SEIS 665 Week 13 Project: Docker Swarm

Overview

Create a Docker swarm cluster and launch an application platform into it.

Requirements

- AWS account.
- SSH terminal application.

The project

Let's get started!

Create a cluster template using Cloud9

We are going to begin this week's project by building a small Docker Swarm cluster manually. This will give you a basic understanding of how to create a Swarm cluster. Start out by downloading this CloudFormation template to your workstation:

```
https://s3.amazonaws.com/seis665/docker-single-server.json
```

This template creates a basic VPC containing a single EC2 instance running Docker. We need to modify this template so that it will create several Docker instances for a cluster environment. We could make these modifications using a local code editor, but just for fun let's use a cloud-based code editor.

Log into the AWS web console and go to the Cloud9 service dashboard. Create a new environment called `swarm` with a t2.micro EC2 instance environment type. It will take a few minutes to create the Cloud9 environment.
Once the editor environment is running, it will automatically open up a code editor window with several different feature panes. The editor should look pretty familiar to you if you have used other code editors in the past.

Select the file option in the editor to upload local files from your desktop and browse to the location of the `docker-single-server.json` file you downloaded. Cloud9 will upload your JSON file and display it in the editor file explorer tree in the left window pane.
Now modify the `docker-single-server.json` template code in the Cloud9 editor to create three Ubuntu linux EC2 instances running Docker. This should be a pretty trivial change for you at this point. Save your file changes. Now we need to store the file in a location that CloudFormation can access. How about an S3 bucket?
The bottom window pane of the Cloud9 editor provides console access to the EC2 instance that the editor is using. You can run any shell command in this window pane — that's because Cloud9 automatically logged you into the instance. Type in the `ls` command and you should see the JSON template file listed in the current working directory.
Copy the `docker-single-server.json` file from your Cloud9 environment into an S3 bucket in your account. It doesn't matter which S3 bucket you use. Verify that the file exists in your S3 bucket.

Create a Swarm Cluster

Launch a new CloudFormation stack using the template you uploaded to your S3 bucket. If the stack launches properly you can delete your Cloud9 environment. Otherwirse you should make the necessary code changes to fix the template and copy it again to the S3 bucket.

Once the stack is running, log into one of the Ubuntu EC2 instances and run the following command:

```
docker swarm init --advertise-addr <private IP address>
```

Substitute `<private IP address>` for the private IP address of the EC2 instance.
Congratulations! You have just created a Docker Swarm cluster — though this is a very basic cluster with only a single manager node. Make sure you keep track of the IP address of this node (write it down) and make a note that this node is the current leader of the cluster.
The next step is to add additional nodes to the Swarm cluster. In order to do that we need to know the cluster join token. Think of this as a password which other cluster nodes have to provide to join the cluster. There are two different types of join tokens available: a manager token and worker token. In this case we are going to add two more manager nodes to the cluster. Run the following command to get the manager token:

```
docker swarm join-token manager
```

Copy the join token and store it somewhere so that you can retrive it later.

Next, log into one of the other two EC2 instances. Let's turn this node into a manager node and tell it to join the Docker Swarm cluster.

```
docker swarm join --token <manager_token> <leader_ip_address>:2377
```

Substitute `<manager_token>` for the swarm manager token generated by the previous step and substitute `<leader_ip_address>` with the private IP address of the cluster leader node. You may have noticed that the previous `docker swarm join-token` command provided you with this complete command string. The node should successfully join the cluster.
Log into the third remaining EC2 instance and join this node to the Swarm cluster. After it successfully joins, run the following command to verify that the Swarm cluster has three nodes:

```
docker node ls
```

The Docker client should respond by listing three available cluster nodes. When you build a Swarm cluster, you always want the cluster to have an odd number of manager nodes — 1, 3, 5, etc. The cluster uses the Raft algorithm to elect a leader and distribute configuration information, and this algorithm requires a quorum (majority) of the nodes to be active.

Test Swarm cluster

Let's test the Swarm cluster by deploying a simple service into it. Type in the following command:

```
docker service create --name helloworld alpine ping docker.com
```

Verify that the service is running:

```
docker service ls
```

View the service configuration by running:

```
docker service inspect helloworld
```

The Docker client will respond with a large JSON document containing the service configuration information. Note the image specified in the configuration document.

Which cluster node is the service task (container) running on? Type in the following command to find out:

```
docker service ps helloworld
```

The Docker client will display the node the helloworld task is running on. Remember that by default Swarm manager nodes can serve as both managers and workers — meaning that these nodes can also run service tasks. In a production setting we would never run service tasks on a manager node for performance reasons, but for this training environment that's okay.

How do we know what the service is doing? There are two different ways to view the service logs. One way is to log into the cluster node running the service container and look at its logs. Let's try that now. Log into the node that the service container is currently running on. Note, it's possible that the container is running on the node you are currently logged into, so in that case you won't have to log into a different node. Once you have logged into the node, run the following command to get a list of the currently running Docker containers:

```
docker ps
```

Next, show the logs of the running container on this node:

```
docker logs <container ID>
```

You should something that looks like this:

```
PING docker.com (54.209.25.207): 56 data bytes
```

Manually looking at the logs from a single container instance is one way to troubleshoot problems, but what if your service is comprised of a dozen containers? Let's scale up the service and see what happens. Type in the following command to scale up the service:

```
docker service scale helloworld=12
```

Check out how many service tasks are running and which nodes these tasks are running on:

```
docker service ps helloworld
```

How many of those service containers are running on the node you are currently logged into? What command would you use to figure that out? Enter the command now.

