Instituto Superior Técnico, Universidade de Lisboa
Large-Scale Systems Engineering

Laboratory Guide – Part II: Large Scale Deployments and System Analysis

Goals

The previous laboratory guide focused on the essential tools to deploy, measure, and plot the behaviour of a simple system. This part builds on the previous one and focus on tools to deploy and manage large scale systems. We will use these tools to reason about the behaviour of a simple application.

1. Docker Swarm

Docker packs applications in containers that are easy to build, deploy and distribute. This works well for few containers, but it becomes impractical to manage applications composed of several services (for instance, a database, an application server, and the frontend) deployed over dozens or thousands of containers.

Docker Swarm addresses this problem by specifying distributed applications as a series of composable *services* and providing tools for cluster management (for instance which containers to allocate to which physical machines), controlling the number of replicas of a service, automatically restarting replicas upon failures, load-balancing and finer control over the network. A service is a set of running containers, known as *tasks*.

There are other popular alternatives to Docker Swarm, namely Kubernetes. In this laboratory, we will focus on Docker Swarm as it is simple and comes bundled with Docker but students are encouraged to explore Kubernetes features (which is more powerful).

As in the previous guide, in the examples below we will use the notation

```
(host)$ <command>
```

to indicate a command should be run on the host (your machine) and

```
(container)$ <command>
```

to indicate a command should be run on the container.

Docker Swarm orchestrates a cluster of physical machines that can be used to deploy containers. The cluster has a manager, which is a machine responsible for issuing commands and managing the status of the cluster, and *workers* which are machines that will be used by the manager to deploy containers. The manager also acts as a worker allowing to have Docker Swarm cluster with a single machine. This is useful for development and testing.

To start a Docker Swarm, determine the IP of the manager machine (your computer) and initialize the Swarm with:

```
(host)$ docker swarm init --advertise-addr <HOST-IP>
```

For simplicity, we will use a single physical node in the laboratory guide, but students are encouraged to try a deployment with several workers. Workers can be other physical machines, virtual machines, or a cloud environment. For a worker to join the Swarm, run the following on the worker, where TOKEN is a string reported when initializing the Swarm in the manager, and HOST_IP:PORT is the address of the manager:

```
(worker host)$ docker swarm join –token <TOKEN> <HOST_IP:PORT>
```

To view information about the current state of the Swarm:

```
(host)$ docker info
```

This shows a detailed status of the cluster. The most relevant information for us is:

```
Containers: 0
  Running: 0
  Paused: 0
  Stopped: 0
Images: 50
```

which shows a status of the containers running in the cluster and the available images.
To view the status of the cluster:

```
(host)$ docker node ls

ID        HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
<ID>  *  <HOSTNAME>     Ready   Active        Leader
```

which shows information about the different nodes in the Swarm cluster. The manager node should have the *Leader* status and the worker nodes that joined with an empty status. The asterisk (*) indicates the node where the command was run.

To create a Swarm service:

```
(host)$ docker service create --replicas 1 --name helloworld
mycontainer:latest ping tecnico.ulisboa.pt
```

Note that this is very similar to the *docker run* command of the previous laboratory. It will create a service named *helloworld* with one replica from the image *mycontainer:latest* created in the previous laboratory and execute the command *ping tecnico.ulisboa.pt* inside each container of the service.

To check the running services:

```
(host)$ docker service ls
```

which shows

```
ID                  NAME        MODE        REPLICAS  IMAGE
kzcfsuwdrmt9  helloworld  replicated  1/1       mycontainer:latest
```

the status of the running services.

We can obtain detailed information about a given service with:

```
(host)$ docker service inspect --pretty <SERVICE-ID>
```

To verify the machines where the containers are running:

```
(host)$ docker service ps helloworld
```

To further check the status of the container run, from the machine where the container is hosted:

```
(host)$ docker ps
```

which shows the same information we saw in the last laboratory.

