Docker Swarms

Fall 2018 – Introduction to Full-Stack Development

Instructor: Matthew Fritter

matt@frit.me

Stepping Back: Docker

- Last week, we discussed how to build Docker images using Dockerfiles and how to deploy them
  - Use a base Docker image to start with that is system-appropriate, i.e. the CentOS 7 image
  - Use commands to build up the environment of the image until dependencies are met for your application
  - Copy your application files and configuration documents into the image
  - Build it and run it using Docker
- We ran through these steps in Lab 9

Stepping Back: Load Balancing

- We also discussed briefly how Load Balancing works, and why it might be important:
  - Spread out load across multiple machines or images to prevent any single application instance from being overloaded
  - Allows you to continue serving your web application, even if one of your machines or images is offline or goes down
  - Different techniques for distributing load: round robin queue (simple but can lead to some problems) versus dynamic load balancing by monitoring load on each server/image (complicated, but can avoid hangups if a server is stalled/fully loaded)

Announcements

- Quiz 3 is marked and can be picked up at the front
- Still working on lab marking
- There will be no lab for this week. However, I will be in the lab to assist anyone who needs help with Labs 8 and 9

Docker Swarms

What is a Docker Swarm?

- Swarms are ways of handling multiple Docker instances, ranging from a handful to thousands
- Like Docker itself, you can set up Swarms on Linux, Windows, and Mac hosts
- Uses a fairly simple manager-worker system to delegate work – commands are routed to the manager, which coordinates the swarm
- Swarms are examples of cluster computing, and are an example of the applications of fast deployment systems (like Docker), and load balancing

Nodes in a Docker Swarm

- Each instance within the swarm is known as a node
- Instances may be virtual machines (such as those made with VirtualBox or Vmware), or they may be actual physical servers
- Every swarm will have a manager node, and one or more worker nodes
- Communication between the nodes requires a couple different ports be open on each instance:
  - TCP port 2377 for cluster management
  - TCP/UDP port 7946 for inter-node communications
  - UDP port 4789
  - If your nodes can’t communicate to each other, your cluster isn’t a cluster!

Creating a Swarm

- When you first create your swarm, you'll only have one node – the instance that you are currently working on:

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

- This command starts the swarm and begins advertising the specified IP address (usually the IP of the server you ran the command on)
  - By default, the machine you created the swarm on will be the manager of the swarm. Take this into consideration if you are working with servers that aren't identical – which machine would be best to have as a manager?
- This command will also generate and print a swarm join command containing a special unique token – make sure you hold onto this
- The command `docker node ls` will list current nodes in the swarm. At this point, it will only show one node, the manager.

Growing the Swarm

- Obviously, it's not particularly useful to have a cluster of only one instance
- Using the command and token that was generated when we started our swarm, we can have other instances join our swarm
- Once we've added additional workers to the swarm, you can use `docker node ls` again; you'll see your new nodes are now listed as workers
- You may add as many nodes as you wish as workers. However, there's no need to rush – you can add additional nodes to your swarm at a later time, even after you've deployed your service.

Deploying Services to Swarms

- Now we have a swarm of nodes, including a manager and workers – but what shall we do with them?
- Now, we can deploy our Docker images that we've previously created:

```
docker service create --replicas <#> --name <name>
<image> ping docker.com
```

  - The - - replicas flag denotes the number of services to create
  - The - - name flag gives the service a human-readable name
  - The image is a Docker image
- Once you've started the service, you can use `docker service ls` to see the services that are currently running

Some Notes on Deploying Services

- If you only deploy a single replica, it will probably be running on the manager – by default, the manager is configured to run services just like workers do
- Conversely, if you define more replicas than there are workers, some nodes may have the service doubled up, running as multiple instances
  - You can, for instance, have 1000 replicas of a service spread across 2 or 3 nodes, if you want

Scaling the Swarm

- You may also dynamically rescale an already-running service. When you use `docker service ls`, you'll notice that each service has a unique ID associated with it.
- With that ID, specify:

```
docker service scale <ID>=<#>
```

- This will start new # number of new instances of the service identified by ID
  - Note: This will kill the existing instance(s) of the service and restart them fresh – keep this in mind if you're scaling a critical service that *cannot* go down!
  - As with designating a new service, the scaled service instances will be distributed across the nodes in your swarm

Killing Services

- To kill a service, simply call:
  - `docker service rm <serviceName>`
- You can then use `docker ps` to confirm the services have been removed. Note that it may take a brief period for the services to be shut down.
- If you want to kill the swarm entirely, there's a couple options:
  - Remove all managers. This can be risky as workers may continue running services without any oversight or management
  - Remove all nodes, and then remove all managers, leaving the swarm empty and defunct
- The command for a node to leave the swarm is:

```
docker swarm leave
```

Using Web Services

- Previously, we looked at how to make web services run in a Docker image available to the outside world
- We can build on that to create distributed web services using swarms:

```
docker service create --name nginxtest --publish
published=8099,target=80 --replicas 2 nginx
```

- The above command will create two instances of the default nginx web server image, and forward the container port 80 to the exterior port 8099
  - You can verify this is true by visiting the IP addresses of your manager and worker at port 8099, and seeing the NGINX web server default startup page

Using Web Services, Continued

- Perhaps most important about this is that when you access any given node via that IP address, you aren't *actually* accessing that node necessarily
- When set up this way, Docker uses a *routing mesh* – any request made to the specified published port (8099) will actually get routed through a built in swarm load balancer, and then passed off to one of the nodes
- You can prevent this by specifying `mode=host` after the -- publish flag; this will ensure that visiting the specified port at a specific IP will return the service that is running on that specific node, and the request is not sent through the load balancer

Why Use Docker Swarms?

- Think of the potential power and ease of use that this provides
  - Build a network of thousands of Amazon AWS instances, all serving a web application run through the routing mesh as a load distributed application => Incredible stability and potential to handle massive traffic
  - From a more evil side: Thousands of machines running hundreds of Docker instances making phony requests and connections => Serious DDoS network
- From a security perspective: Docker instances are probably less susceptible to attacks than your machine is. If someone manages to break into a Docker instance, it's usually only a case of having to kill off that particular instance

  *

  and restart it

That’s All Folks!