You will likely see 4 containers running on the current node. That makes sense because Docker Swarm tries to evenly distribute the containers across the cluster (12 service tasks / 3 cluster nodes = 4 tasks per node).

Now, if we wanted to look at the service logs manually we would have to access each cluster node and look at the individual logs for each running container. Ugh, that's too

much work. Fortunately Docker Swarm provides an easier way. Type in the following command to see the logs for all the containers running in the helloworld service:

```
docker service logs helloworld
```

Notice how each log statement is prefixed with the task name and node ID where the log statement was generated. A running container may generate hundreds of log statements, and when you have a service running a dozen tasks these logs might become a little difficult to read. That's why we would typically push these logs into some sort of centralized log management platform such as an ElasticSearch stream or CloudWatch logs.

We are done using the helloworld service, so go ahead and remove it by running the command:

```
docker service rm helloworld
```

Check to see if any containers are still running on your current node. Docker Swarm automatically stops and removes all of the running containers for you. Pretty nice, eh? This is a simple example of how a container orchestration platform can save you significant time and effort.

Working with secrets

One of the really cool features we can use in Docker Swarm is secrets management. What is a secret? It's any piece of information you need to securely store and transmit to a service running in a Docker container — like a password, token, or security credential.

Traditionally we would provide secrets to a service in a container by setting an environment variable. However, this isn't a secure practice because environment variable values can "leak" out of a container. Docker Swarm provides a more secure method. Let's try it now.

Start out by creating a new Swarm secret (note the dash character at the end):

```
echo secretpass123 | docker secret create my_secret -
```

This will create a new Swarm secret called `my_secret` containing the value `secretpass123`. Swarm will store this secret in an encrypted format (AES256) across all of the cluster manager nodes in the Raft logs. Verify that this secret is available in the cluster by running:

```
docker secret ls
```

Next, launch a new service running a Redis database server and make this secret available to the service containers:

```
docker service create --replicas 3 --name redis --secret my_secret redis
```

At least one of those service replicas is likely running on the current Swarm node. Check to see if that's the case. If not, log into one of the other cluster nodes before proceeding.

When we created the service we made the secret available to all of the service containers. How does Swarm do this? It mounts a temporary file system in the containers and makes any secrets provided available as text files within that file system.

Note the container ID of one of the running service containers and attach to the container using the `docker exec` command:

```
docker exec -it <container ID> bash
```

You are now connected to an interactive shell inside the container. Change to the directory `/run/secrets` and do a directory listing. Do you notice that the Swarm secret you created is displayed as a text file? Look at the contents of this file. You will see the value of the secret displayed.
You can use this secret value when bootstrapping a new service inside the container. If you stop this container and create a new container using the same image, the secrets won't be copied along with the image data because the secrets are located on a temporary file system and not a layer in the container image.

Now you know how to create a Docker Swarm cluster, launch services into it, and store secrets in the cluster. We're done using this environment so you can delete the CloudFormation stack.

Launch a production cluster and stack

In the hands-on portion of the this week's lecture we used a more production-like cluster environment created by a CloudFormation template. Let's launch that cluster environment now. Here's the template:

```
https://s3.amazonaws.com/seis665/docker-swarm.json
```

This environment will take around 10 minutes to launch and when it's complete you will see a set of manager nodes and worker nodes. One of the manager nodes is called the `initiator` node. This isn't an official Docker label, just something that I came up with to designate the node which initially created the Swarm cluster. This is also the only node that you can log into from the Internet. Go ahead and log into this node now.

In the lecture I created a Pets-as-a-Service application environment in Swarm by running a series of commands. First I created an overlay network called `pets-overlay`, then I created two services called `consul` and `web`. Typically we don't launch services into a Docker Swarm cluster by manually running Docker client commands. We use a specially-crafted Docker compose file to launch a Swarm stack (not to be confused with a CloudFormation stack).
Your mission is to write a Docker compose file called `pet-compose.yml` which defines the overlay network and the two application services. The compose file must use the version 3 Docker compose file format. Here are a couple references you an use to help you build the compose file:

```
https://docs.docker.com/compose/compose-file/#service-configuration-reference

https://docs.docker.com/engine/swarm/stack-deploy/
```

A couple things to note:

- Each of the services must be attached to the `pets-overlay` network.
- The `web` service should have 2 replicas
- The `web` service also requires an environment variable (`DB`) to be set.

- Remember you can look at the lecture presentation to see all the settings required for these services.

After creating the compose file, launch a stack called `pets` into your cluster by running the command:

```
docker stack deploy --compose-file pet-compose.yml pets
```

Once the stack has launched in your cluster, you should be able to verify that it is running:

```
docker stack ls
```

You can also see the services the stack created:

```
docker stack services pets
```

The CloudFormation stack created an ELB which distributes requests across the worker nodes in the cluster. Go to the ELB endpoint address in a web browser to verify that the Pets-as-a-Service application is running.

Show me your work

Please show me your Swarm stack compose file code.

Terminate AWS resources

Remember to terminate the CloudFormation stack.