DCCN Docker Swarm Cluster Documentation

Release 1.0.0

Hurng-Chun Lee

Sep 12, 2017

Contents

CHAPTER 1

Introduction to Docker Swarm

Docker in a Nutshell

- what is docker?
- Learning docker

Docker swarm cluster

- docker swarm overview
- Raft consensus
- Swarm administration guide

CHAPTER 2

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.

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.

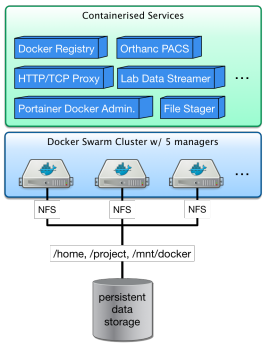

Fig. 2.1: The DCCN swarm cluster - a simplified illustration of the architecture.

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`. An overview of repository images can be seen here.

Note: For the sake of simplicity, the internal private registry is not using the SSL encryption. Therefore the docker daemon needs to be instructed to trust the registry. It can be done by adding `insecure-registries` in `/etc/docker/daemon.json`. For example,

```
{
    "insecure-registries": ["docker-registry.dccn.nl:5000"],
    "storage-driver": "devicemapper",
    "storage-opts": [
         "dm.thinpooldev=/dev/mapper/docker-thinpool",
         "dm.use_deferred_removal=true",
         "dm.use_deferred_deletion=true"
    ]
}
```

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
        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 3

Swarm cluster operation procedures

Terminologies

- **cluster** is a group of docker-engine-enabled nodes (bare-matel or virtual machines). Each node has either a **master** or **worker** role in the cluster. At least one master node is required for a cluster to operate.
- **master** refers to the node maintaining the state of the cluster. There can be one or more masters in a cluster. The more masters in the cluster, the higher level of the cluster fault-tolerance.
- **worker** refers to the node sharing the workload in the cluster.
- **(micro-)service** is a logical representation of multiple replicas of the same container. Replicas are used for service load-balancing and/or failover.
- **stack** is a set of linked **services**.

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
```

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.

Node operation

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/docker/scripts/node-management/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
```

```
17
18  # install/update the LVM package
19  yum install -y lvm2
20
21  # create a physical volume on device
22  pvcreate ${device}
23
24  # create a volume group called 'docker'
25  vgcreate docker ${device}
26
27  # create logical volumes within the 'docker' volume group: one for data, one␣
    ↪for metadate
28  # assign volume size with respect to the size of the volume group
29  lvcreate --wipesignatures y -n thinpool docker -l 95%VG
30  lvcreate --wipesignatures y -n thinpoolmeta docker -l 1%VG
31  lvconvert -y --zero n -c 512K --thinpool docker/thinpool --poolmetadata␣
    ↪docker/thinpoolmeta
32
33  # update the lvm profile for volume autoextend
34  cat >/etc/lvm/profile/docker-thinpool.profile <<EOL
35  activation {
36      thin_pool_autoextend_threshold=80
37      thin_pool_autoextend_percent=20
38  }
39  EOL
40
41  # apply lvm profile
42  lvchange --metadataprofile docker-thinpool docker/thinpool
43
44  lvs -o+seg_monitor
45
46  # create daemon.json file to instruct docker using the created logical␣
    ↪volumes
47  cat >/etc/docker/daemon.json <<EOL
48  {
49      "hosts": ["unix:///var/run/docker.sock", "tcp://0.0.0.0:2375"],
50      "insecure-registries": ["docker-registry.dccn.nl:5000"],
51      "storage-driver": "devicemapper",
52      "storage-opts": [
53           "dm.thinpooldev=/dev/mapper/docker-thinpool",
54           "dm.use_deferred_removal=true",
55           "dm.use_deferred_deletion=true"
56      ]
57  }
58  EOL
59
60  # remove legacy deamon configuration through docker.service.d to avoid␣
    ↪confliction with daemon.json
61  if [ -f /etc/systemd/system/docker.service.d/swarm.conf ]; then
62      mv /etc/systemd/system/docker.service.d/swarm.conf /etc/systemd/system/
    ↪docker.service.d/swarm.conf.bk
63  fi
64
65  # reload daemon configuration
66  systemctl daemon-reload
```

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.

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.

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.

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>
```

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>
```

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.

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 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" \
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.

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.

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.

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>
```

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.

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 stack deploy -c docker-compose.yml <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`.

Remove a stack

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

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

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>
```

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

Reboot from shutdown

Note: By the accidental network outage in August 2017 (Domain Controller upgrade), 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.

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

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 4

Docker swarm health monitoring

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

CHAPTER 5

Service development

This document will walk you though few steps to build and run a WordPress application in the docker swarm cluster. The WordPress application consists of two service components:

- The WordExpress web application hosted in a Apache HTTP server
- MySQL database

For each of the two services, we will build a corresponding Docker container.

Write Dockerfile

Every docker container is built on top a basic container image, the OS container. Almost all Linux distributions have their mainstream systems published as container images on Docker Hub.

Build image

Upload image to registry

Deploy service