19 Creating a Swarm

In this exercise, we'll see how to set up a swarm using Docker Swarm Mode, including joining workers and promoting workers into the manager consensus.

19.1 Start Swarm Mode

1. Enable Swarm Mode on whatever node is to be your first manager node:

   ```
   $ docker swarm init
   ```

2. Confirm that Swarm Mode is active by inspecting the output of:

   ```
   $ docker info
   ```

3. See all nodes currently in your swarm by doing:

   ```
   $ docker node ls
   ```

   A single node is reported in the cluster.

19.2 Add Workers to the Swarm

A single node swarm is not a particularly interesting swarm; let's add some workers to really see Swarm Mode in action.

1. On your manager node, get the swarm 'join token' you'll use to add worker nodes to your swarm:

   ```
   $ docker swarm join-token worker
   ```

2. SSH into a second node, and paste the result of the last step there. This new node will have joined the swarm as a worker.
3. Do `docker node ls` on the manager again, and you should see both your nodes and their status; note that `docker node ls` won't work on a worker node, as the cluster status is maintained only by the manager nodes.
4. Finally, use the same join token to add two more workers to your swarm. When you're done, confirm that `docker node ls` on your one manager node reports 4 nodes in the cluster - one manager, and three workers.

19.3 Promoting Workers to Managers

At this point, our swarm has a single manager. If this node goes down, the whole Swarm is lost. In a real deployment, this is unacceptable; we need some redundancy to our system, and Swarm Mode achieves this by allowing a Raft consensus of multiple managers to preserve swarm state.

1. Promote two of your workers to manager status by executing, on the current manager node:

   ```
   $ docker node promote worker-0 worker-1
   ```

   where `worker-0` and `worker-1` are the hostnames of the two workers you want to promote to managerial status (look at the output of `docker node ls` if you're not sure what your hostnames are).

2. Finally, do a `docker node ls` to check and see that you now have three managers. Note that manager nodes also count as worker nodes - tasks can still be scheduled on them as normal.

19.4 Conclusion

In this exercise, you saw how to set up a swarm with with basic API objects `docker swarm init` and `docker swarm join`, as well as how to inspect the state of the swarm with `docker node ls` and `docker info`. Finally, you promoted worker nodes to the manager consensus with `docker node promote`.

20 Starting a Service

So far, we've set up a four-node swarm with three managers; in order to use a swarm to actually execute anything, we have to define *services* for that swarm to run; services are the fundamental logical entity that users interact with in a distributed application engineering environment, while things like individual processes or containers are handled by the swarm scheduler; similarly, the scheduler handles routing tasks to specific nodes, so the user can approach the swarm as a whole without explicitly interacting with individual nodes.

20.1 Start a Service

1. Create a service featuring an `alpine` container pinging Google resolvers:

   ```
   $ docker service create alpine ping 8.8.8.8
   ```

   Note the syntax is a lot like `docker container run`; an image (`alpine`) is specified, followed by the PID 1 process for that container (`ping 8.8.8.8`).
2. Get some information about the currently running services:

   ```
   $ docker service ls
   ```

3. Check which node the container was created on:

   ```
   $ docker service ps <service ID>
   ```

4. SSH into the node you found in the last step, find the container ID with `docker container ls`, and check its logs with `docker container logs <container ID>`. The results of the ongoing ping should be visible.

20.2 Scaling a Service

1. Scale up the number of concurrent tasks that our `alpine` service is running:

   ```
   $ docker service update <service name> --replicas=8
   ```

2. Now run `docker service ps <service name>` to inspect the service. Are all the containers running right away? How were they distributed across your swarm?

20.3 Cleanup

1. Remove all existing services, in preparation for future exercises:

   ```
   $ docker service rm $(docker service ls -q)
   ```

20.4 Conclusion

In this example, we saw the basic syntax for defining a service based on an image, and for changing the number of replicas, or concurrent containers, running of that image. We also saw how to investigate the state of services on our swarm with `docker service ls` and `docker service ps`.

21 Node Failure Recovery

In Swarm Mode, services created with `docker service create` are primary; if something goes wrong with the cluster, the manager leader does everything possible to restore the state of all services.

21.1 Set up a Service

1. Set up an `nginx` service with four replicas on your manager node:

   ```
   manager$ docker service create --replicas 4 --name nginx nginx
   ```

2. Now watch the output of `docker service ps` on the same node:

   ```
   manager$ watch docker service ps nginx
   ```

21.2 Simulate Node Failure

1. SSH into the one non-manager node in your swarm, and simulate a node failure by rebooting it:

   ```
   worker$ sudo reboot now
   ```

2. Back on your manager node, watch the updates to `docker service ps`; what happens to the task running on the rebooted node?

