DCCN Docker Swarm Cluster Documentation

Release 1.0.0

Hurng-Chun Lee

Nov 30, 2018

Contents

CHAPTER 1

Introduction to Docker Swarm

1.1 Docker in a Nutshell

- what is docker?
- Learning docker

1.2 Docker swarm cluster

- docker swarm overview
- Raft consensus
- Swarm administration guide

CHAPTER 2

Terminology

Docker engine is the software providing the libraries, services and toolsets of Docker. It enables computer to build Docker images and lauch Docker containers. Docker engine has two different editions: the community edition (**Docker CE**) and the enterprise edition (**Docker EE**).

Docker node/host is a physical or virtual computer on which the Docker engine is enabled.

Docker swarm cluster is a group of connected Docker nodes. Each node has either a **manager** or **worker** role in the cluster. At least one master node is required for a docker swarm cluster to function.

Manager refers to the node maintaining the state of a docker swarm cluster. There can be one or more managers in a cluster. The more managers in the cluster, the higher level of the cluster fault-tolerance. The level of fault-tolerance is explained in this document.

Worker refers to the node sharing the container workload in a docker swarm cluster.

Docker image is an executable package that includes everything needed to run an application–the code, a runtime, libraries, environment variables, and configuration files.

Docker container is a runtime instance of an image. A container is launched by running an Docker image.

Docker service is a logical representation of multiple replicas of the same container. Replicas are used for service load-balancing and/or failover.

Docker stack is a set of linked **Docker services**.

CHAPTER 3

Docker swarm cluster at DCCN

The first swarm cluster at DCCN was developed in order to deploy and manage service components (e.g. DICOM services, data streamer, data stager) realising the automatic lab-data flow. The inital setup consists of 8 nodes repurposed from the HPC and the EXSi clusters.

3.1 System architecture

All docker nodes are bare-matel machines running CentOS operating system. The nodes are provisioned using the DCCN linux-server kickstart. They all NFS-mount the `/home` and `/project` directories, and use the active directory service for user authentication and authorisation. Only the TG members are allowed to SSH login to the docker nodes.

All docker nodes also NFS-mount the `/mnt/docker` directory for sharing container data. The figure below shows the architecture of the DCCN swarm cluster.

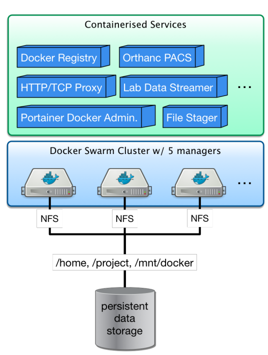

Fig. 3.1: The DCCN swarm cluster - a simplified illustration of the architecture.

3.2 Image registry

Within the swarm cluster, a private image registry is provided to as a central repository of all container images. The data store of the registry is located in `/mnt/docker/registry` which is a shared NFS volume on the central storage.

The registry endpoint is `docker-registry.dccn.nl:5000`. It requires user authentication for uploading (push) and downloading (pull) container images. New user can be added by using the script `/mnt/docker/scripts/microservices/registry/add-user.sh`.

An overview of image repositories can be browsed here.

Note: For the sake of simplicity, the internal private registry is using a self-signed X.509 certificate. In order to trust it, one needs to copy the certificate of the docker registry server to the docker host, under the directory, e.g. `/etc/docker/certs.d/docker-registry.dccn.nl:5000/ca.crt`.

3.3 Service orchestration

For deploying multiple service components as a single application stack, the docker compose specification v3 is used together with the docker stack management interface (i.e. the `docker stack` command).

An example docker-compose file for orchestrating three services for the data-stager application is shown below:

```
version: "3"

services:

    db:
        image: docker-registry.dccn.nl:5000/redis
        volumes:
            - /mnt/docker/data/stager/ui/db:/data
        networks:
            default:
                aliases:
                    - stagerdb4ui
        deploy:
            placement:
                constraints: [node.labels.function == production]

    service:
        image: docker-registry.dccn.nl:5000/stager:1.7.0
        ports:
            - 3100:3000
        volumes:
            - /mnt/docker/data/stager/config:/opt/stager/config
            - /mnt/docker/data/stager/cron:/cron
            - /mnt/docker/data/stager/ui/log:/opt/stager/log
            - /project:/project
            - /var/lib/sss/pipes:/var/lib/sss/pipes
            - /var/lib/sss/mc:/var/lib/sss/mc:ro
        networks:
            default:
                aliases:
                    - stager4ui
```

(continues on next page)

(continued from previous page)

```
        environment:
            - REDIS_HOST=stagerdb4ui
            - REDIS_PORT=6379
        depends_on:
            - db
        deploy:
            placement:
                constraints: [node.labels.function == production]

    ui:
        image: docker-registry.dccn.nl:5000/stager-ui:1.1.0
        ports:
            - 3080:3080
        volumes:
            - /mnt/docker/data/stager/ui/config:/opt/stager-ui/config
        networks:
            default:
                aliases:
                    - stager-ui
        depends_on:
            - service
        deploy:
            placement:
                constraints: [node.labels.function == production]

networks:
    default:
```

Whenever the docker compose specification is not applicable, a script to start a docker service is provided. It is a bash script wrapping around the `docker service create` command.

All the scripts are located in the `/mnt/docker/scripts/microservices` directory.

CHAPTER 4

Swarm cluster operation procedures

4.1 Cluster initialisation

Note: In most of cases, there is no need to initialse another cluster.

Before there is anything, a cluster should be initialised. Simply run the command below on a docker node to initialise a new cluster:

```
$ docker swarm init
```

4.1.1 Force a new cluster

In case the quorum of the cluster is lost (and you are not able to bring other manager nodes online again), you need to reinitiate a new cluster forcefully. This can be done on one of the remaining manager node using the following command:

```
$ docker swarm init --force-new-cluster
```

After this command is issued, a new cluster is created with only one manager (i.e. the one on which you issued the command). All remaining nodes become workers. You will have to add additional manager nodes manually.

Tip: Depending on the number of managers in the cluster, the required quorum (and thus the level of fail tolerance) is different. Check this page for more information.

4.2 Node operation

4.2.1 System provisioning

The operating system and the docker engine on the node is provisioned using the DCCN linux-server kickstart. The following kickstart files are used:

- `/mnt/install/kickstart-*/ks-*-dccn-dk.cfg`: the main kickstart configuration file
- `/mnt/install/kickstart-*/postkit-dccn-dk/script-selection`: main script to trigger post-kickstart scripts
- `/mnt/install/kickstart-*/setup-docker-*`: the docker-specific post-kickstart scripts

Configure devicemapper to direct-lvm mode

By default, the devicemapper storage drive of docker is running the loop-lvm mode which is known to be suboptimal for performance. In a production environment, the direct-lvm mode is recommended. How to configure the devicemapper to use direct-lvm mode is described here.

Before configuring the direct-lvm mode for the devicemapper, make sure the directory */var/lib/docker* is removed. Also make sure the physical volume, volume group, logical volumes are removed, e.g.

```
$ lvremove /dev/docker/thinpool
$ lvremove /dev/docker/thinpoolmeta
$ vgremove docker
$ pvremove /dev/sdb
```

Hereafter is a script summarizing the all steps. The script is also available at `/mnt/install/kickstart-7/docker/docker-thinpool.sh`.

```
#!/bin/bash

if [ $# -ne 1 ]; then
    echo "USAGE: $0 <device>"
    exit 1
fi

# get raw device path (e.g. /dev/sdb) from the command-line argument
device=$1

# check if the device is available
file -s ${device} | grep 'cannot open'
if [ $? -eq 0 ]; then
    echo "device not found: ${device}"
    exit 1
fi

# install/update the LVM package
yum install -y lvm2

# create a physical volume on device
pvcreate ${device}

# create a volume group called 'docker'
vgcreate docker ${device}

# create logical volumes within the 'docker' volume group: one for data, one␣
↪for metadate
```

(continues on next page)

(continued from previous page)

```
# assign volume size with respect to the size of the volume group
lvcreate --wipesignatures y -n thinpool docker -l 95%VG
lvcreate --wipesignatures y -n thinpoolmeta docker -l 1%VG
lvconvert -y --zero n -c 512K --thinpool docker/thinpool --poolmetadata docker/thinpoolmeta

# update the lvm profile for volume autoextend
cat >/etc/lvm/profile/docker-thinpool.profile <<EOL
activation {
    thin_pool_autoextend_threshold=80
    thin_pool_autoextend_percent=20
}
EOL

# apply lvm profile
lvchange --metadataprofile docker-thinpool docker/thinpool

lvs -o+seg_monitor

# create daemon.json file to instruct docker using the created logical volumes
cat >/etc/docker/daemon.json <<EOL
{
    "hosts": ["unix:///var/run/docker.sock", "tcp://0.0.0.0:2375"],
    "storage-driver": "devicemapper",
    "storage-opts": [
         "dm.thinpooldev=/dev/mapper/docker-thinpool",
         "dm.use_deferred_removal=true",
         "dm.use_deferred_deletion=true"
    ]
}
EOL

# remove legacy deamon configuration through docker.service.d to avoid confliction with daemon.json
if [ -f /etc/systemd/system/docker.service.d/swarm.conf ]; then
    mv /etc/systemd/system/docker.service.d/swarm.conf /etc/systemd/system/docker.service.d/swarm.conf.bk
fi

# reload daemon configuration
systemctl daemon-reload
```

4.2.2 Join the cluster

After the docker daemon is started, the node should be joined to the cluster. The command used to join the cluster can be retrieved from one of the manager node, using the command:

```
$ docker swarm join-token manager
```

Note: The example command above obtains the command for joining the cluster as a manager node. For joining the cluster as a worker, replace the `manager` on the command with `worker`.