To check the logs (stdout) of the service use:

```
(host)$ docker service logs <SERVICE-ID>
```

Up to this point, we could have achieved the same with the regular *docker run* studied in the previous laboratory. One of the advantages of Swarm is that it allows to adjust the service configurations dynamically and automatically.
To add additional tasks (running containers) to the *helloworld* service:

```
(host)$ docker service scale helloworld=5
```

which will create four new replicas of the service, for a total of five. Check the status of the service with *docker service inspect* and the status of the containers with *docker ps*. The same command can be used to reduce the number of replicas.

To remove the *helloworld* service:

```
(host)$ docker service rm helloworld
```

Services can take some time to be removed, in particular for a large number of replicas spread across several machines. You can check the progress of this operation with *docker ps*.

Volumes

The data that is written in a container is stored in a layered filesystem. This is useful for versioning but has a non-negligible cost, in particular for large files, or files that change very often such as log files. Also, a container cannot, by default, access files in the host filesystem. These limitations can be overcome by mounting a volume in the container or service which will map to a directory in the host. To create a new volume:

```
(host)$ docker volume create esle-vol
```

To check the details of the volume just created, including the mapping to the local filesystem:

```
(host)$ docker volume inspect esle-vol
```

To create a container with an associated volume:

```
(host)$ docker run -it -v esle-vol:/data mycontainer:latest sh
```

This will map the volume *esle-vol,* created before to the */data* directory inside the container. Create a file in the */data* directory inside the container and shutdown the container. Start another container with the *else-vol* mounted and confirm that you can observe the file created before.

To create a service with an associated volume:

```
(host)$ docker service create --replicas 1 --name helloworld --mount
source=esle-vol,target=/data <IMAGE_NAME>
```

This will result in every container (task) inside the service to have access to the */data* volume. Note that in recent versions of Docker you can also use, with *docker run* the *--mount* switch instead of the *–v*.

Sometimes it is useful to execute a command in an already running container for debugging purposes. For instance, to run bash on an existing container:

```
(host)$ docker exec -it <CONTAINER_ID> bash
```

Network

Docker Swarm service can be attached to a (virtual) network which allows for isolation between services, load balancing and finer grained control over the network configuration. To create a network

```
(host)$ docker network create --driver overlay esle-network
```

this will create a network named *else-network* with the overlay driver. There are other drivers available but discussing them is out of the scope of this guide. Students are encouraged to investigate the other drivers available.

To check the available Docker networks:

```
(host)$ docker network ls
```

this will show, along with the *esle-network* just created other networks that are automatically created by Docker. To create a service, and attach it to a network:

```
(host)$ docker service create --name web-service --publish
published=8080,target=80 --replicas 2  --network esle-network nginx:latest
```

this will create a service with *two* replicas of *the nginx:latest* image and attach it to the *esle-network*. Port *80* in the service containers will be mapped to the service port *8080*. Docker Swarm automatically maps port 80, on every worker to the

service port 80. Requests done in the host port are automatically redirected to a random service task (containers), acting as a simple load balancer.

Using curl, check the following:

- Access the service using the redirected port on the host
- Access the service by using directly the IP of one of the containers
- Create a container with Docker run, and try accessing the nginx server with curl

Are the results expected? Why?

2. Putting it all together

The goal is to extend the nginx evaluation of the previous laboratory. To this end:

- Create an nginx service with one replica
- Create a siege service with one replica using the same parameters of the previous laboratory (one client, one minute run, delay of 1 second) and log the results to a logfile in a volume. Recall the following:
  - To check the options of siege you can use the –help option.
  - To limit the CPU usage of a service you can use the *--limit-cpu* option.
  - Docker swarm by default assumes a replicated mode which automatically restarts containers that failed/terminated. For siege we do not want this. Use the option –mode replicated-job when creating the service.
  - Wait for the service to terminate, plot and analyse the results. Are they expected?
- Repeat the above experience by increasing the number of siege replicas to 2, 3 and 4. Plot and analyse the results. What did you observe?
- Now increase the nginx service to 1,2,3,4 replicas while using 4 siege replicas. Plot and analyse the results. What did you observe?

Bibliography

Docker documentation - https://docs.docker.com/get-started/

Docker Swarm - https://docs.docker.com/engine/swarm/