Container-based Virtualization: Esempi

This tutorial requires three Linux hosts with Docker installed and can communicate over a network creating a Docker swarm. These can be physical machines, virtual machines, or hosted in some other way. One of these machines will be the swarm manager (called manager1) and the other two the swarm workers (worker1 and worker2).

1. Install Docker

Firstly, you need to install Docker on all of the 3 host machines (let's assume the names equal to *dockertest1*, *dockertest2*, and *dockertest3*):

```
apt-get update
apt-get install apt-transport-https ca-certificates curl gnupg-agent sof
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key a
```

Add Docker repository (assume x86_64 architecture):

```
add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/u
```

Install Docker engine:

```
apt-get update
apt-get install docker-ce docker-ce-cli containerd.io
```

To manage docker as non-root user:

```
sudo groupadd docker
sudo usermod -aG docker $USER
```

Run the following to check if Docker was installed: `docker run hello-world` If all went well, you will see the following:

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest: sha256:f9dfddf63636d84ef479d645ab5885156ae030f611a56f3a7ac7f2fdd
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub
    (amd64)
 3. The Docker daemon created a new container from that image which runs
 4. The Docker daemon streamed that output to the Docker client, which s

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

The following ports must be available. On some systems, these ports are open by default.

- TCP port 2377 for cluster management communications
- TCP and UDP port 7946 for communication among nodes
- UDP port 4789 for overlay network traffic

The commands to open a port are:

- If you want to open an incoming TCP port, type the following:

```
iptables -I INPUT -p tcp --dport 12345 --syn -j ACCEPT
```

- If you want to open a UDP port (perhaps for DHT in Tixati), type the following:

```
iptables -I INPUT -p udp --dport 12345 -j ACCEPT
```

So,

```
iptables -I INPUT -p tcp --dport 2377 --syn -j ACCEPT
iptables -I INPUT -p tcp --dport 7946 --syn -j ACCEPT
iptables -I INPUT -p udp --dport 7946 -j ACCEPT
iptables -I INPUT -p udp --dport 4789 -j ACCEPT
```

Anyway, you could also open all ports (bad option :D):

```
iptables -F
```

2. Create the swarm

Let's assume we have 3 hosts named *dockertest1*, *dockertest2*, and *dockertest3* with the following private IPs:

- *dockertest1*: 192.168.100.101
- *dockertest2*: 192.168.100.102
- *dockertest3*: 192.168.100.103

Assume that *dockertest1* will be the manager node and *dockertest2* and *dockertest3* the worker nodes.

Run:

```
root@dockertest1:~# docker swarm init --advertise-addr 192.168.100.101
Swarm initialized: current node (skj4v2cjmqw4ymh39yckr93x8) is now a mana

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

    docker swarm join --token SWMTKN-1-474qylwts63dqamyzf4g8dipew73t4uha

To add a manager to this swarm, run 'docker swarm join-token manager' an

root@dockertest1:~#
```

After that, you can check if the *dockertest1* node is actually the manager:

```
root@dockertest1:~# docker info|grep "Is Manager"
  Is Manager: true
```

You can list nodes in the swarm by `docker node ls` command:

```
root@dockertest1:~# docker node ls
ID                            HOSTNAME            STATUS              AV
skj4v2cjmqw4ymh39yckr93x8 *   dockertest1         Ready               Ac
root@dockertest1:~#
```

The * next to the node ID indicates that we are on this node Docker Engine swarm mode automatically names the node for the machine hostname.

Now we can join the swarm (manager) by running the following command on *dockertest2* and *dockertest3* nodes, using the token generated on the manager node:

```
root@dockertest2:~# docker swarm join --token SWMTKN-1-474qylwts63dqamyz
This node joined a swarm as a worker.
root@dockertest2:~#
```

```
root@dockertest3:~# docker swarm join --token SWMTKN-1-474qylwts63dqamyz
This node joined a swarm as a worker.
root@dockertest3:~#
```

We can also retrieve info about how to join the swarm by running the following on the manager node:

```
root@dockertest1:~# docker swarm join-token worker
To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-474qylwts63dqamyzf4g8dipew73t4uha

root@dockertest1:~# docker swarm join-token manager
```