21.3 Cleanup

1. Remove all existing services, in preparation for future exercises:

   ```
   manager$ docker service rm $(docker service ls -q)
   ```

21.4 Conclusion

In this exercise, you saw Swarm Mode's scheduler in action - when a node is lost from the swarm, tasks are automatically rescheduled to restore the state of our services.

22 Load Balancing & the Routing Mesh

In this exercise, you will observe the behaviour of the built in load balancing abilities of the Ingress network.

22.1 Deploy a service

1. Start by deploying a simple service which spawns containers that echo back their hostname when `curl`'ed:

   ```
   $ docker service create --name who-am-I --publish 8000:8000 --replicas 3 training/whoami:latest
   ```

22.2 Observe load-balancing and Scale

1. Run `curl localhost:8000` and observe the output. You should see something similar to the following:

   ```
   $ curl localhost:8000
   I'm a7e5a21e6e26
   ```

   Take note of the response. In this example, our value is `a7e5a21e6e26`. The `whoami` containers uniquely identify themselves by returning their respective hostname. So each one of our `whoami` instances should have a different value.

2. Run `curl localhost:8000` again. What can you observe? Notice how the value changes each time. This shows us that the routing mesh has sent our 2nd request over to a different container, since the value was different.

3. Repeat the command two more times. What can you observe? You should see one new value and then on the 4th request it should revert back to the value of the first container. In this example that value is `a7e5a21e6e26`
4. Scale the number of Tasks for our `who-am-I` service to 6:

   ```
   $ docker service update who-am-I --replicas=6
   $ docker service ps who-am-I
   ```

5. Now run `curl localhost:8000` multiple times again. Use a script like this:

   ```
   $ for n in {1..10}; do
   >   curl localhost:8000
   > done;
   I'm 263fc24d0789
   I'm 57ca6c0c0eb1
   I'm c2ee8032c828
   I'm c20c1412f4ff
   I'm e6a88a30481a
   I'm 86e262733b1e
   I'm 263fc24d0789
   I'm 57ca6c0c0eb1
   I'm c2ee8032c828
   I'm c20c1412f4ff
   ```

   You should be able to observe some new values. Note how the values repeat after the 6th curl command.

22.3 The Routing Mesh

1. Run an nginx service and expose the service port 80 on port 8080:

   ```
   $ docker service create --name nginx --publish 8080:80 nginx
   ```

2. Check which node your nginx service task is scheduled on:

   ```
   $ docker service ps nginx
   ```

3. Open a web browser and hit the IP address of that node at port 8080. You should see the NGINX welcome page. Try the same thing with the IP address of any other node in your cluster (using port 8080). You should still be able to see the NGINX welcome page due to the routing mesh.

22.4 Cleanup

1. Remove all existing services, in preparation for future exercises:

   ```
   $ docker service rm $(docker service ls -q)
   ```

22.5 Conclusion

In these examples, you saw that requests to an exposed service will be automatically load balanced across all tasks providing that service. Furthermore, exposed services are reachable on all nodes in the swarm - whether they are running a container for that service or not.

23 Dockercoins On Swarm

In this example, we'll go through the preparation and deployment of a sample application, our dockercoins miner, on our swarm. We'll define our app using a `docker-compose.yml` file, and deploy is as our first example of a stack.

23.1 Prepare Service Images

1. If you haven't done so already, follow the 'Prepare Service Images' step in the 'Starting a Compose App' exercise in this book, on your manager node. In this step, you built all the images you need for your app and pushed them to Docker Store, so they'd be available for every node in your swarm.

23.2 Start our Services

1. Now that everything is prepped, we can start our stack. On the manager node:

   ```
   $ docker stack deploy -c=docker-compose.yml dc
   ```

2. Check and see how your services are doing:

   ```
   $ docker stack services dc
   ```

   Notice the `REPLICAS` column in the output of above command; this shows how many of your desired replicas are running. At first, a few might show 0/1; before those tasks can start, the worker nodes will need to download the appropriate images from Docker Store.

3. Wait a minute or two, and try `docker stack services dc` again; once all services show 100% of their replicas are up, things are running properly and you can point your browser to port 8000 on one of the swarm nodes (does it matter which one)? You should see a graph of your dockercoin mining speed, around 3 hashes per second.

4. Finally, check out the details of the tasks running in your stack with `stack ps`:

   ```
   $ docker stack ps dc
   ```

   This shows the details of each running container involved in your stack - if all is well, there should be five, one for each service in the stack.

