Docker Swarm: Preparing for Cloud Deployment

Cloud Bootcamp 6.4

Today's Objectives

- Create your first Docker Swarm

Our sample application

Note:

> `node1` = `localhost` your own laptop / desktop
> `nodeX` = other Docker hosts we will create, X=1,2,...

- We will clone a GitHub repository onto our `node1`

- The repository also contains scripts and tools for other labs/worksheets (next week, if time allows)

- Clone the repository on `node1`:

  ```
  git clone https://github.com/jpetazzo/container.training
  ```

(You can also fork the repository on GitHub and clone your fork if you prefer that.)

Downloading and running the application

Let's start this before we look around, as downloading will take a little time...

- Go to the `dockercoins` directory, in the cloned repository:

  ```
  cd container.training/dockercoins
  ```

- Use Compose to build and run all containers:

  ```
  docker compose up
  ```

Compose tells Docker to build all container images (pulling the corresponding base images), then starts all containers, and displays aggregated logs.

What's this application?

- It is a DockerCoin miner! 💰🐳📦🚢
- No, you can't buy coffee with DockerCoin
- How dockercoins works:
    - generate a few random bytes
    - hash these bytes
    - increment a counter (to keep track of speed)
    - repeat forever!
- DockerCoin is *not* a cryptocurrency
  (the only common points are "randomness," "hashing," and "coins" in the name)

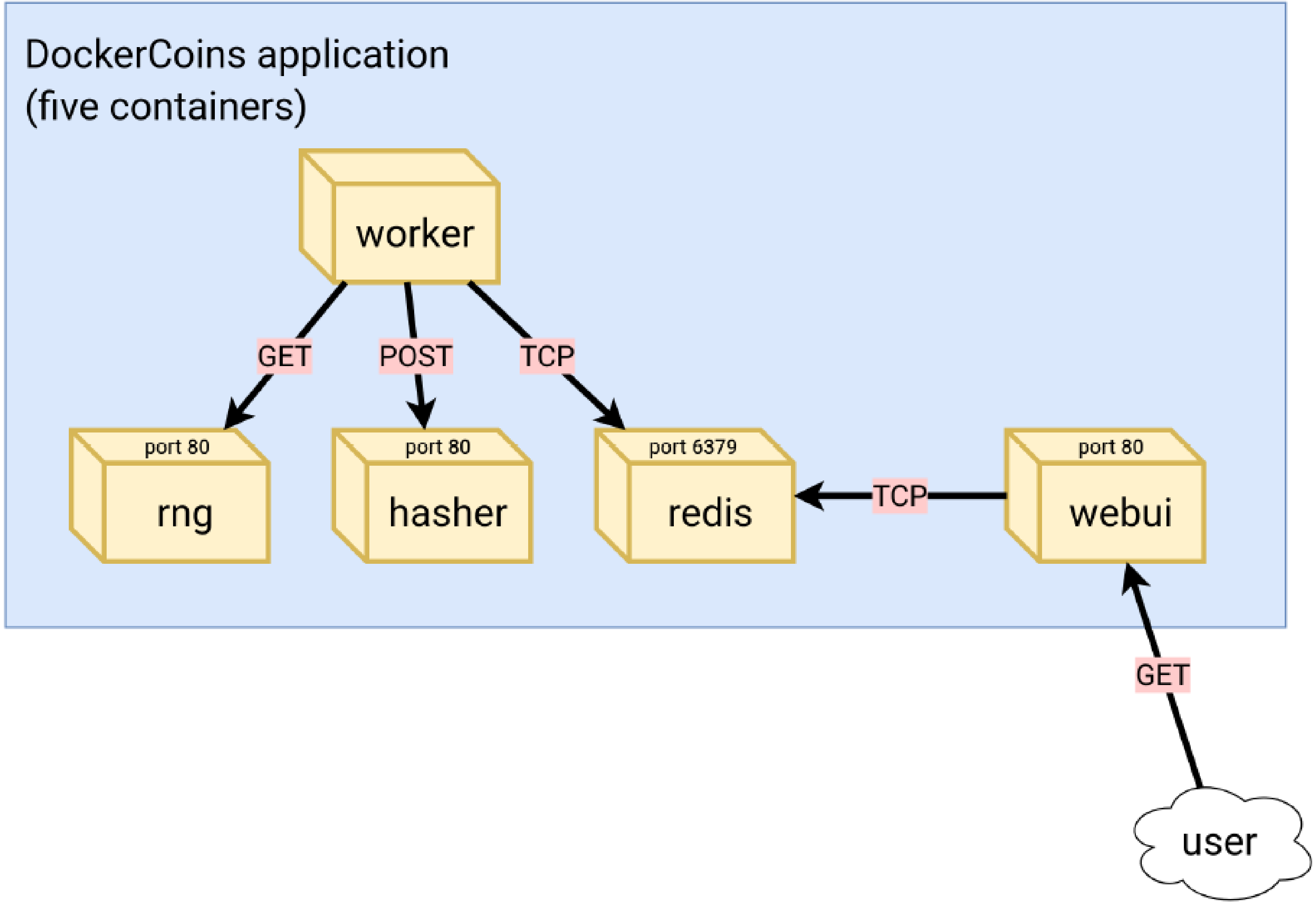
DockerCoin in the microservices era

- The dockercoins app is made of 5 services:
  - `rng` = web service generating random bytes
  - `hasher` = web service computing hash of POSTed data
  - `worker` = background process calling `rng` and `hasher`
  - `webui` = web interface to watch progress
  - `redis` = data store (holds a counter updated by `worker`)
- These 5 services are visible in the application's Compose file `docker-compose.yml`

How dockercoins works

- `worker` invokes web service `rng` to generate random bytes
- `worker` invokes web service `hasher` to hash these bytes
- `worker` does this in an infinite loop
- every second, `worker` updates `redis` to indicate how many loops were done
- `webui` queries `redis`, and computes and exposes "hashing speed" in our browser

(See diagram on next slide!)