After the command is retrieved, it should be run on the node that is about to join to the cluster.

4.2.3 Set Node label

Node label helps group nodes in certain features. Currently, the node in production is labled with `function=production` using the following command:

```
$ docker node update --label-add function=production <NodeName>
```

When deploying a service or stack, the label is used for locate service tasks.

4.2.4 Leave the cluster

Run the following command on the node that is about to leave the cluster.

```
$ docker swarm leave
```

If the node is a manager, the option `-f` (or `--force`) should also be used in the command.

Note: The node leaves the cluster is **NOT** removed automatically from the node table. Instead, the node is marked as `Down`. If you want the node to be removed from the table, you should run the command `docker node rm`.

Tip: An alternative way to remove a node from the cluster directly is to run the `docker node rm` command on a manager node.

4.2.5 Promote and demote node

Node in the cluster can be demoted (from manager to worker) or promoted (from worker to manager). This is done by using the command:

```
$ docker node promote <WorkerNodeName>
$ docker node demote <ManagerNodeName>
```

4.2.6 Monitor nodes

To list all nodes in the cluster, do

```
$ docker node ls
```

To inspect a node, do

```
$ docker node inspect <NodeName>
```

To list tasks running on a node, do

```
$ docker node ps <NodeName>
```

4.3 Service operation

In swarm cluster, a service is created by deploying a container in the cluster. The container can be deployed as a singel instance (i.e. task) or multiple instances to achieve service failover and load-balancing.

4.3.1 Start a service

To start a service in the cluster, one uses the `docker service create` command. Hereafter is an example for starting a `nginx` web service in the cluster using the container image `docker-registry.dccn.nl:5000/nginx:1.0.0`:

```
$ docker login docker-registry.dccn.nl:5000
$ docker service create \
--name webapp-proxy \
--replicas 2 \
--publish 8080:80/tcp \
--constaint "node.labels.function == production" \
--mount "type=bind,source=/mnt/docker/webapp-proxy/conf,target=/etc/nginx/conf.d" \
--with-registry-auth \
docker-registry.dccn.nl:5000/nginx:1.0.0
```

Options used above is explained in the following table:

option	function
`--name`	set the service name to `webapp-proxy`
`--replicas`	deploy `2` tasks in the cluster for failover and loadbalance
`--publish`	map internal `tcp` port `80` to `8080`, and expose it to the world
`--constaint`	restrict the tasks to run on nodes labled with `function = production`
`--mount`	mount host's `/mnt/docker/webapp-proxy/conf` to container's `/etc/nginx/conf.d`

More options can be found here.

4.3.2 Remove a service

Simply use the `docker service rm <ServiceName>` to remove a running service in the cluster. It is not normal to remove a productional service.

Tip: In most of cases, you should consider **updating the service** rather than removing it.

4.3.3 Update a service

It is very common to update a productional service. Think about the following conditions that you will need to update the service:

- a new node is being added to the cluster, and you want to move an running service on it, or
- a new container image is being provided (e.g. software update or configuration changes) and you want to update the service to this new version, or
- you want to create more tasks of the service in the cluster to distribute the load.

To update a service, one uses the command `docker service update`. The following example update the `webapp-proxy` service to use a new version of nginx image `docker-registry.dccn.nl:5000/nginx:1.2.0`:

```
$ docker service update \
--image docker-registry.dccn.nl:5000/nginx:1.2.0 \
webapp-proxy
```

More options can be found here.

4.3.4 Monitor services

To list all running services:

```
$ docker service ls
```

To list tasks of a service:

```
$ docker service ps <ServieName>
```

To inspect a service:

```
$ docker service inspect <ServiceName>
```

To retrieve logs written to the STDOU/STDERR by the service process, one could do:

```
$ docker service logs [-f] <ServiceName>
```

where the option `-f` is used to follow the output.

4.4 Stack operation

A stack is usually defined as a group of related services. The defintion is described using the docker-compose version 3 specification.

Here is *an example* of defining the three services of the DCCN data-stager.

Using the `docker stack` command you can manage multiple services in one consistent manner.

4.4.1 Deploy (update) a stack

Assuming the docker-compose file is called `docker-compose.yml`, to launch the services defined in it in the swarm cluster is:

```
$ docker login docker-registry.dccn.nl:5000
$ docker stack deploy -c docker-compose.yml --with-registry-auth <StackName>
```

When there is an update in the stack description file (e.g. `docker-compose.yml`), one can use the same command to apply changes on the running stack.

Note: Every stack will be created with an overlay network in swarm, and organise services within the network. The name of the network is `<StackName>_default`.

4.4.2 Remove a stack

Use the following command to remove a stack from the cluster:

```
$ docker stack rm <StackName>
```

4.4.3 Monitor stacks

To list all running stacks:

```
$ docker stack ls
```

To list all services in a stack:

```
$ docker stack services <StackName>
```

To list all tasks of the services in a stack:

```
$ docker stack ps <StackName>
```

4.5 Emergancy shutdown

Note: The emergency shutdown should take place **before** the network and the central storage are down.

1. login to one manager
2. *demote* other managers
3. remove running *stacks* and *services*
4. shutdown all workers
5. shutdown the manager

4.5.1 Reboot from shutdown

Note: In several network outage in 2017 and 2018, the cluster nodes were not reacheable and required hard (i.e. push the power button) to reboot. In this case, the emergancy shutdown procedure was not followed. Interestingly, the cluster was recovered automatically after sufficient amount of master nodes became online. All services were also re-deployed immediately without any human intervention.

One thing to notice is that if the network outage causes the NFS mount to `/mnt/docker` not accessible, one may need to reboot the machines once the network connectivity is recovered as they can be irresponsive due to the hanging NFS connections.

1. boot on the manager node (the last one being shutted down)
2. boot on other nodes
3. *promote nodes* until a desired number of managers is reached
4. deploy firstly the docker-registry stack

```
$ cd /mnt/docker/scripts/microservices/registry/
$ sudo ./start.sh
```

Note: The docker-registry stack should be firstly made available as other services/stacks will need to pull container images from it.

5. deploy other stacks and services

4.6 Disaster recovery

Hopefully there is no need to go though it!!

For the moment, we are not backing up the state of the swarm cluster. Given that the container data has been stored (and backedup) on the central storage, the impact of losing a cluster is not dramatic (as long as the container data is available, it is already possible to restart all services on a fresh new cluster).

Nevertheless, here is the official instruction of disaster recovery.

CHAPTER 5

Docker swarm health monitoring

Various management and monitoring web-based tools can be found on http://docker.dccn.nl.

The health of the swarm nodes are monitored by the Xymon monitor.

CHAPTER 6

Tutorial: basic

This tutorial is based on an example of building and running a container of the Apache HTTPd server which serves a simple PHP-based helloworld application. Throught the tutorial you will learn:

- the docker workflow and basic UI commands,
- network port mapping,
- data persistency

6.1 Preparation

Files used in this tutorial are available on GitHub. Preparing those files within the `~/tmp` using the commands below:

```
$ mkdir -p ~/tmp
$ cd ~/tmp
$ wget https://github.com/Donders-Institute/docker-swarm-setup/raw/master/doc/
↪tutorial/centos-httpd/basic.tar.gz
$ tar xvzf basic.tar.gz
$ cd basic
$ ls
Dockerfile  Dockerfile_php  htmldoc  run-httpd.sh
```

6.2 The Dockerfile

Before starting a container with Docker, we need a docker container image that is either pulled from a image registry (a.k.a. docker registry), such as the Docker Hub, or built by ourselves. In this exercise, we are going to build a container image ourselves.

For building a docker image, one starts with writing an instruction file known as the Dockerfile.

Dockerfile is a YAML document describing how a docker container should be built. Hereafter is an example of the Dockerfile for an Apache HTTPd image:

```
1  FROM centos:7
2  MAINTAINER The CentOS Project <cloud-ops@centos.org>
3  LABEL Vendor="CentOS" \
4        License=GPLv2 \
5        Version=2.4.6-40
6
7
8  RUN yum -y --setopt=tsflags=nodocs update && \
9      yum -y --setopt=tsflags=nodocs install httpd && \
10     yum clean all
11
12 EXPOSE 80
13
14 # Simple startup script to avoid some issues observed with container restart
15 ADD run-httpd.sh /run-httpd.sh
16 RUN chmod -v +x /run-httpd.sh
17
18 CMD ["/run-httpd.sh"]
```

The Dockerfile above is explained below.

Each line of the Dockerfile is taken as a *step* of the build. It started with a **keyword** followed by **argument(s)**.

Line 1: all container images are built from a basis image. This is indicated by the `FROM` keyword. In this example, the basis image is the official CentOS 7 image from the Docker Hub.

Line 2-3: a container image can be created with metadata. For instance, the `MAINTAINER` and `LABEL` attributes are provided in the example.

Line 8-10: given that we want to build a image for running the Apache HTTPd server, we uses the YUM package manager to install the `httpd` package within the container. It is done by using the `RUN` keyword followed by the actual YUM command.

Line 12: we know that the HTTPd service will run on port number 80, we expose that port explicitly for the connectivity.

Line 14: comments in Dockerfile are started with the `#`.

Line 15: the run-httpd.sh is a script for bootstraping the HTTPd service. It is the main program to be executed after the container is started. In order to make this script available in the image, we use the `ADD` keyword here. The example here can be interpreted as *copying the file "run-httpd.sh" on the host to file "/run-http.sh" in the container image*.

