Docker Swarm

CONNECTING AND MANAGING MULTIPLE CONTAINERS

Agenda

Intro / Prep Environments

Day 1: Docker Deep Dive

Day 2: Docker Advanced Deep Dive

Day 3: Kubernetes Deep Dive

Day 4: Advanced Kubernetes: Concepts, Management, Middleware

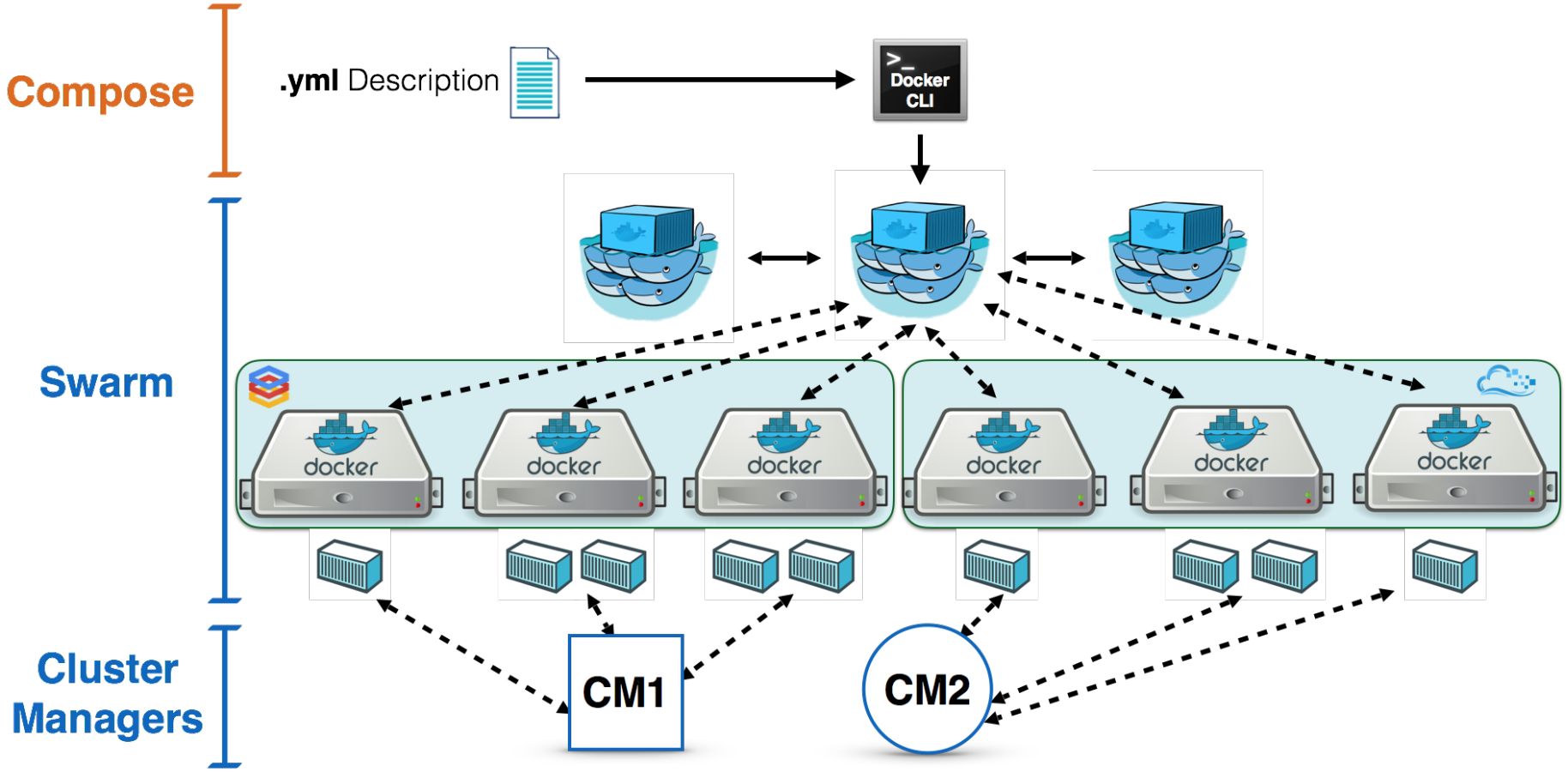
Recap

Recap Docker

- Containers run on **single** Docker host
- Containers are **ephemeral**
- Containers can have external persistence
- Containers do not contain
- Operating system **matters**

Swarm Overview

- Describe swarm in a sentence of two, here.



