! I have been seen 2 times.
core@w3 ~ $
```

At last, we need to access the flask web service from the outside of the kubernetes cluster, that need to create another service.

```
  Kubernetes git:(master)  kubectl create -f web-svc.yml
service "web" created
  Kubernetes git:(master)
  Kubernetes git:(master)  kubectl get svc
NAME         CLUSTER-IP   EXTERNAL-IP   PORT(S)        AGE
kubernetes   10.3.0.1     <none>        443/TCP        3h
redis        10.3.0.201   <none>        6379/TCP       11m
web          10.3.0.51    <nodes>       80:32204/TCP   5s
  Kubernetes git:(master)  curl 172.17.4.203:32204
Hello Container World! I have been seen 3 times.
  Kubernetes git:(master)
  Kubernetes git:(master)  curl 172.17.4.201:32204
Hello Container World! I have been seen 4 times.
  Kubernetes git:(master)  curl 172.17.4.202:32204
Hello Container World! I have been seen 5 times.
  Kubernetes git:(master)
```

Now we can access the flask web from the outside, actually from any node.

Scaling Pods with Replication Controller

```
  Kubernetes git:(master)  kubectl create -f web-rc.yml
replicationcontroller "web" created
  Kubernetes git:(master)  kubectl get pods -o wide
NAME         READY     STATUS    RESTARTS   AGE       IP           NODE
redis        1/1       Running   0          3h        10.2.26.2    172.17.4.201
web          1/1       Running   0          57m       10.2.14.6    172.17.4.203
web-jlzm4    1/1       Running   0          3m        10.2.71.3    172.17.4.202
web-sz150    1/1       Running   0          3m        10.2.26.3    172.17.4.201
  Kubernetes git:(master)
```

Rolling Update

To update a service without an outage through rolling update. We will update our flask web container image from 1.0 to 2.0.

```
  kubernetes-101 git:(master)  kubectl get pods
NAME         READY     STATUS    RESTARTS   AGE
redis        1/1       Running   0          6h
web          1/1       Running   0          4h
web-jlzm4    1/1       Running   0          3h
web-sz150    1/1       Running   0          3h
  kubernetes-101 git:(master)  kubectl rolling-update web --image=xiaopeng163/docker-
 ↪flask-demo:2.0
Created web-db65f4ce913c452364a2075625221bec
Scaling up web-db65f4ce913c452364a2075625221bec from 0 to 3, scaling down web from 3␣
 ↪to 0 (keep 3 pods available, do not exceed 4 pods)
Scaling web-db65f4ce913c452364a2075625221bec up to 1
Scaling web down to 2
Scaling web-db65f4ce913c452364a2075625221bec up to 2
Scaling web down to 1
Scaling web-db65f4ce913c452364a2075625221bec up to 3
Scaling web down to 0
Update succeeded. Deleting old controller: web
Renaming web to web-db65f4ce913c452364a2075625221bec
replicationcontroller "web" rolling updated
  kubernetes-101 git:(master)  kubectl get pods
NAME                                          READY     STATUS    RESTARTS   AGE
redis                                         1/1       Running   0          6h
web-db65f4ce913c452364a2075625221bec-13011    1/1       Running   0          3m
web-db65f4ce913c452364a2075625221bec-85365    1/1       Running   0          4m
web-db65f4ce913c452364a2075625221bec-tsr41    1/1       Running   0          2m
  kubernetes-101 git:(master)
```

After update, check the service.

```
  kubernetes-101 git:(master)  for i in `seq 4`; do curl 172.17.4.203:32204; done
Hello Container World! I have been seen 26 times and my hostname is web-
 ↪db65f4ce913c452364a2075625221bec-13011.
Hello Container World! I have been seen 27 times and my hostname is web-
 ↪db65f4ce913c452364a2075625221bec-85365.
Hello Container World! I have been seen 28 times and my hostname is web-
 ↪db65f4ce913c452364a2075625221bec-13011.
Hello Container World! I have been seen 29 times and my hostname is web-
 ↪db65f4ce913c452364a2075625221bec-13011.
  kubernetes-101 git:(master)
```

We can see it automatically load balanced.

Clear Environment

```
$ kubectl delete services web
$ kubectl delete services redis
$ kubectl delete rc web
$ kubectl delete pod redis
$ kubectl delete pod web
```

1.4 CoreOS

CHAPTER 2

Feedback

Please go to github https://github.com/xiaopeng163/docker-k8s-lab and create issue or PR, thanks.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search