Line 16: here we make the bootstrap script in the container image executable so that it can be run directly. It is done using the `RUN` keyword again.

Line 18: the keyword `CMD` specifies the command to be executed when the container is started. Here we simply run the bootstrap script we have just copied into the container.

6.3 Building the container image

With the Dockerfile in place, we can proceed for building the container image. Make sure you are in the `basic` folder, and run the following command:

```
$ docker build -t httpd:centos .
```

Here we give the image a *name:tag* with the `-t` option. With that, the image can be later referred by `httpd:centos`.

Keep your eyes on the output of the build process. You will find the steps in the Dockerfile are executed sequencially, and some output (e.g. the output from `yum install`) looks like as if you are running in a CentOS7 system.

What interesting to notice are lines with hash strings. For example:

```
---> 5182e96772bf
Step 2/8 : MAINTAINER The CentOS Project <cloud-ops@centos.org>
---> Running in 52daee99ca6c
Removing intermediate container 52daee99ca6c
---> cf9a7fe73efc
```

6.3.1 Image layers

During the build process, each step in the Dockerfile triggers creation of two image layers. One intermediate layer for executing the step; the other is a persistent layer containing results of the step. Those layers are indicated by the hash strings we see in the output snippet above.

The intermediate layer is forked from the persistent layer of the previous step, except for the first step on which the persistent image is always from an existing image built somewhere else (a reason that we always see keyword `FROM` as the first step in the Dockerfile). The intermediate layer is removed after the execution of the step.

Each persistent layer only consists of the "delta" to the one from its previous step. As illustrated in Fig. 6.1, the final image is then constructed as a stack of those persisten layers; and it is locked for read-only.

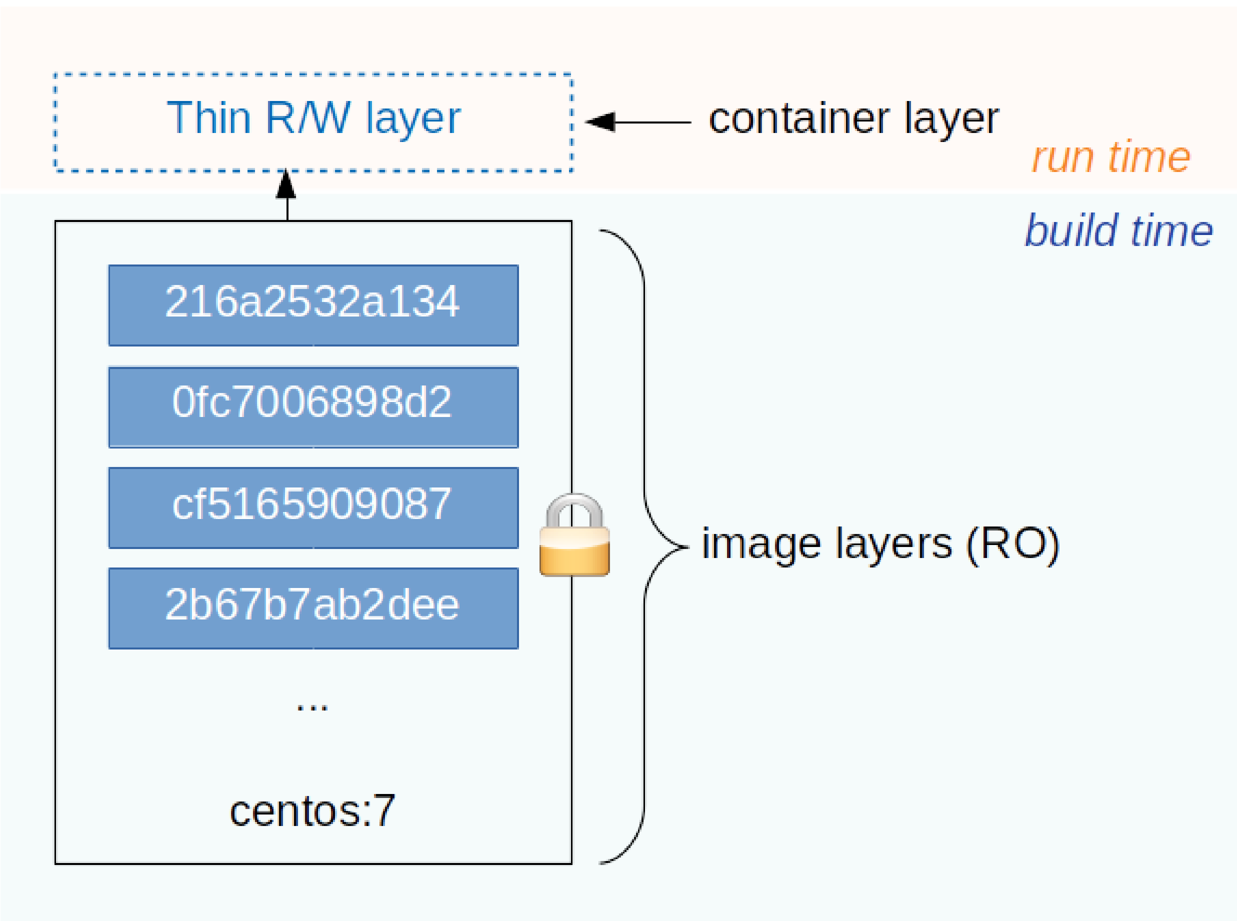

Fig. 6.1: an illustration of the Docker image and container layers. This figure is inspired by the one on the Docker document.

Persistent layers are reused when they are encountered in different/independent build processes. For example, the persistent layer created by the first step (`FROM centos:7`) is very likely to be reused for building a variety of container images based on CentOS 7. In this case, Docker will reuse the image downloaded before instead of duplicating it for using the host's storage efficiently.

The image layers of a final docker image can be examinated by the `docker history <image name:tag>` command. For example,

```
$ docker history httpd:centos
```

6.4 Running the container

With the image built successfully, we can now start a container with the image using the `docker run [options] <image name:tag>` command. For example,

```
$ docker run --rm -d -p 8080:80 --name myhttpd httpd:centos
```

Let's connect the browser to the URL http://localhost:8080. You will see a default welcome page of the Apache HTTPd server.

A few options are used here:

Option `--rm` instructs Docker to remove the container layer (see below) when the container is stopped.

Option `-d` instructs Docker to run the container in a detached mode.

Option `-p` instructs Docker to map the host's network port `8080` to the container's network port `80` so that this service is accessible from the host's external network.

Option `--name` names the container so that the container can be later referred easily.

6.4.1 Container layer

When running the container from a image, Docker creates a new writable layer (a.k.a. container layer) on top of the image layers. Changes made within the container are delta to the image layers and kept in this container layer. In this way, Docker makes the image layers read-only; and thus can be used by multiple independent containers without interference.

Note: In fact, the way Docker organise deltas in the image layers and the container layer is similar to how the Linux life CD manages the filesystems. They are both based on a stackable filesystem with the Copy-on-Write (CoW) strategy.

The concept of the image layers and the container layer is illustrated in Fig. 6.1.

6.4.2 Exercise: PHP with MySQL support

Can you extend/modify the `Dockerfile` and build a image called `php:centos`? In this image, we want to add PHP with MySQL support to the Apache HTTPd server.

The container should be started with

```
$ docker run --rm -d -p 8080:80 --name myphp php:centos
```

Hint: In a CentOS system, one can just run `yum -y install php php-mysql` to add PHP with MySQL support to the Apache HTTPd server.

To verify the PHP support, you can create a file `/var/www/html/index.php` in the container, and visit the page http://localhost:8080/index.php. Hereafter is an example:

```
$ docker exec -it myphp bash
$ cat > /var/www/html/index.php <<EOF
<?php phpinfo(); ?>
EOF
```

6.5 Network port mapping

Networkk port mapping is the way of making the container service accessible to the network of the host.

In the Dockerfile example above, we explicitly expose the port `80` as we know that the HTTPd will listen on this TCP port.

However, the container runs in an internal virtual network, meaning that our HTTPd service is not accessible from the network on which the host is running.

To make the service accessible externally, one uses the `-p` option to map the host's port to the container's port. For instance, the option `-p 8080:80` implies that if the client connects to the port `8080` of the host, the connection will be redirected to the port `80` of the container.

6.5.1 Exercise: network

How do you make the HTTPd container accessible on port `80`?

6.6 Data persistency

The default welcome page of the Apache HTTPd is boring. We are going to create our own homepage.

Let's access to the bash shell of the running httpd container:

```
$ docker exec -it myphp bash
```

In Apache HTTPd, the way to replace the default homepage is creating our own `index.html` file within the folder `/var/www/html`. For example, using the command below to create a HTML form in `/var/www/html/index.html`:

```
$ cat > /var/www/html/index.html <<EOF
<html>
<head></head>
<body>
<h2>Welcome to my first HTML page served by Docker</h2>
<form action="hello.php" method="POST">
    Your name: <input type="text" name="name"></br>
    Your email: <input type="text" name="email"></br>
<input value="submit" name="submit" type="submit">
</form>
```

(continues on next page)

(continued from previous page)

```
</body>
</html>
EOF
```

If you revisit the page http://localhost:8080/index.html, you will see the new homepage we just created.

Now imaging that we have to restart the container for a reason. For that, we do:

```
$ docker stop myphp
$ docker run --rm -d -p 8080:80 --name myphp php:centos
```

Try connect to the page http://localhost:8080/index.html again with the browser. **Do you see the homepage we just added to the container?**

Hint: Changes made in the container are stored in the container layer which is only available during the container's lifetime. When you stop the container, the container layer is removed from the host and thus the data in this layer is **NOT** persistent.