Now from the manager node, we can check the swarm status:

```
root@dockertest1:~# docker node ls
ID                            HOSTNAME            STATUS              AV
skj4v2cjmqw4ymh39yckr93x8 *   dockertest1         Ready               Ac
2rckhcout64izd59zz8qqehit     dockertest2         Ready               Ac
ypznkanigilfgznqvj4u6meu7     dockertest3         Ready               Ac
root@dockertest1:~#
```

3. Deploy and scale service

Start a service from the manager node. We specify the helloworld service, with alpine image, that performs the ping to www.google.it:

```
docker service create --replicas 1 --name helloworld alpine ping www.goo
```

To inspect the service details:

```
root@dockertest1:~# docker service inspect --pretty helloworld

ID:     w1gvhq4hfycf278cx1a2qxg8h
Name:       helloworld
Service Mode:   Replicated
 Replicas:  1
Placement:
UpdateConfig:
 Parallelism:   1
 On failure:    pause
 Monitoring Period: 5s
 Max failure ratio: 0
 Update order:      stop-first
RollbackConfig:
 Parallelism:   1
 On failure:    pause
 Monitoring Period: 5s
 Max failure ratio: 0
 Rollback order:    stop-first
ContainerSpec:
 Image:     alpine:latest@sha256:b276d875eeed9c7d3f1cfa7edb06b22ed22b142
 Args:      ping docker.com
 Init:      false
Resources:
Endpoint Mode:  vip

root@dockertest1:~#
```

To check instances of the service, run the following:

```
root@dockertest1:~# docker service ps helloworld
ID                  NAME                IMAGE               NODE
lv8vp2ttjhjs        helloworld.1        alpine:latest       dockertest2
root@dockertest1:~#
```

You can notice that the instance (container) of the service *helloworld* is running on *dockertest2* in this case. This is confirmed by running `docker ps` on node *dockertest2*:

```
root@dockertest2:~# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED
b66a160d0ce3        alpine:latest       "ping docker.com"   6 days ago
root@dockertest2:~#
```

4. Deploy Flask application

The Flask application to be deployed is a simply application with one REST API (default

route /), which informs who is replying to HTTP requests. You will deploy the Flask application as a service with 3 replicas managed automatically by Docker Swarm. To deploy the service, you firstly need to create the proper Docker image, by copying *dockerized*flask*service* dir in all the machines in the testbed. Then, you need to run the following on all the nodes (manager and workers):

```
# cd ~/dockerized_flask_service/app/
# docker build -t flask_image_hello_world .
```

To check if the image was built properly, run:

```
root@dockertest1:~# docker images |grep flask_image_hello_world
flask_image_hello_world              latest        fc27446c3f30   20 hou
root@dockertest1:~#
```

Now, you can deploy the service by running:

```
docker service create --replicas 3 --name flask_helloworld_service --pub
```

and check the deployment:

```
root@dockertest1:~# docker service ps flask_helloworld_service
ID             NAME                          IMAGE
vzfee3x6pizh   flask_helloworld_service.1    flask_image_hello_world:late
1rmmmxmx24q3   flask_helloworld_service.2    flask_image_hello_world:late
6cey6sqthp9t   flask_helloworld_service.3    flask_image_hello_world:late
root@dockertest1:~#
```

You can see that is running an instance of flask*helloworld*service service (the Flask application) on each node in the testbed. By running the following command `while True; do curl http://192.168.100.101:5001/; echo; sleep 1; done` we make HTTP requests only towards the manager node (dockertest1: 192.168.100.101:5001). We can observe that Docker Swarm automatically balances requests towards service replicas.

```
# while True; do curl http://192.168.100.101:5001/; echo; sleep 1; done
This is an example Flask app served from 20d293bc9fb8 to 10.0.0.2
This is an example Flask app served from 7e9d169241d3 to 10.0.0.2
This is an example Flask app served from 66b6d787a2cd to 10.0.0.2
This is an example Flask app served from 20d293bc9fb8 to 10.0.0.2
. . .
#
```

To test high availability, you can update the status of some worker nodes. Docker Swarm

allows you to DRAIN a node and prevent that node from receiving new tasks from the swarm manager. It also means the manager stops tasks running on the node and launches replica tasks on a node with ACTIVE availability.

