Clustering with Docker Swarm

Raúl Estrada

Octubre 2020

Docker Swarm in practice

Docker Engine includes the Swarm mode by default, so there is no additional installation process required. Since Docker Swarm is a native Docker clustering system, managing cluster nodes is done by the `docker` command and is therefore very simple and intuitive. Let's start by creating a manager node with two worker nodes. Then, we will run and scale a service from a Docker image.

Setting up a Swarm

In order to set up a Swarm, we need to initialize the manager node. We can do this using the following command on a machine that is supposed to become the manager:

```
$ docker swarm init

Swarm initialized: current node (qfqzhk2bumhd2h0ckntrysm8l) is now a
manager.

To add a worker to this swarm, run the following command:
docker swarm join \
--token SWMTKN-1-253vezc1pqqgb93c5huc9g3n0hj4p7xik1ziz5c4rsdo3f7iw2-
df098e2jpe8uvwe2ohhhcxd6w \
192.168.0.143:2377

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

In our case, the manager machine has the IP address `192.168.0.143` and, obviously, it has to be reachable from the worker nodes (and vice versa). Note that the command to execute on worker machines was printed to the console. Also note that a special token has been generated. From now on, it will be used to connect a machine to the cluster and should be kept secret.

We can check that the Swarm has been created using the `docker node` command:

```
$ docker node ls
ID                            HOSTNAME        STATUS  AVAILABILITY  MANAGER
STATUS
qfqzhk2bumhd2h0ckntrysm8l *  ubuntu-manager  Ready   Active        Leader
```

When the manager is up and running, we are ready to add worker nodes to the Swarm.

Adding worker nodes

In order to add a machine to the Swarm, we have to log in to the given machine and execute the following command:

```
$ docker swarm join \
--token SWMTKN-1-253vezc1pqqgb93c5huc9g3n0hj4p7xik1ziz5c4rsdo3f7iw2-
df098e2jpe8uvwe2ohhhcxd6w \
192.168.0.143:2377

This node joined a swarm as a worker.
```

We can check that the node has been added to the Swarm with the `docker node ls` command. Assuming that we've added two node machines, the output should look as follows:

```
$ docker node ls
ID                            HOSTNAME        STATUS  AVAILABILITY  MANAGER
STATUS
cr7vin5xzu0331fvxkdxla22n     ubuntu-worker2  Ready   Active
md4wx15t87nn0c3pyv24kewtz     ubuntu-worker1  Ready   Active
qfqzhk2bumhd2h0ckntrysm81 *   ubuntu-manager  Ready   Active        Leader
```

Deploying a service

In order to run an image on a cluster, we don't use `docker run` but the Swarm-dedicated `docker service` command (which is executed on the manager node). Let's start a single `tomcat` application and give it the name `tomcat`:

```
$ docker service create --replicas 1 --name tomcat tomcat
```

The command created the service and therefore sent a task to start a container on one of the nodes. Let's list the running services:

```
$ docker service ls
ID            NAME    MODE         REPLICAS   IMAGE
x65aeojumj05  tomcat  replicated   1/1        tomcat:latest
```

The log confirms that the `tomcat` service is running, and it has one replica (one Docker container is running). We can examine the service even more closely:

```
$ docker service ps tomcat
ID            NAME      IMAGE          NODE             DESIRED STATE CURRENT
STATE
kjy1udwcnwmi tomcat.1  tomcat:latest  ubuntu-manager  Running     Running
about a minute ago
```

If you are interested in the detailed information about a service, you can use the `docker service inspect <service_name>` command.

From the console output, we can see that the container is running on the manager node (`ubuntu-manager`). It could have been started on any other node as well; the manager automatically chooses the worker node using the scheduling strategy algorithm. We can confirm that the container is running using the well-known `docker ps` command:

```
$ docker ps
CONTAINER ID      IMAGE
COMMAND             CREATED              STATUS               PORTS
NAMES
6718d0bcba98
tomcat@sha256:88483873b279aaea5ced002c98dde04555584b66de29797a4476d5e94874e
6de
"catalina.sh run" About a minute ago Up About a minute   8080/tcp
tomcat.1.kjy1udwcnwmiosiw2qn71nt1r
```

Scaling service

When the service is running, we can scale it up or down so that it will be running in many replicas:

```
$ docker service scale tomcat=5
tomcat scaled to 5
```

We can check that the service has been scaled:

```
$ docker service ps tomcat
ID             NAME      IMAGE           NODE             DESIRED STATE
CURRENT STATE
kjy1udwcnwmi   tomcat.1  tomcat:latest   ubuntu-manager   Running    Running 2
minutes ago
536p5zc3kaxz   tomcat.2  tomcat:latest   ubuntu-worker2   Running    Preparing
18 seconds ago npt6ui1g9bdp  tomcat.3  tomcat:latest   ubuntu-manager
Running    Running 18 seconds ago zo2kger1rmqc  tomcat.4  tomcat:latest
ubuntu-worker1  Running    Preparing 18 seconds ago 1fb24nf94488  tomcat.5
tomcat:latest   ubuntu-worker2  Running    Preparing 18 seconds ago
```

Note that this time, two containers are running on the `manager` node, one on the `ubuntu-worker1` node, and the other on the `ubuntu-worker2` node. We can check that they are really running by executing `docker ps` on each of the machines.

If we want to remove the services, it's enough to execute the following command:

```
$ docker service rm tomcat
```

You can check with the `docker service ls` command that the service has been removed, and therefore all related `tomcat` containers were stopped and removed from all the nodes.

Publishing ports

Docker services, similar to the containers, have a port forwarding mechanism. We use it by adding the `-p <host_port>:<container:port>` parameter. Here's what starting a service could look like:

```
$ docker service create --replicas 1 --publish 8080:8080 --name tomcat
tomcat
```

Now, we can open a browser and see the Tomcat's main page under the address `http://192.168.0.143:8080/`.

The Docker Swarm rolling update process looks as follows:

1. Stop the `<update-parallelism>` number of tasks (replicas).
2. In their place, run the same number of updated tasks.
3. If a task returns the **RUNNING** state, then wait for the `<update-delay>` period.
4. If, at any time, any task returns the **FAILED** state, then pause the update.

Let's look at an example and change the Tomcat application from version 8 to version 9. Suppose we have the `tomcat:8` service with five replicas:

```
$ docker service create --replicas 5 --name tomcat --update-delay 10s
tomcat:8
```

We can check that all replicas are running with the `docker service ps tomcat` command. Another useful command that helps examine the service is the `docker service inspect` command:

```
$ docker service inspect --pretty tomcat

ID:     au1nu396jzdewyq2y8enm0b6i
Name:     tomcat
Service Mode:     Replicated
  Replicas:     5
Placement:
UpdateConfig:
  Parallelism:     1
  Delay:     10s
  On failure:     pause
  Max failure ratio: 0
ContainerSpec:
  Image:
tomcat:8@sha256:835b6501c150de39d2b12569fd8124eaebc53a899e2540549b6b6f86765
38484
Resources:
Endpoint Mode:     vip
```

We can see that the service has five replicas created out of the image `tomcat:8`. The command output also includes the information about the parallelism and the delay time between updates (as set by the options in the `docker service create` command).

Now, we can update the service into the `tomcat:9` image:

```
$ docker service update --image tomcat:9 tomcat
```

Let's check what happens:

```
$ docker service ps tomcat
ID            NAME      IMAGE     NODE            DESIRED STATE  CURRENT
STATE
4dvh6ytn4lsq  tomcat.1  tomcat:8  ubuntu-manager  Running        Running 4
minutes ago
2mop96j5q4aj  tomcat.2  tomcat:8  ubuntu-manager  Running        Running 4
minutes ago
owurmusr1c48  tomcat.3  tomcat:9  ubuntu-manager  Running        Preparing 13
seconds ago
r9drfjpizuxf   \_ tomcat.3  tomcat:8  ubuntu-manager  Shutdown   Shutdown
12 seconds ago
0725ha5d8p4v  tomcat.4  tomcat:8  ubuntu-manager  Running        Running 4
minutes ago
wl25m2vrqgc4  tomcat.5  tomcat:8  ubuntu-manager  Running        Running 4
minutes ago
```

Note that the first replica of `tomcat:8` has been shut down and the first `tomcat:9` is already running. If we kept on checking the output of the `docker service ps tomcat` command, we would notice that after every 10 seconds, another replica is in the shutdown state and a new one is started. If we also monitored the `docker inspect` command, we would see that the value **UpdateStatus: State** will change to **updating** and then, when the update is done, to **completed**.

A rolling update is a very powerful feature that allows zero downtime deployment and it should always be used in the Continuous Delivery process.

Let's look how this works in practice. Suppose we have three cluster nodes and a Tomcat service with five replicas:

```
$ docker node ls
ID                           HOSTNAME         STATUS   AVAILABILITY   MANAGER
STATUS
4mrrmibdrpa3yethhmy13mwzq    ubuntu-worker2   Ready    Active
kzgm7erw73tu2rjjninxdb4wp *  ubuntu-manager   Ready    Active         Leader

yllusy42jp08w8fmze43rmqqs    ubuntu-worker1   Ready    Active

$ docker service create --replicas 5 --name tomcat tomcat
```

Let's check on which nodes the replicas are running:

```
$ docker service ps tomcat
ID             NAME      IMAGE          NODE             DESIRED STATE
CURRENT STATE
zrnawwpupuql   tomcat.1  tomcat:latest  ubuntu-manager   Running    Running
17 minutes ago
x6rqhyn7mrot   tomcat.2  tomcat:latest  ubuntu-worker1   Running    Running
16 minutes ago
rspgxcfv3is2   tomcat.3  tomcat:latest  ubuntu-worker2   Running    Running 5
weeks ago
cf00k61vo7xh   tomcat.4  tomcat:latest  ubuntu-manager   Running    Running
17 minutes ago
otjo08e06qbx   tomcat.5  tomcat:latest  ubuntu-worker2   Running    Running 5
weeks ago
```

There are two replicas running on the `ubuntu-worker2` node. Let's drain that node:

```
$ docker node update --availability drain ubuntu-worker2
```

The node is put into the **drain** availability, so all replicas should be moved out of it:

```
$ docker service ps tomcat
ID             NAME      IMAGE          NODE             DESIRED STATE
CURRENT STATE
zrnawwpupuql  tomcat.1  tomcat:latest  ubuntu-manager  Running    Running
18 minutes ago
x6rqhyn7mrot  tomcat.2  tomcat:latest  ubuntu-worker1  Running    Running
17 minutes ago qrptjztd777i  tomcat.3  tomcat:latest  ubuntu-worker1
Running    Running less than a second ago
rspgxcfv3is2   \_ tomcat.3  tomcat:latest  ubuntu-worker2  Shutdown
Shutdown less than a second ago
cf00k61vo7xh  tomcat.4  tomcat:latest  ubuntu-manager  Running    Running
18 minutes ago k4c14tyo7leq  tomcat.5  tomcat:latest  ubuntu-worker1
Running    Running less than a second ago
otjo08e06qbx   \_ tomcat.5  tomcat:latest  ubuntu-worker2  Shutdown
Shutdown less than a second ago
```

We can see that new tasks were started on the `ubuntu-worker1` node and the old replicas were shut down. We can check the status of the nodes:

```
$ docker node ls
ID                            HOSTNAME         STATUS   AVAILABILITY   MANAGER
STATUS
4mrrmibdrpa3yethhmy13mwzq     ubuntu-worker2   Ready    Drain
kzgm7erw73tu2rjjninxdb4wp *   ubuntu-manager   Ready    Active         Leader
yllusy42jp08w8fmze43rmqqs     ubuntu-worker1   Ready    Active
```

As expected, the `ubuntu-worker2` node is available (status `Ready`), but its availability is set to drain, which means it doesn't host any tasks. If we would like to get the node back, we can check its availability to `active`:

```
$ docker node update --availability active ubuntu-worker2
```

A very common practice is to drain the manager node and, as a result, it will not receive any tasks, but do management work only.

An alternative method to draining the node would be to execute the `docker swarm leave` command from the worker. This approach, however, has two drawbacks:

- For a moment, there are fewer replicas than expected (after leaving the swarm and before the master starts new tasks on other nodes)
- The master does not control if the node is still in the cluster

For these reasons, if we plan to stop the worker for some time and then get it back, it's recommended to use the draining node feature.