6.6.1 Volumes

One way to persistent container data is using the so-called *volumes*. Volumes is managed by Docker and thus it is more portable and manageable.

For the example above, we could create a volume in Docker as

```
$ docker volume create htmldoc
```

Hint: One could use `docker volume ls` and `docker volume inspect` to list and inspect detail of a Docker volume.

When the volume is available, one could map the volume into the container's path `/var/www/html`, using the `-v` option (i.e. line 3 of the command block below) at the time of starting the container.

```
1 $ docker stop myphp
2 $ docker run --rm -d -p 8080:80 \
3 -v htmldoc:/var/www/html \
4 --name myphp php:centos
```

Now get into the shell of the container, and create our own `index.html` again:

```
$ docker exec -it myphp bash
$ cat > /var/www/html/index.html <<EOF
<html>
<head></head>
<body>
<h2>Welcome to my first HTML page served by Docker</h2>
<form action="hello.php" method="POST">
    Your name: <input type="text" name="name"></br>
    Your email: <input type="text" name="email"></br>
<input value="submit" name="submit" type="submit">
</form>
</body>
```

(continues on next page)

(continued from previous page)

```
</html>
EOF
$ exit
```

Check if the new `index.html` is in place by reloading the page http://localhost:8080/index.html.

Restart the container again:

```
$ docker stop myphp
$ docker run -rm -d -p 8080:80 \
-v htmldoc:/var/www/html \
--name myphp php:centos
```

You should see that our own `index.html` page is still available after restarting the container.

If you want to start from the scratch without any container data, one can simply remove the volume followed by creating a new one.

```
$ docker volume rm htmldoc
$ docker volume create htmldoc
```

6.6.2 Bind mounts

Bind mount is another way of keeping container data persistent by binding host's filesystem structure into the container.

In the files you downloaded to the host you are working on, there is a directory called `htmldoc`. In this directory, we have prepared our `index.html` file.

```
$ ls ~/tmp/basic/htmldoc
hello.php index.html
```

By binding the directory `~/basic/htmldoc` into the container's `/var/www/html` directory, the `index.html` file will appear as `/var/www/html/index.html` in the container. This is done by the following command at the time of starting the container:

```
$ docker stop myphp
$ docker run --rm -d -p 8080:80 \
-v ~/tmp/basic/htmldoc:/var/www/html \
--name myphp php:centos
```

Hint: While doing the bind mounts in the container, the benefit is that one can change the files on the host and the changes will take effect right in the container. In addition, if new files are created in the container, they will also appear on the host.

6.6.3 Exercise: preserving HTTPd's log files

We know that the log files of the Apache HTTPd server are located in `/var/log/httpd`. How do you make those log files persistent?

CHAPTER 7

Tutorial: single-host orchestration

This tutorial focuses on orchestrating multiple containers together on a single Docker host as an application stack. For doing so, we will be using the docker-compose tool.

The application we are going to build is a user registration application. The application has two interfaces, one is the user registration form; the other is an overview of all registered users. Information of the registered users will be stored in the MySQL database; while the interfaces are built with PHP.

Throught the tutorial you will learn:

- the docker-compose file
- the usage of the docker-compose tool

7.1 Preparation

The docker-compose tool is not immediately available after the Docker engine is installed on Linux. Nevertheless, the installation is very straightforward as it's just a single binary file to be downloaded. Follow the commands below to install it:

```
$ sudo curl -L https://github.com/docker/compose/releases/download/1.22.0/docker-
↪compose-$(uname -s)-$(uname -m) \
-o /usr/local/bin/docker-compose
$ chmod +x /usr/local/bin/docker-compose
$ docker-compose --version
```

Files used in this tutorial are available on GitHub. Preparing those files within the `~/tmp` using the commands below:

```
$ mkdir -p ~/tmp
$ cd ~/tmp
$ wget https://github.com/Donders-Institute/docker-swarm-setup/raw/master/doc/
↪tutorial/centos-httpd/orchestration.tar.gz
$ tar xvzf orchestration.tar.gz
$ cd orchestration
```

(continues on next page)

(continued from previous page)

```
$ ls
app  cleanup.sh  docker-compose.yml  initdb.d
```

Important: In order to make the following commands in this tutorial work, you also need to prepare the files we used in the *Tutorial: basic* section.

7.2 The docker-compose file

Container orchestration is to manage multiple containers in a controlled manner so that they work together as a set of integrated components. The docker-compose file is to describe the containers and their relationship in the stack. The docker-compose file is also written in YAML. Hereafter is the docker-compose file for our user registration application.

Tip: The filename of the docker-compose file is usually `docker-compose.yml` as it is the default the `docker-compose` tool looks up in the directory.

```
version: '3.1'

networks:
    dbnet:

services:
    db:
        image: mysql:latest
        hostname: db
        command: --default-authentication-plugin=mysql_native_password
        environment:
            - MYSQL_ROOT_PASSWORD=admin123
            - MYSQL_DATABASE=registry
            - MYSQL_USER=demo
            - MYSQL_PASSWORD=demo123
        volumes:
            - ./initdb.d:/docker-entrypoint-initdb.d
            - ./data:/var/lib/mysql
        networks:
            - dbnet
    web:
        build:
            context: ../basic
            dockerfile: Dockerfile_php
        image: php:centos
        volumes:
            - ./app:/var/www/html
            - ./log:/var/log/httpd
        networks:
            - dbnet
        ports:
            - 8080:80
        depends_on:
            - db
```

The docker-compose file above implements a service architecture shown in Fig. 7.1 where we have two services (`web` and `db`) running in a internal network `dbnet` created on-demand.

Tip: The docker-compose file starts with the keyword `version`. It is important to note that keywords of the docker-compose file are supported differently in different Docker versions. Thus, the keyword `version` is to tell the docker-compose tool which version it has to use for interpreting the entire docker-compose file.

The compatibility table can be found here.

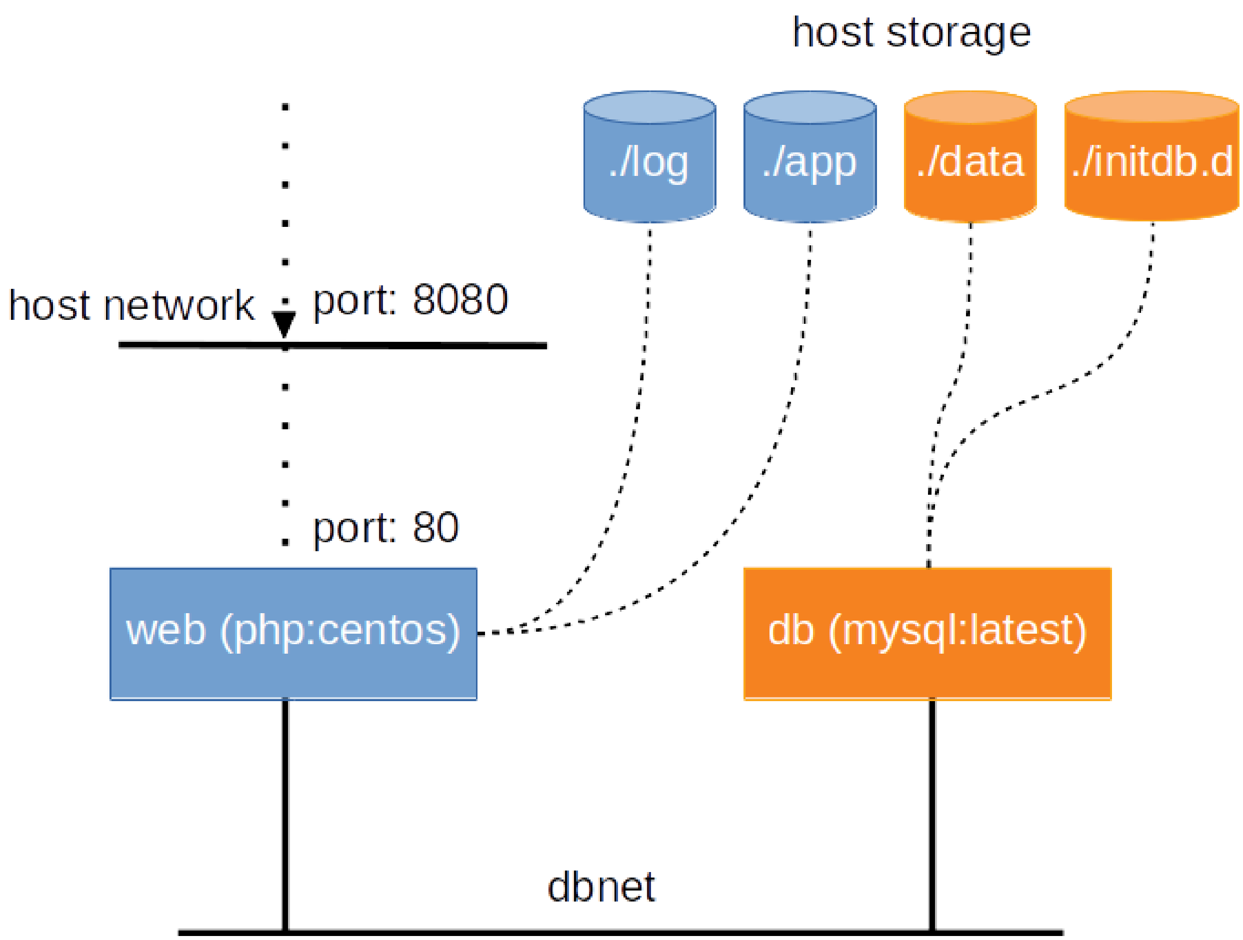

Fig. 7.1: an illustration of the service architecture implemented by the docker-compose file used in this tutorial.