Initialize Swarm Process

- Turn single host Docker host into a Multi-host with Docker Swarm Mode.
- All containers are only **deployed** onto the engine.
- Swarm Mode turns it into a multi-host **cluster-aware** engine.
- The first node to initialize the Swarm Mode becomes the **manager**.
- As new nodes join the cluster, they can **adjust** their roles between managers or workers.
- You **should** run 3-5 managers in a production environment to ensure high availability.

Create Swarm Mode Cluster

- Swarm Mode is built **into** the Docker CLI. You can find an overview the possibility commands via:

```
docker swarm --help
```

- The most important one is how to **initialize** Swarm Mode. Initialization is done via *init*, like this:

```
docker swarm init
```

- After running the command, the Docker Engine knows how to work with a cluster and becomes the **manager**.
- The results of an initialization is a **token** used to add additional nodes in a secure fashion.

Swarm Init Output

- Once you initialize Docker Swarm, the token output will look something like this:

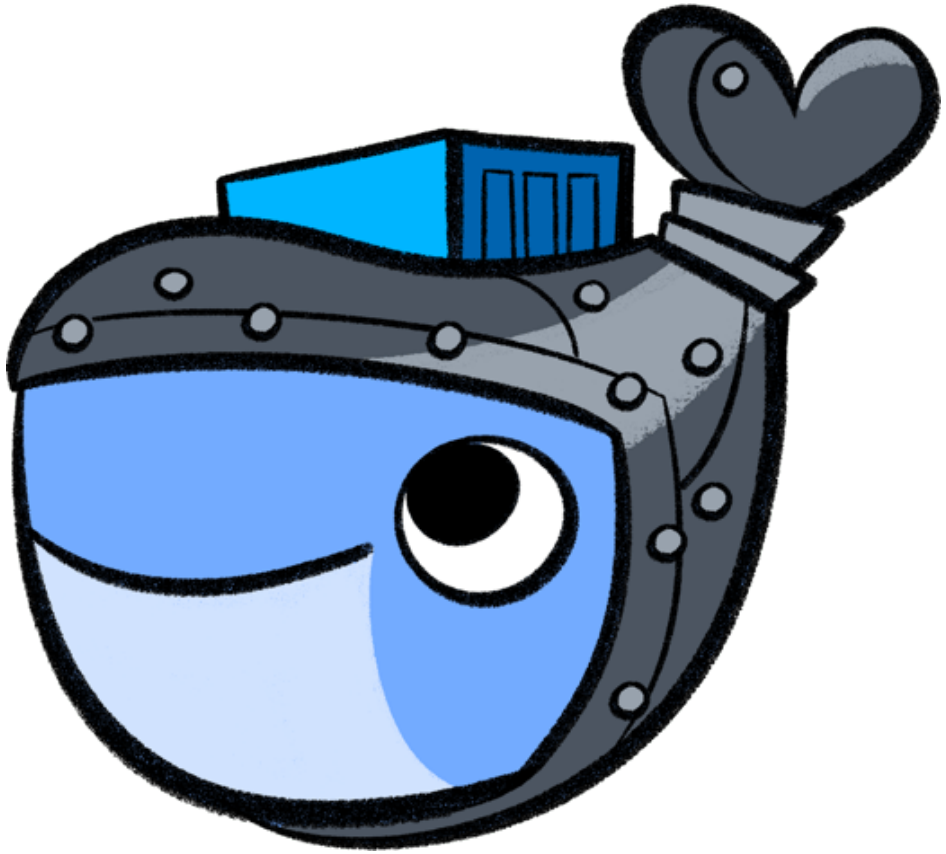
```
WSwarm initialized: current node (gpnnuxrri0tzaszblvhdrwqsh) is
now a manager.

To add a worker to this swarm, run the following command:

docker swarm join \

--token SWMTKN-1-
1xffnpg61r5wgel28ehuh8o3ip5z2iiwmplbemnksfho5akdqy-
3zkn8ssrkcnjei34f5rmd4n3q \

172.17.0.65:2377

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

Creating a Cluster

- With Swarm Mode enabled, it is possible to **add** additional nodes and issues commands across all of them.
- If nodes happen to disappear, for example because of a **crash**, the containers which were running on those hosts will be automatically rescheduled onto other available nodes.
- The rescheduling ensures you do not lose capacity and provides **high-availability**.
- On each additional node, you wish to add to the cluster, use the Docker CLI to **join** the existing group.
- Joining is done by pointing the other host to a **current** manager of the cluster.

Creating a Cluster

- Docker now uses an **additional** port, *2377*, for managing the Swarm.
- The port should be blocked from public access and only accessed by **trusted** users and nodes.
- We recommend using VPNs or private networks to **secure** access.



Join Cluster

- The first task is to **obtain** the token required to add a node to the cluster.
- For demonstration purposes, we'll ask the manager what the **token** is via *swarm join-token*.