24 Scaling and Scheduling Services

24.1 Scaling up a Service

If we've written our services to be stateless, we might hope for linear performance scaling in the number of replicas of that service. For example, our `worker` service requests a random number from the `rng` service and hands it off to the `hasher` service; the faster we make those requests, the higher our throughput of dockercoins should be, as long as there are no other confounding bottlenecks.

1. Modify the `worker` service definition in `docker-compose.yml` to set the number of replicas to create using the `deploy` key:

   ```
     worker:
       image: user/dockercoins_worker:1.0
       networks:
       - dockercoins
       deploy:
          replicas: 2
   ```

2. Update your app by running the same command you used to launch it in the first place:

   ```
   $ docker stack deploy -c=docker-compose.yml dc
   ```

3. Once both replicas of the `worker` service are live, check the web frontend; you should see about double the number of hashes per second, as expected.

4. Scale up even more by changing the `worker` replicas to 10. A small improvement should be visible, but certainly not an additional factor of 5. Something else is bottlenecking dockercoins.

24.2 Scheduling Services

Something other than `worker` is bottlenecking dockercoins's performance; the first place to look is in the services that `worker` directly interacts with.

1. The `rng` and `hasher` services are exposed on host ports 8001 and 8002, so we can use `httping` to probe their latency:

   ```
   $ httping -c 10 localhost:8001
   $ httping -c 10 localhost:8002
   ```

   `rng` is much slower to respond, suggesting that it might be the bottleneck. If this random number generator is based on an entropy collector (random voltage microfluctuations in the machine's power supply, for example), it won't be able to generate random numbers beyond a physically limited rate; we need more machines collecting more entropy in order to scale this up. This is a case where it makes sense to run exactly one copy of this service per machine, via `global` scheduling (as opposed to potentially many copies on one machine, or whatever the scheduler decides as in the default `replicated` scheduling).

2. Modify the definition of our `rng` service in `docker-compose.yml` to be globally scheduled:

   ```
     rng:
       image: user/dockercoins_rng:1.0
       networks:
       - dockercoins
       ports:
       - "8001:80"
       deploy:
         mode: global
   ```

3. Scheduling can't be changed on the fly, so we need to stop our app and restart it:

   ```
   $ docker stack rm dc
   $ docker stack deploy -c=docker-compose.yml dc
   ```

4. Check the web frontend again; the overall factor of 10 improvement (from ~3 to ~35 hashes per second) should now be visible.

24.3 Conclusion

In this exercise, you explored the performance gains a distributed application can enjoy by scaling a key service up to have more replicas, and by correctly scheduling a service that needs to be replicated across physically different nodes.

25 Updating a Service

25.1 Rolling Updates

1. First, let's change one of our services a bit: open `orchestration-workshop/dockercoins/worker/worker.py` in your favorite text editor, and find the following section:

   ```
   def work_once():
       log.debug("Doing one unit of work")
        time.sleep(0.1)
   ```

   Change the `0.1` to a `0.01`. Save the file, exit the text editor.

2. Rebuild the worker image with a tag of `<Docker Store username>/dockercoins_worker:1.1`, and push it to Docker Store.

3. Open a new ssh connection to your manager node, and set it to watch the following:

```
$ watch -n1 "docker service ps dc_worker | grep -v Shutdown.*Shutdown"
```

(this last step isn't necessary to do an update, but is just for illustrative purposes to help us watch the update in action).

4. Switch back to your original connection to your manager, and start the update:

```
$ docker service update dc_worker --image <user>/dockercoins_worker:1.1
```

5. Switch back to your new terminal and observe the output. You should notice the tasks images being updated to our new 1.1 image one at a time.

25.2 Parallel Updates

1. We can also set our updates to run in batches by configuring some options associated with each service. On your first connection to your manager, change the parallelism to 2 and the delay to 5 seconds on the `worker` service by editing its definition in the `docker-compose.yml`:

```
  worker:
    image: billmills/dockercoins_worker:1.0
    networks:
    - dockercoins
    deploy:
      replicas: 10
      update_config:
        parallelism: 2
        delay: 5s
```

2. Roll back the worker service to 1.0:

```
$ docker stack deploy -c=docker-compose.yml dc
```

3. On the second connection to manager, the `watch` should show instances being updated two at a time, with a five second pause between updates.

25.3 Shutting Down a Stack

1. To shut down a running stack:

```
$ docker stack rm <stack name>
```

Where the stack name can be found in the output of `docker stack ls`.