The service `web` uses the `php:centos` image we have built in *Tutorial: basic*. It has two bind-mounts: one for the application codes (i.e. HTML and PHP files) and the other for making the HTTPd logs persistent on the host. The `web` service is attached to the `dbnet` network and has its network port 80 mapped to the port 8080 on the host. Furthermore, it waits for the readiness of the `db` service before it can be started.

Another service `db` uses the official MySQL image from the Docker Hub. According to the documentation of this official MySQL image, commands and environment variables are provided for initialising the database for our user registration application.

The `db` service has two bind-mounted volumes. The `./init.d` directory on host is bind-mounted to the `/docker-entrypoint-initdb.d` directory in the container as we will make use the bootstrap mechanism provided by the container to create a database schema for the `registry` database; while the `./data` is bind-mounted to `/var/lib/mysql` for preserving the data in the MySQL database. The `db` service is also joint into the `dbnet` network so that it becomes accessible to the `web` service.

7.3 Building services

When the service stack has a container based on local image build (e.g. the `web` service in our example), it is necessary to build the container via the docker-compose tool. For that, one can do:

```
$ docker-compose build --force-rm
```

Tip: The command above will loads the `docker-compose.yml` file in the current directory. If you have a different filename/location for your docker-compose file, add the `-f <filepath>` option in front of the `build` command.

7.4 Bringing services up

Once the docker-compose file is reasy, bring the whole service stack up is very simple. Just do:

```
$ docker-compose up -d
Creating network "orchestration_dbnet" with the default driver
Creating orchestration_db_1 ...
Creating orchestration_db_1 ... done
Creating orchestration_web_1 ...
Creating orchestration_web_1 ... done
```

Let's check our user registration application by connecting the browser to http://localhost:8080.

7.4.1 service status

```
$ docker-compose ps
       Name                        Command                State          Ports
-----------------------------------------------------------------------------------
orchestration_db_1    docker-entrypoint.sh --def ...   Up      3306/tcp, 33060/tcp
orchestration_web_1   /run-httpd.sh                    Up      0.0.0.0:8080->80/tcp
```

7.4.2 service logs

The services may produce logs to its STDOUT/STDERR. Those logs can be monitored using

```
$ docker-compose logs -f
```

where the option `-f` follows the output on STDOUT/STDERR.

7.5 Bringing services down

```
$ docker-compose down
Stopping orchestration_web_1 ...
Stopping orchestration_db_1  ...
Removing orchestration_web_1 ... done
Removing orchestration_db_1  ... done
Removing network orchestration_dbnet
```

7.6 Exercise: HAProxy

In this exercise, you are going to update the docker-compose file to add on top of the web service a HAProxy loadbalancer. The overall architecture looks like the figure below:

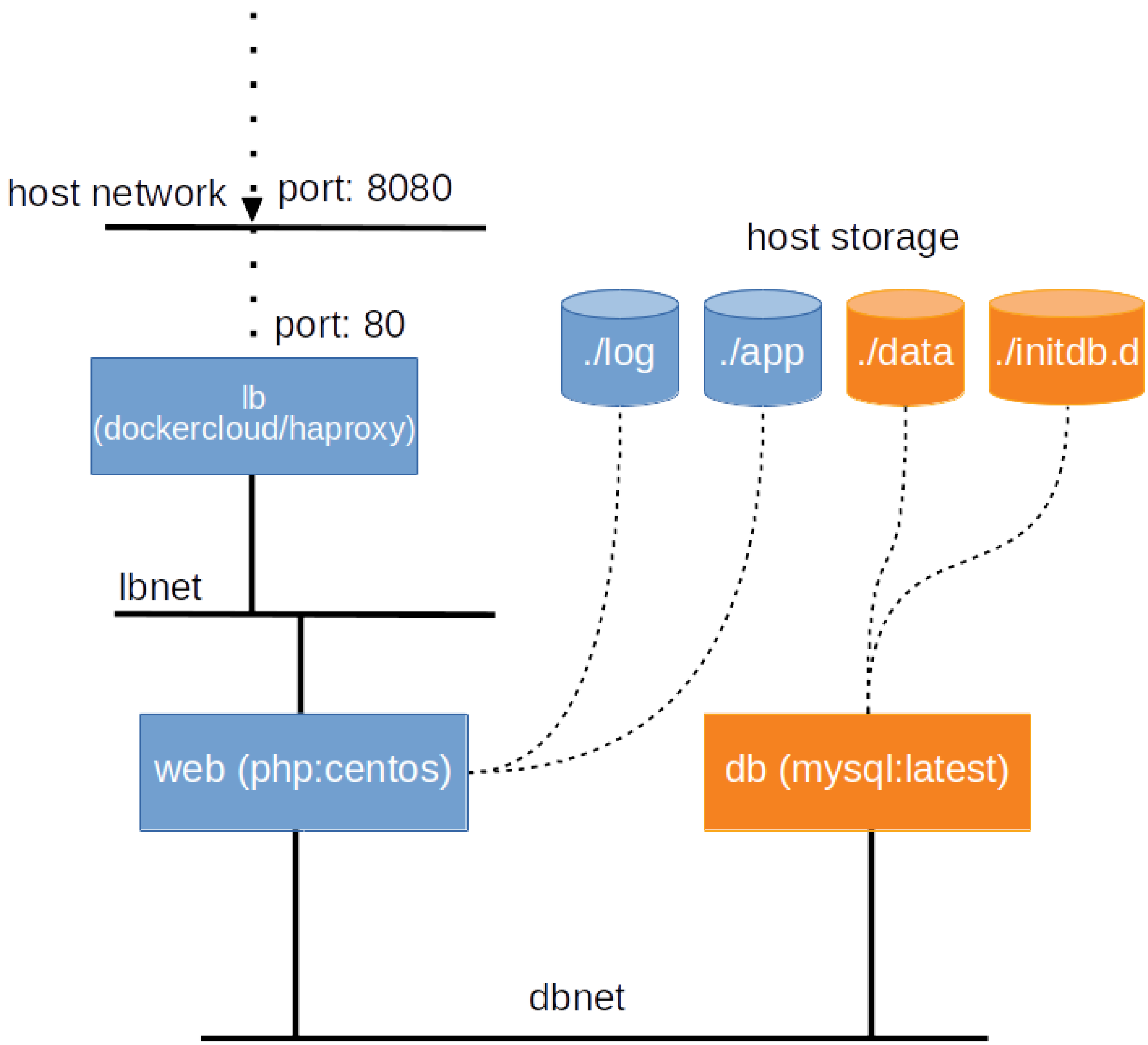

Fig. 7.2: an illustration of the service architecture with HAProxy as the loadbalancer.

7.6.1 Step 1: add service `dockercloud/haproxy`

The HAProxy we are going to use is customised by DockerCloud, and is available here. Adding the following service description into the `docker-compose.yml` file.

```
lb:
    image: dockercloud/haproxy
    volumes:
        - /var/run/docker.sock:/var/run/docker.sock
    links:
        - web
    ports:
        - 8080:80
    depends_on:
        - web
    networks:
        - lbnet
```

Tip: In real-world situation, it is very often to use existing container images from the Docker Hub. It is a good practise to read the usage of the container image before using it.

7.6.2 Step 2: adjust `web` service

Task 1

From the documentation of the `dockercloud/haproxy`, it requires services attached to the proxy to set an environment variable `SERVICE_PORT`. The `SERVICE_PORT` of the `web` service is 80.

Could you modify the docker-compose file accordingly for it?

Task 2

Instead of mapping host port 8080 to container port 80, we just need to join the `web` service into the network of the loadbalancer.

Could you modify the docker-compose file accordingly for it?

7.6.3 Step 3: add `lbnet` network

We have made use of the network `lbnet`; but we haven't ask the docker-compose to create it.

Could you modify the docker-compose file accordingly so that the network `lbnet` is created when bring up the services?

7.6.4 Service scaling

The final docker-compose file is available here.

Save the file as `docker-compose.lb.yml` in the `~/tmp/orchestration` directory; and do the following to start the services:

```
$ docker-compose -f docker-compose.lb.yml build --force-rm
$ docker-compose -f docker-compose.lb.yml up
```

Try connecting to http://localhost:8080. You should see the same user registration application. The difference is that we are now accessing the web service through the HAProxy.

With this setting, we can now scale up the web service whenever there is a load. For example, to create 2 loadbalancing instances of the web service, one does:

```
$ docker-compose -f docker-compose.lb.yml scale web=2
```

Check the running processes with

```
$ docker-compose -f docker-compose.lb.yml ps
       Name                          Command                 State                        Ports
--------------------------------------------------------------------------------------
↪--------------
orchestration_db_1    docker-entrypoint.sh --def ...   Up      3306/tcp, 33060/tcp
orchestration_lb_1    /sbin/tini -- dockercloud- ...   Up      1936/tcp, 443/tcp,0.0.
↪0.0:8080->80/tcp
orchestration_web_1   /run-httpd.sh                    Up      80/tcp
orchestration_web_2   /run-httpd.sh                    Up      80/tcp
```

You should see two web services running on port 80. You could try the followng curl command to check whether the loadbalancer does its job well:

```
$ for i in {1..10}; do curl http://localhost:8080 2>/dev/null \
| grep 'Served by host'; done
```

CHAPTER 8

Tutorial: Docker swarm

In the previous tutorial, we have learnd about container orchestration for running a service stack with a feature of load balance. However, the whole stack is running on a single Docker host, meaning that there will be service interruption when the host is down, a single point of failure.