```
token=$(docker -H 172.17.0.65:2345 swarm join-token -q
worker) && echo $token
```

- NOTE: In production, this token should be stored securely and only accessible by trusted individuals.

Join Cluster

- On the second host, **join** the cluster by requesting access via the manager.
- The **token** is provided as an additional parameter, like this:

```
docker swarm join 172.17.0.65:2377 --token $token
```

Join Cluster

- By **default**, the manager will automatically accept new nodes being added to the cluster.
- You can **view** all nodes in the cluster using:

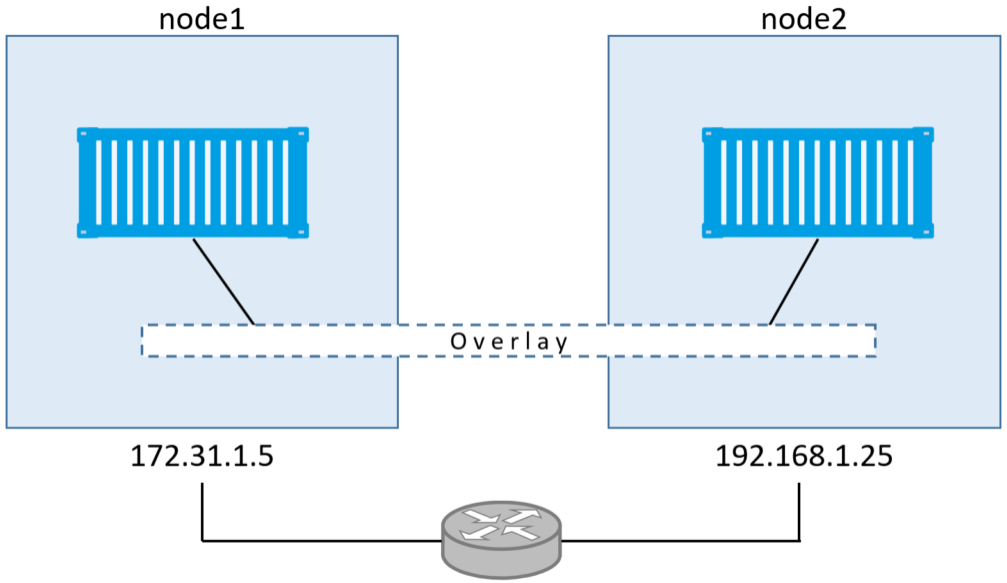
```
docker node ls
```

- Output:

```
 ID                                  HOSTNAME   STATUS   AVAILABILITY   MANAGER
STATUS5re1humv246m6clujn0zng9jy      docker     Ready
Activerbavg04vwiijyzzjwdjoc3io2 *   docker     Ready    Active         Leader
```

Overlay Network

- Swarm Mode also introduces another **networking** model.
- In **previous** versions, Docker required the use of an external key-value store, such as Consul, to ensure consistency across the network.
- The need for consensus and KV has now been incorporated **internally** into Docker and no longer depends on external services.



Overlay Network

- The improved networking approach follows the **same** syntax as previously.
- The **overlay** network is used to enable containers on different hosts to communicate.
- Under the covers, this is a Virtual Extensible LAN (**VXLAN**), designed for large scale cloud based deployments.

Overlay Network

- The following command will create a new overlay network called **skynet**:

```
docker network create -d overlay skynet
```

- All containers **registered** to this network can communicate with each other, regardless of which node they are deployed onto.

Deploy Service

- By default, Docker uses a **spread** replication model for deciding which containers should run on which hosts.
- The spread approach ensures that containers are deployed across the cluster **evenly**.
- If one of the nodes are **removed** from the cluster, the containers running are spread across the other available nodes.

Deploy Service

- The new **Services** concept is used to run containers across the cluster.
- This is a **higher-level** concept than containers.
- A service allows you to define how applications should be deployed at **scale**.
- By updating the service, Docker updates the container required in a **managed** way.

Deployment

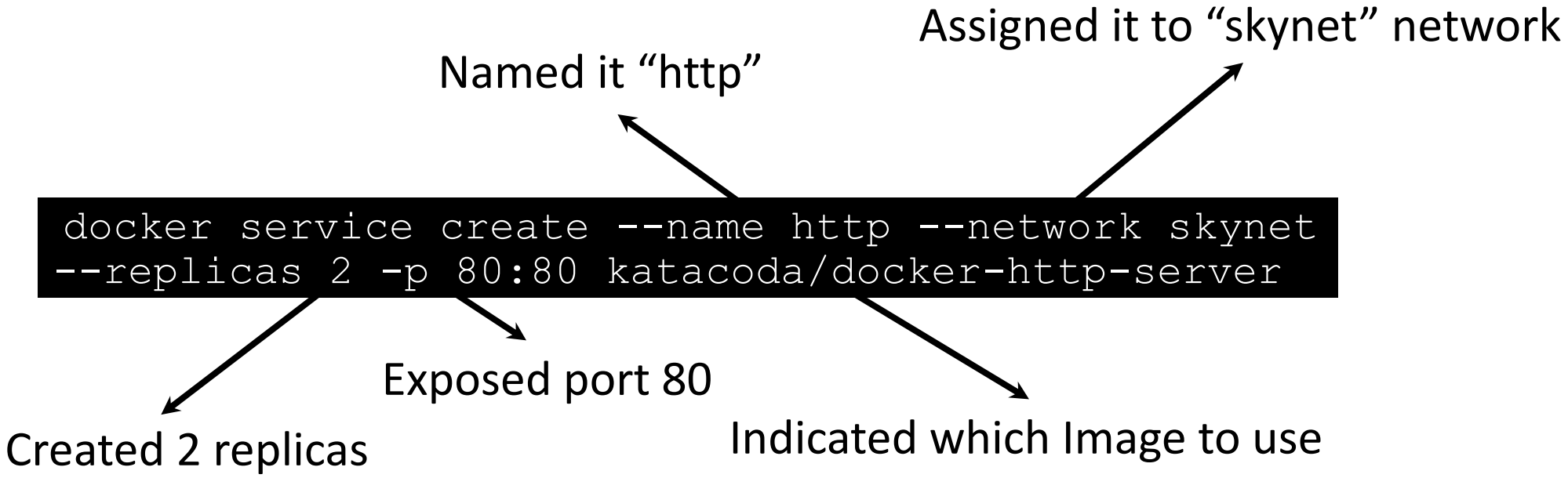
- As an **example**, let's deploy the Docker Image `katacoda/docker-http-server`.
- We will give it a friendly **name** like, `http` **.**
- And we will **attach** it to the newly created `skynet` network.
- For ensuring replication and availability, we will run two instances of **replicas**, of the container across our cluster.

Deployment

- Finally, we will load balance these two containers **together** on port *80*.
- Sending an HTTP request to any of the nodes in the cluster will process the request by **one** of the containers within the cluster.
  - NOTE: The node which accepts the request might not be the node where the container responses. Instead, Docker load-balances requests across all available containers.
- So, what should our **code** look like?

Deployment

- **Here** it is:



Deployment Verification

- You can **view** the services running on the cluster using the CLI command

```
docker service ls
```

- Output:

```
ID             NAME   MODE         REPLICAS   IMAGE
4l4yr9hv663w   http   replicated   2/2        katacoda/docker-http-
      server:latest
```

Deployment Verification

- As containers are started you will see them using the `ps` command. You should see one instance of the container on **each** host.
- **Listing** containers on the first host using `docker ps` will produce:

```
CONTAINER ID    IMAGE…
<host-1-id>     Image:name
```

- Listing containers on the **second** host using `docker ps` produces:

```
CONTAINER ID    IMAGE…
<host-2-id>     Image:name
```

Deployment Verification

- If we issue an HTTP request to the **public** port, it will be processed by the two containers:

```
curl <localhost>
```

- Output:

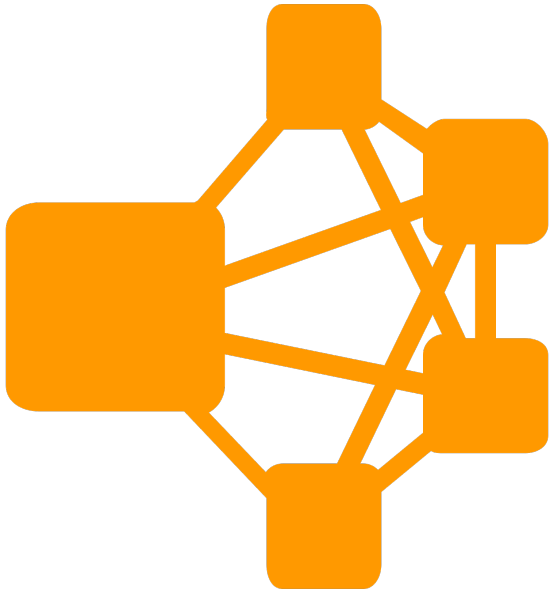
```
<h1>This request was processed by host: “host-1-id”</h1>
```

- If we issue the same command again, what do you think the results will be?

Deployment Verification

- Everything happens just like before, except this request was processed by host 2's container instance:

```
<h1>This request was processed by host: "host-2-id"</h1>
```



Notice which container id is given. Again, using `ps` we could verify this.

What if we do it again?

Health Inspection

- The Service concept allows you to inspect the health and state of your cluster and the running applications.
- You can view the list of all the tasks associated with a service across the cluster:

```
docker service ps http
```

  - NOTE: In this case, each task is a container.
- Output:

```
ID                  NAME    IMAGE                                  NODE
<service-id-1> http.1   katacoda/docker-http-server:latest   docker
<service-id-2> http.2   katacoda/docker-http-server:latest   docker
```

Health Inspection

- You can view more details and the configuration of a service by using:

```
docker service inspect --pretty http
```

- Output:

```
ID:             9b60zdcri52tvo1sskwsta44f
Name:           http
Service Mode:   Replicated
Replicas:       2
...
```

Health Inspection

- On each node, you can ask what tasks it is currently running.

```
docker node ps self
```

- NOTE: Self refers to the manager node Leader:

- Output:

```
ID                  NAME     IMAGE                                    NODE …
<service-id-1> http.2   katacoda/docker-http-server:latest   docker …
```

Health Inspection

- Using the ID of a node you can query individual hosts

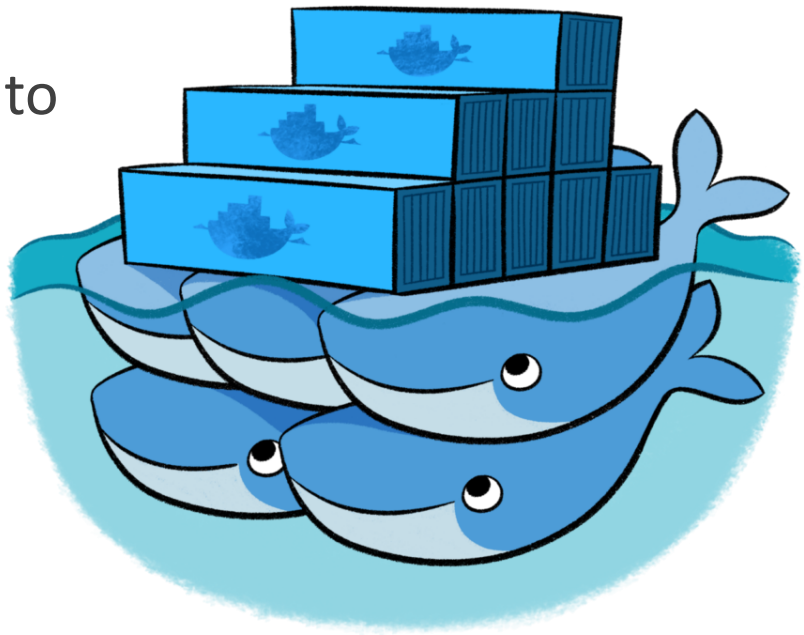
```
docker node ps $(docker node ls -q | head -n1)
```

- Output:

```
ID                  NAME    IMAGE                                  NODE …
<service-id-2> http.1   katacoda/docker-http-server:latest   docker …
```

Scale Service

- A Service also allows us to scale how many instances of a task is running across the cluster.
- As it understands how to launch containers and which containers are running, it can easily start, or remove, containers as required.
- At the moment the scaling is manual.
  - However, the API could be hooked up to an external system such as a metrics dashboard.



Scale Service

- In our example, we had two load-balanced containers running, processing our requests `curl docker.`
- Command below would scale our `http` service to be run across five containers:

```
docker service scale http=5
```

- On each host, you will see additional nodes being started.
- The load balancer will automatically be updated.
- Requests will now be processed across the new containers.

Result Summary

- The result of this scenario is a two-node Swarm cluster which can run load-balanced containers that can be scaled up and down.

Lab

End of Chapter