26 Docker Secrets

Docker Swarm mode offers tooling to manage secrets securely, ensuring they are never transmitted or stored unencrypted on disk. In this exercise, we will explore basic secret usage.

26.1 Creating Secrets

1. Create a new secret named `my-secret` with the value `some sensitive value` by using the following command to pipe STDIN to the secret value:

```
$ echo 'my sensitive value' | docker secret create my-secret -
```

2. Alternatively, secret values can be read from a file. In the current directory create a file called `mysql-password.txt` and add the value `1PaSsw0rd2` to it. Create a secret with this value:

```
$ docker secret create mysql-password ./mysql-password.txt
```

26.2 Managing Secrets

The Docker CLI provides API objects for managing secrets similar to all other Docker assets:

1. List your current secrets:

   ```
   $ docker secret ls
   ```

2. Print secret metadata:

   ```
   $ docker secret inspect <secret name>
   ```

3. Delete a secret (don't delete `my-secret` or `mysql-secret`, since we'll need them in the next step):

   ```
   $ docker secret rm <secret name>
   ```

26.3 Using Secrets

Secrets are assigned to Swarm services upon creation of the service, and provisioned to containers for that service as they spin up.

1. Create a service authorized to use the secrets `my-secret` and `mysql-password` secret.

   ```
   $ docker service create \
       --name demo \
       --secret my-secret \
       --secret mysql-password \
       alpine:latest ping 8.8.8.8
   ```

2. Use `docker service ps demo` to determine what node your service container is running on; ssh into that node, and connect to the container (remember to use `docker container ls` to find the container ID):

   ```
   $ docker container exec -it <container ID> sh
   ```

3. Inspect the secrets in this container where they are mounted, at `/run/secrets`:

   ```
   $ cd /run/secrets
   $ ls
   $ cat my-secret
   $ exit
   ```

   This is the *only* place secret values sit unencrypted in memory.

26.4 Updating a Secret

Secrets are never updated in-flight; secrets are added or removed, restarting the service each time.

1. Create a new version of the `my-secret` secret; add the `-v2` suffix just to distinguish it from the original secret, in case we need to roll back:

   ```
   $ echo 'updated value v2' | docker secret create my-secret-v2 -
   ```

2. Update our `demo` service first by deleting the old secret:

   ```
   $ docker service update --secret-rm my-secret demo
   ```

3. Assign the new value of the secret to the service, using `source` and `target` to alias the `my-secret-v2` outside the container as `my-secret` inside:

```
$ docker service update --secret-add source=my-secret-v2,target=my-secret demo
```

4. exec into the running container and demonstrate that the value of the `my-secret` secret has changed.

26.5 Preparing an image for use of secrets

Containers need to consume secrets per their mounting in `/run/secrets`. In many cases, existing application logic expects secret values to appear behind environment variables; in the following, we set up such a situation as an example.

1. Create a new directory `image-secrets` and navigate to this folder. In this folder create a file named `app.py` and add the following content:

```python
import os
print '***** DOCKER Secrets ******'
print 'USERNAME: {0}'.format(os.environ['USERNAME'])

fname = os.environ['PASSWORD_FILE']
with open(fname) as f:
    content = f.readlines()

print 'PASSWORD_FILE: {0}'.format(fname)
print 'PASSWORD: {0}'.format(content[0])
```

2. Create a file called `Dockerfile` with the following content:

```
FROM python:2.7
RUN mkdir -p /app
WORKDIR /app
COPY . /app
CMD python ./app.py && sleep 1000
```

3. Build the image and push it to a registry so it's available to all nodes in your swarm:

```
$ docker image build -t <username>/secrets-demo:1.0 .
$ docker image push  <username>/secrets-demo:1.0
```

4. Create and run a service using this image, and use the `-e` flag to create environment variables that point to your secrets:

```
$ docker service create \
    --name secrets-demo \
    --replicas=1 \
    --secret source=mysql-password,target=db_password,mode=0400 \
    -e USERNAME="jdoe" \
    -e PASSWORD_FILE="/run/secrets/db_password" \
    <username>/secrets-demo:1.0
```

5. Figure out which node your container is running on, head over there, connect to the container, and run `python app.py`; the `-e` flag in `service create` has set environment variables to point at your secrets, allowing your app to find them where it expects.

26.6 Conclusion

In this lab we have learned how to create, inspect and list secrets. We also have seen how we can assign secrets to services and investigated how containers actually get the secrets. We then updated the secret of a given service, and finally we created an image that is prepared to consume secrets.