In this tutorial, we are going to eliminate this single point of failure by orchestrating containers in a cluster of Docker nodes, a Docker swarm cluster. We will revisit our web application developed in the *Tutorial: single-host orchestration* session, and make the web service redundent for eventual node failure.

You will learn:

- how to create a swarm cluster from scratch,
- how to deploy a stack in a swarm cluster,
- how to manage the cluster.

Tip: Docker swarm is not the only solution for orchestrating containers on multiple computers. A platform called Kubenetes was originally developed by Google and used in the many container infrastructure.

8.1 Preparation

For this tutorial, we need multiple Docker hosts to create a swarm cluster. For that, we are going to use the docker machine, a light weight virtual machine with Docker engine.

We will need to install VirtualBox on the computer as the hypervisor for running the docker machines. Follow the commands below to install the VirtualBox RPM.

```
$ wget https://download.virtualbox.org/virtualbox/5.2.22/VirtualBox-5.2-5.2.22_126460_
↪el7-1.x86_64.rpm
$ sudo yum install VirtualBox-5.2-5.2.22_126460_el7-1.x86_64.rpm
```

Next step is to download the files prepared for this exercise:

```
$ mkdir -p ~/tmp
$ cd ~/tmp
$ wget https://github.com/Donders-Institute/docker-swarm-setup/raw/master/doc/
↪tutorial/centos-httpd/swarm.tar.gz
$ tar xvzf swarm.tar.gz
$ cd swarm
```

Install the *docker-machine* tool following this page.

Bootstrap two docker machines with the prepared script:

```
$ ./docker-machine-bootstrap.sh vm1 vm2
```

For your convenience, open two new terminals, each logs into one of the two virtual machines. For example, on terminal one, do

```
$ docker-machine ssh vm1
```

On the second terminal, do

```
$ docker-machine ssh vm2
```

8.2 Architecture

The architecture of the Docker swarm cluster is relatively simple comparing to other distributed container orchestration platforms. As illustrated in Fig. 8.1, each Docker host in the swarm cluster is either a *manager* or a *worker*.

By design, manager nodes are no difference to the worker nodes in sharing container workload; except that manager nodes are also responsible for maintaining the status of the cluster using a distributed state store. Managers exchange information with each other in order to maitain sufficient quorum of the Raft consensus which is essential to the cluster fault tolerance.

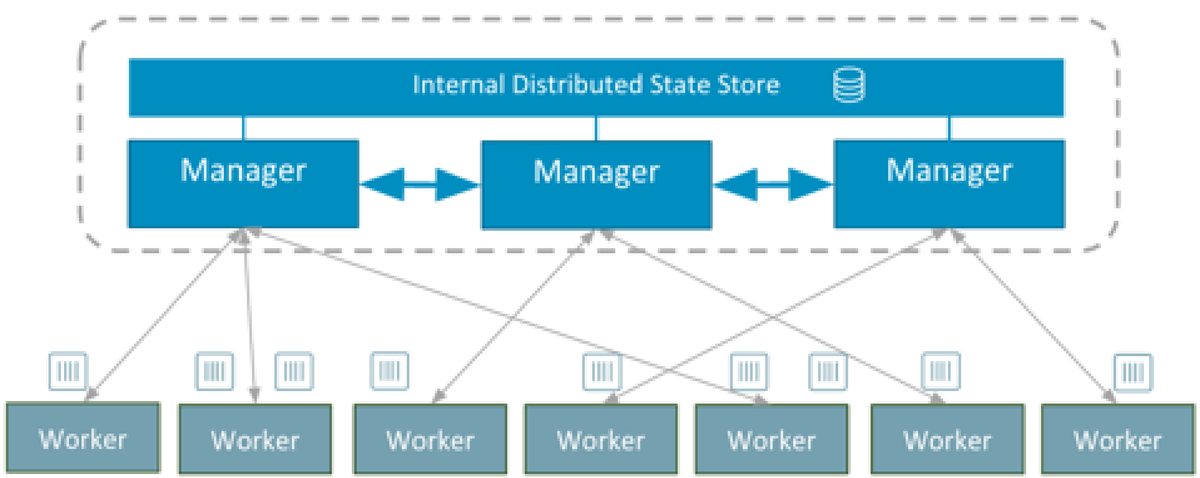

Fig. 8.1: the swarm architecture, an illustration from the docker blog.

8.2.1 Service and stack

In the swarm cluster, a container can be started with multiple instances (i.e. replicas). The term *service* is used to refer to the replicas of the same container.

A *stack* is referred to a group of connected *services*. Similar to the single-node orchestration, a stack is also described by a *docker-compose* file with extra attributes specific for the Docker swarm.

8.3 Creating a cluster

Docker swarm is essentially a "mode" of the Docker engine. This mode has been introduced to the Docker engine since version 1.12 in 2016. To create a new cluster, we just need to pick up the first Docker host (*vm1* for instance) and do:

```
[vm1]$ docker swarm init --advertise-addr 192.168.99.100
```

Note: The `--advertise-addr` should be the IP address of the docker machine. It may be different in different system.

Note: The notation `[vm1]` on the command-line prompt indicates that the command should be executed on the specified docker machine. All the commands in this tutorial follow the same notation. If there is no such notation on the prompt, the command is performed on the host of the docker machines.

After that you could check the cluster using

```
[vm1]$ docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY       ␣
↪    MANAGER STATUS      ENGINE VERSION
svdjh0i3k9ty5lsf4lc9d94mw *   vm1                 Ready               Active             ␣
↪    Leader              18.06.1-ce
```

Et voilà! You have just created a swarm cluster, as simple as one command... Obviously, it is a cluster with only one node, and the node is by default a manager. Since it is the only manager, it is also the leading manager (*Leader*).

8.4 Join tokens

Managers also hold tokens (a.k.a. join token) for other nodes to join the cluster. There are two join tokens; one for joining the cluster as a mansger, the other for a worker. To retrieve the token for the manager, use the following command on the first manager.

```
[vm1]$ docker swarm join-token manager
To add a manager to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-
↪2i60ycz95dbpblm0bewz0fyypwkk5jminbzpyheh7yzf5mvrla-1q74k0ngm0br70ur93h7pzdg4 192.
↪168.99.100:2377
```

For the worker, one does

```
[vm1]$ docker swarm join-token worker
To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-
↪2i60ycz95dbpblm0bewz0fyypwkk5jminbzpyheh7yzf5mvrla-9br20buxcon364sgmdbcfobco 192.
↪168.99.100:2377
```

The output of these two commands simply tells you what to run on the nodes that are about to join the cluster.

8.5 Adding nodes

Adding nodes is done by executing the command suggested by the `docker swarm join-token` on the node that you are about to add. For example, let's add our second docker machine (`vm2`) to the cluster as a manager:

```
[vm2]$ docker swarm join --token \
SWMTKN-1-2i60ycz95dbpblm0bewz0fyypwkk5jminbzpyheh7yzf5mvrla-1q74k0ngm0br70ur93h7pzdg4␣
↪\
192.168.99.100:2377
```

After that, you can see the cluster has more nodes available.

```
[vm2]$ docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        ␣
↪     MANAGER STATUS      ENGINE VERSION
svdjh0i3k9ty5lsf4lc9d94mw     vm1                 Ready               Active              ␣
↪     Leader              18.06.1-ce
m5r1j48nnl1u9n9mbr8ocwoa3 *   vm2                 Ready               Active              ␣
↪     Reachable           18.06.1-ce
```

Note: The `docker node` command is meant for managing nodes in the cluster, and therefore, it can only be executed on the manager nodes. Since we just added `vm2` as a manager, we could do the `docker node ls` right away.

8.5.1 Labeling nodes

Sometimes it is useful to lable the nodes. Node lables are useful for container placement on nodes. Let's now lable the two nodes with *os=linux*.

```
[vm1]$ docker node update --label-add os=linux vm1
[vm1]$ docker node update --label-add os=linux vm2
```

8.5.2 Promoting and demoting nodes

The manager node can demote other manager to become worker or promote worker to become manager. This dynamics allows administrator to ensure sufficient amount of managers (in order to maintain the state of the cluster); while some manager nodes need to go down for maintenance. Let's demote `vm2` from manager to worker:

```
[vm1]$ docker node demote vm2
Manager vm2 demoted in the swarm.

[vm1]$ docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY      ␣
↪    MANAGER STATUS      ENGINE VERSION
svdjh0i3k9ty5lsf4lc9d94mw *   vm1                 Ready               Active            ␣
↪    Leader              18.06.1-ce
m5r1j48nnl1u9n9mbr8ocwoa3     vm2                 Ready               Active            ␣
↪                        18.06.1-ce
```

Promote the `vm2` back to manager:

```
[vm1]$ docker node promote vm2
Node vm2 promoted to a manager in the swarm.
```

8.6 docker-compose file for stack

The following docker-compose file is modified from the one we used in the *Tutorial: single-host orchestration*. Changes are:

- we stripped down the network part,
- we added container placement requirements via the `deploy` section,
- we persistented MySQL data in a docker volume (*due to the fact that I don't know how to make bind-mount working with MySQL container in a swarm of docker machines*),
- we made use of a private docker image registry for the `web` service.

```
version: '3.1'

networks:
    default:

volumes:
    dbdata:
    weblog:

services:
    db:
        image: mysql:latest
        hostname: db
        command: --default-authentication-plugin=mysql_native_password
        environment:
            - MYSQL_ROOT_PASSWORD=admin123
            - MYSQL_DATABASE=registry
            - MYSQL_USER=demo
            - MYSQL_PASSWORD=demo123
        volumes:
            - ./initdb.d:/docker-entrypoint-initdb.d
            - dbdata:/var/lib/mysql
        networks:
            - default
        deploy:
            restart_policy:
```

(continues on next page)

(continued from previous page)

```
                condition: none
            mode: replicated
            replicas: 1
            placement:
                constraints:
                    - node.hostname == vm1
    web:
        build:
            context: ./app
        image: docker-registry.dccn.nl:5000/demo_user_register:1.0
        volumes:
            - weblog:/var/log/httpd
        networks:
            - default
        ports:
            - 8080:80
        depends_on:
            - db
        deploy:
            mode: replicated
            replicas: 1
            placement:
                constraints:
                    - node.hostname == vm2
                    - node.labels.os == linux
```

8.7 Launching stack

The docker-compose file above is already provided as part of the downloaded files in the preparation step. The filename is `docker-compose.swarm.yml`. Follow the steps below to start the application stack.