To DRAIN the *dockertest2* node:

```
root@dockertest1:~# docker node update --availability drain dockertest2

root@dockertest1:~# docker node ls
ID                            HOSTNAME            STATUS              AV
skj4v2cjmqw4ymh39yckr93x8 *   dockertest1         Ready               Ac
2rckhcout64izd59zz8qqehit     dockertest2         Ready               Dr
ypznkanigilfgznqvj4u6meu7     dockertest3         Ready               Ac
root@dockertest1:~/nginx_test#
```

Check flask container "Exited" status on *dockertest2*:

```
root@dockertest2:~# docker ps -a
CONTAINER ID        IMAGE                               COMMAND
7e9d169241d3        flask_image_hello_world:latest   "python3 flask_app.
root@dockertest2:~#
```

The Swarm manager reschedules the instance on other nodes in the swarm. To check this run on manager node:

```
root@dockertest1:~# docker service ps flask_helloworld_service
ID             NAME                                IMAGE
vzfee3x6pizh   flask_helloworld_service.1       flask_image_hello_world:
u7aqz355pa0h   flask_helloworld_service.2       flask_image_hello_world:
1rmmmxmx24q3    \_ flask_helloworld_service.2   flask_image_hello_world:
6cey6sqthp9t   flask_helloworld_service.3       flask_image_hello_world:
root@dockertest1:~#
```

The *flask*helloworld*service.2* is in a Shutdown state on *dockertest2* node and it is in a *Running* state on *dockertest1* node (the first available in the swarm). In the meanwhile, the service availability is kept, and the Swarm manager keeps the desired state (3 running instances). Indeed, by running again *test*nginx.sh_ script, you can notice that there are still 3 replicas responses:

```
# while True; do curl http://192.168.100.101:5001/; echo; sleep 1; done
This is an example Flask app served from 66b6d787a2cd to 10.0.0.2
This is an example Flask app served from b1701656b1b6 to 10.0.0.2
This is an example Flask app served from 20d293bc9fb8 to 10.0.0.2
This is an example Flask app served from 66b6d787a2cd to 10.0.0.2
```

You can restore to available state the *dockertest2* node by running:

```
root@dockertest1:~# docker node update --availability active dockertest2
```

In that case, as soon as a task terminates or fails, the swarm manager reschedules another task on the *dockertest2* node.

5. Delete the swarm

In order to remove the swarm, you need to remove each worker node and the master from the swarm itself, by using `docker swarm leave`. Note that you need to specify the `--force` flag when you run the command within the master node. E.g.:

```
root@dockertest1:~# docker swarm leave --force
Node left the swarm.
```

6. Deploy HA service with `docker stack`

When running Docker Engine in swarm mode, we can run `docker stack deploy` command to deploy a complete application stack to the swarm. The deploy command accepts a stack description in the form of a Compose file. Compose files (.yml) used in the following examples specify the behavior for the swarm. In particular, let's check the next snippet:

```
...
deploy:
     replicas: 5
     restart_policy:
       condition: on-failure
       max_attempts: 3
       window: 120s
...
```

We can notice:

- `deploy` : specify configuration related to the deployment and running of services. This only takes effect when deploying to a swarm with docker stack deploy, and is ignored by docker-compose up and docker-compose run.
- `replicas` : If the service is replicated (which is the default), specify the number of containers that should be running at any given time.
- `restart_policy` : Configures if and how to restart containers when they exit. Replaces restart.
    - `condition` : One of `none` , `on-failure` (non-zero exit code) or `any`

(always restart the container if it stops) (default: any).

- `delay` : How long to wait between restart attempts, specified as a duration (default: 5s).
- `max_attempts` : How many times to attempt to restart a container before giving up (default: never give up). If the restart does not succeed within the configured window, this attempt doesn't count toward the configured max*attempts value. For example, if max*attempts is set to '2', and the restart fails on the first attempt, more than two restarts may be attempted.
- `window` : How long to wait before deciding if a restart has succeeded, specified as a duration (default: decide immediately).

Flask hello world example

Check compose_with_stack.yaml compose file that drives the master node to deploy properly the service.

To deploy the stack:

```
# // On all worker nodes
# docker build -t /PATH/TO/flask_image_hello_world_DOCKERFILE
# // On master node
# docker stack deploy --compose-file=/PATH/TO/compose_with_stack.yaml doc
Creating network dockerized_flask_service_replicated_default
Creating service dockerized_flask_service_replicated_web
```

To remove the stack:

```
# docker stack rm dockerized_flask_service_with_stack
Removing service dockerized_flask_service_replicated_web
Removing network dockerized_flask_service_replicated_default
```