1. On `vm1`, go to the directory in which you have downloaded the files for this tutorial. It is a directory mounted under the `/hosthome` directory in the VM, e.g.

   ```
   [vm1]$ cd /hosthome/tg/honlee/tmp/swarm
   ```

2. Login to the private Docker registry with user *demo*:

   ```
   [vm1]$ docker login docker-registry.dccn.nl:5000
   ```

3. Start the application stack:

   ```
   [vm1]$ docker stack deploy -c docker-compose.swarm.yml --with-registry-
   ↪auth webapp
   Creating network webapp_default
   Creating service webapp_db
   Creating service webapp_web
   ```

   Note: The `--with-registry-auth` is very important for pulling image from the private repository.

4. Check if the stack is started properly:

```
[vm1]$ docker stack ps webapp
ID                  NAME                IMAGE                                        ␣
↪                   NODE                DESIRED STATE       CURRENT␣
↪STATE           ERROR               PORTS
j2dqr6xs9838        webapp_db.1         mysql:latest                                 ␣
↪                   vm1                 Running             Running 2␣
↪seconds ago
drm7fexzlb9t        webapp_web.1        docker-registry.dccn.nl:5000/demo_
↪user_register:1.0   vm2                 Running             Running 4␣
↪seconds ago
```

5. Note that our web service (`webapp_web`) is running on `vm2`. So it is obvous that if we try to get the index page from `vm2`, it should work. Try the following commands on the host of the two VMs.

```
$ docker-machine ls
NAME    ACTIVE   DRIVER       STATE     URL                         SWARM ␣
↪ DOCKER        ERRORS
vm1     -        virtualbox   Running   tcp://192.168.99.100:2376         ␣
↪ v18.06.1-ce
vm2     -        virtualbox   Running   tcp://192.168.99.101:2376         ␣
↪ v18.06.1-ce

$ curl http://192.168.99.101:8080
```

But you should note that getting the page from another VM `vm1` works as well even though the container is not running on it:

```
$ curl http://192.168.99.100:8080
```

This is the magic of Docker swarm's routing mesh mechanism, which provides intrinsic feature of load balance and failover.

Since we are running this cluster on virtual machines, the web service is not accessible via the host's IP address. The workaround we are doing below is to start a NGINX container on the host, and proxy the HTTP request to the web service running on the VMs.

```
$ cd /home/tg/honlee/tmp/swarm
$ docker-compose -f docker-compose.proxy.yml up -d
$ docker-compose -f docker-compose.proxy.yml ps
Name                 Command            State         Ports
-----------------------------------------------------------------
swarm_proxy_1   nginx -g daemon off;   Up      0.0.0.0:80->80/tcp
```

Tip: This workaround is also applicable for a production environment. Imaging you have a swarm cluster running in a private network, and you want to expose a service to the Internet. What you need is a gateway machine proxying requests from Internet to the internal swarm cluster. NGINX is a very powerful engine for proxying HTTP traffic. It also provides capability of load balancing and failover.

You may want to have a look of the NGINX configuration in the `proxy.conf.d` directory (part of the downloaded files) to see how to leverage on the Docker swarm's routing mesh mechanism (discussed below) for load balance and failover.

8.7.1 Docker registry

One benefit of using Docker swarm is that one can bring down a Docker node and the system will migrate all containers on it to other nodes. This feature assumes that there is a central place where the Docker images can be pulled from.

In the example docker-compose file above, we make use of the official MySQL image from the DockerHub and the `demo_user_register:1.0` image from a private registry, `docker-registry.dccn.nl`. This private registry requires user authentication, therefore we need to login to this registry before starting the application stack.

8.7.2 Overlay network

The following picuture illustats the network setup we have created with the `webapp` stack. The way Docker swarm interconnects containers on different docker hosts is using the so-called *overlay network*.

Technical details on how Docker swarm sets up the overlay network is described in this blog by Nigel Poulton. In short, the overlay network makes use of the virtual extensible LAN (VXLAN) tunnel to route layer 2 traffic accross IP networks.

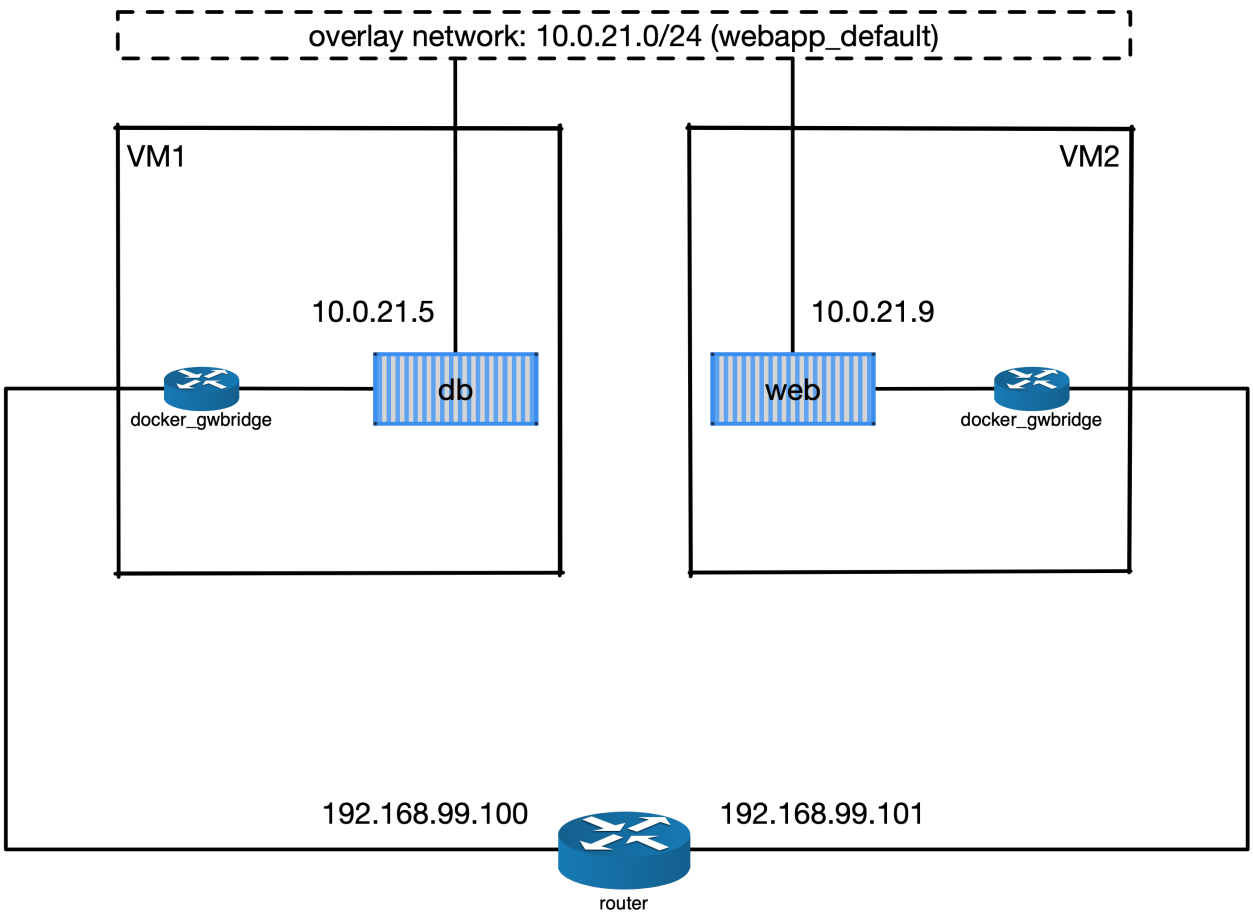

Fig. 8.2: An illustration of the Docker overlay network.

Hint: There are also YouTube videos explaining the Docker overlay network. For example, the Deep dive in Docker Overlay Networks by Laurent Bernaille is worth for watching.

8.7.3 Container placement

You may notice that the containers db and web services are started on a node w.r.t. the container placement requirement we set in the docker-compose file. You can dynamically change the requirement, and the corresponding containers will be moved accordingly to meet the new requirement. Let's try to move the container of the web service from vm2 to vm1 by setting the placement constraint.

Get the current placement constraints:

```
[vm1]$ docker service inspect \
--format='{{.Spec.TaskTemplate.Placement.Constraints}}' webapp_web
[node.hostname == vm2 node.labels.os == linux]
```

Move the container from vm2 to vm1 by removing the constraint `node.hostname == vm2` followed by addeing `node.hostname == vm1`:

```
[vm1]$ docker service update \
--constraint-rm 'node.hostname == vm2' \
--with-registry-auth webapp_web
```

Note: By removing the constraint `node.hostname == vm2`, the container is not actually moved since the node the container is currently running on, vm2, fulfills the other constraint `node.labels.os == linux`.

```
[vm1]$ docker service update \
--constraint-add 'node.hostname == vm1' \
--with-registry-auth webapp_web
```

Check again the location of the container. It should be moved to vm1 due to the newly added constraint.

```
[vm1]$ docker service ps --format='{{.Node}}' webapp_web
vm1

[vm1]$ docker service inspect \
--format='{{.Spec.TaskTemplate.Placement.Constraints}}' webapp_web
[node.hostname == vm1 node.labels.os == linux]
```

Let's now remove the hostname constaint:

```
[vm1]$ docker service update \
--constraint-rm 'node.hostname == vm1' \
--with-registry-auth webapp_web

[vm1]$ docker service inspect \
--format='{{.Spec.TaskTemplate.Placement.Constraints}}' webapp_web
[node.labels.os == linux]
```

8.7.4 Network routing mesh

In the Docker swarm cluster, routing mesh is a mechanism making services exposed to the host's public network so that they can be accessed externally. This mechanism also enables each node in the cluster to accept connections on published ports of any published service, even if the service is not running on the node.

Routing mesh is based on an overlay network (ingress) and a IP Virtual Servers (IPVS) load balancer (via a hindden ingress-sbox container) running on each node of the swarm cluster.

The figure below illustrates the overall network topology of the `webapp` stack with the `ingress` network and `ingress-sbox` load balancer for the routing mesh.

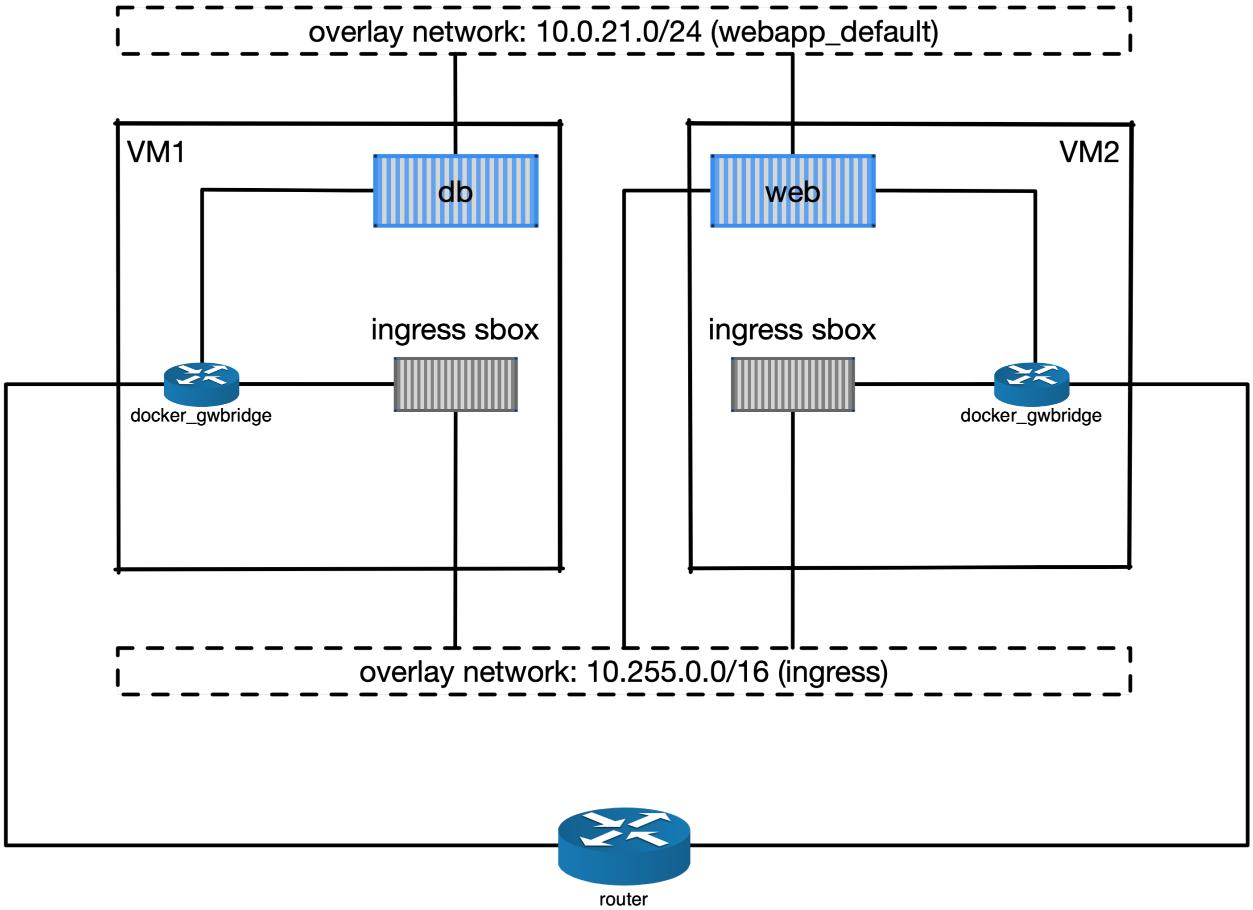

Fig. 8.3: An illustration of the Docker ingress network and routing mesh.

8.8 Service management

8.8.1 Scaling

Service can be scaled up and down by updating the number of *replicas*. Let's scale the `webapp_web` service to 2 replicas:

```
[vm1]$ docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE␣
↪                                                 PORTS
qpzws2b43ttl        webapp_db           replicated          1/1                 ␣
↪mysql:latest
z92cq02bqr4b        webapp_web          replicated          1/1                 ␣
↪docker-registry.dccn.nl:5000/demo_user_register:1.0   *:8080->80/tcp

[vm1]$ docker service update --replicas 2 webapp_web

[vm1]$ docker service ls
```

(continues on next page)

(continued from previous page)

```
ID                  NAME                MODE                REPLICAS            IMAGE
↪                                                 PORTS
qpzws2b43ttl        webapp_db           replicated          1/1
↪mysql:latest
z92cq02bqr4b        webapp_web          replicated          2/2
↪docker-registry.dccn.nl:5000/demo_user_register:1.0    *:8080->80/tcp
```

8.8.2 Rotating update

Since we have two `webapp_web` replicas running in the cluster, we could now perform a rotating update without service downtime.

Assuming that the app developer has update the Docker registry with a new container image, the new image name is `docker-registry.dccn.nl:5000/demo_user_registry:2.0`, and we want to apply this new image in the cluster without service interruption. To achieve it, we do a rotating update on the service `webapp_web`.

To demonstrate the non-interrupted update, let's open a new terminal and keep pulling the web page from the service:

```
$ while true; do curl http://192.168.99.100:8080 2>/dev/null | grep 'Served by host';
↪sleep 1; done
```

Use the following command to perform the rotating update:

```
[vm1]$ docker service update \
--image docker-registry.dccn.nl:5000/demo_user_register:2.0 \
--update-parallelism 1 \
--update-delay 10s \
--with-registry-auth webapp_web
```

8.9 Node management

Sometimes we need to perform maintenance on a Docker node. In the Docker swarm cluster, one first drains the containers on the node we want to maintain. This is done by setting the node's availability to `drain`. For example, if we want to perform maintenance on `vm2`:

```
[vm1]$ docker node update --availability drain vm2
[vm1]$ docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY
↪     MANAGER STATUS      ENGINE VERSION
svdjh0i3k9ty5lsf4lc9d94mw *   vm1                 Ready               Active
↪     Leader              18.06.1-ce
m5r1j48nnl1u9n9mbr8ocwoa3     vm2                 Ready               Drain
↪     Reachable           18.06.1-ce
```

Once you have done that, you will notice all containers running on `vm2` are automatically moved to `vm1`.

```
[vm1]$ docker stack ps webapp
ID                  NAME                IMAGE
↪         NODE                DESIRED STATE       CURRENT STATE              ERROR
↪         PORTS
cwoszv8lupq3        webapp_web.1        docker-registry.dccn.nl:5000/demo_user_
↪register:2.0   vm1                 Running             Running 41 seconds ago
rtv2hndyxveh         \_ webapp_web.1    docker-registry.dccn.nl:5000/demo_user_
↪register:2.0   vm2                 Shutdown            Shutdown 41 seconds ago
```

(continues on next page)

(continued from previous page)

```
3he78fis5jkn         \_ webapp_web.1    docker-registry.dccn.nl:5000/demo_user_
↪register:1.0   vm2                 Shutdown            Shutdown 6 minutes ago
675z5ukg3ian        webapp_db.1        mysql:latest                             ␣
↪         vm1                 Running             Running 14 minutes ago
mj1547pj2ac0        webapp_web.2       docker-registry.dccn.nl:5000/demo_user_
↪register:2.0   vm1                 Running             Running 5 minutes ago
yuztiqacgro0         \_ webapp_web.2    docker-registry.dccn.nl:5000/demo_user_
↪register:1.0   vm1                 Shutdown            Shutdown 5 minutes ago
```

After the maintenance work, just set the node's availability to `active` again:

```
[vm1]$ docker node update --availability active vm2
```

And run the following command to rebalance the service so that two replicas runs on two different nodes:

```
[vm1]$ docker service update --force --with-registry-auth webapp_web
```