DockerCoins application
(five containers)
worker
GET
POST
TCP
port 80
rng
port 80
hasher
port 6379
redis
TCP
port 80
webui
GET
user

Service discovery in container-land

How does each service find out the address of the other ones?

- We do not hard-code IP addresses in the code
- We do not hard-code FQDNs in the code, either
- We just connect to a service name, and container-magic does the rest

  (And by container-magic, we mean "a crafty, dynamic, embedded DNS server")

Example in `worker/worker.py`

```
redis = Redis("redis")


def get_random_bytes():
    r = requests.get("http://rng/32")
    return r.content


def hash_bytes(data):
    r = requests.post("http://hasher/",
                      data=data,
                      headers={"Content-Type": "application/octet-stream"})
```

Our application at work

- On the left-hand side, the "rainbow strip" shows the container names
- On the right-hand side, we see the output of our containers
- We can see the `worker` service making requests to `rng` and `hasher`
- For `rng` and `hasher`, we see HTTP access logs

Connecting to the web UI

- "Logs are exciting and fun!" (No-one, ever)
- The `webui` container exposes a web dashboard; let's view it

> With a web browser, connect to `node1` on port 8000

- A drawing area should show up
  - after a few seconds, a blue graph will appear.

Stopping the application

- If we interrupt Compose (with `^C`), it will politely ask the Docker Engine to stop the app
- The Docker Engine will send a `TERM` signal to the containers
- If the containers do not exit in a timely manner, the Engine sends a `KILL` signal

> Stop the application by hitting `^C`

- Some containers exit immediately, others take longer.
- The containers that do not handle `SIGTERM` end up being killed after a 10s timeout. If we are very impatient, we can hit `^C` a second time!

Restarting in the background

- Many flags and commands of Compose are modeled after those of `docker`
- Start the app in the background with the `-d` option:

  ```
  docker compose up -d
  ```

- Check that our app is running with the `ps` command:

  ```
  docker compose ps
  ```

`docker compose ps` also shows the ports exposed by the application.

Viewing logs

- The `docker-compose logs` command works like `docker logs`
- View all logs since container creation and exit when done:

  ```
  docker-compose logs
  ```

- Stream container logs, starting at the last 10 lines for each container:

  ```
  docker-compose logs --tail 10 --follow
  ```

Tip: use `^S` to pause and any other key to resume log output.

Looking at resource usage

- Let's look at CPU, memory, and I/O usage

> [Linux/Mac only] run `top` to see CPU and memory usage (you should see idle cycles)

- We have available resources.
    - How can we use them?

Scaling workers on a single node

- Docker Compose supports scaling
- Let's scale `worker` and see what happens!

> Start one more `worker` container:

```
docker-compose up -d --scale worker=2
```

> Look at the performance graph (it should show a x2 improvement)

> Look at the aggregated logs of our containers (`worker_2` should show up)

> Look at the impact on CPU load with e.g. top (it should be negligible)

Adding more workers

- Great, let's add more workers and call it a day, then!

> Start eight more `worker` containers:

```
docker-compose up -d --scale worker=10
```

> Look at the performance graph: does it show a x10 improvement?

> Look at the aggregated logs of our containers

> Look at the impact on CPU load and memory usage

Identifying bottlenecks

- You should have seen a 3x speed bump (not 10x)
- Adding workers didn't result in linear improvement
- The code doesn't have instrumentation
- *Something else* is slowing us down ... But what?
- We'd now use HTTP performance analysis!

Accessing internal services

- `rng` and `hasher` are exposed on ports 8001 and 8002
- This is declared in the Compose file:

  ```
  ...
  rng:
    build: rng
    ports:
    - "8001:80"

  hasher:
    build: hasher
    ports:
    - "8002:80"
  ...
  ```

Measuring latency under load

We will use `httping` running on `node1`.

- Check the latency of `rng`:

  ```
  docker run --rm --network="host" bretfisher/httping -c 5 localhost:8001
  ```

- Check the latency of `hasher`:

  ```
  docker run --rm --network="host" bretfisher/httping -c 5 localhost:8002
  ```

You will see `rng` has a much higher latency than `hasher`.

Let's draw hasty conclusions

- The bottleneck seems to be `rng`
- *What if* we don't have enough entropy and can't generate enough random numbers?
- We need to scale out the `rng` service on multiple machines!

Note: this is a fiction! We have enough entropy. But we need a pretext to scale out.

(In fact, the code of `rng` uses `/dev/urandom`, which never runs out of entropy and is just as good as `/dev/random`.)

Clean up

- Before moving on, let's remove those containers

> Tell Compose to remove everything:
>
> ```
> docker-compose down
> ```

SwarmKit

- SwarmKit is an open source toolkit to build multi-node systems
- It is a reusable library, like libcontainer, libnetwork, vpnkit ...
- It is a plumbing part of the Docker ecosystem

SwarmKit features

- Highly-available, distributed store based on Raft
  - (avoids depending on an external store: easier to deploy; higher performance)
- Dynamic reconfiguration of Raft without interrupting cluster operations
- *Services* managed with a *declarative API*
  - (implementing *desired state* and *reconciliation loop*)
- Integration with overlay networks and load balancing
- Strong emphasis on security:
  - automatic TLS keying and signing; automatic cert rotation
  - full encryption of the data plane; automatic key rotation
  - least privilege architecture (single-node compromise ≠ cluster compromise)
  - on-disk encryption with optional passphrase

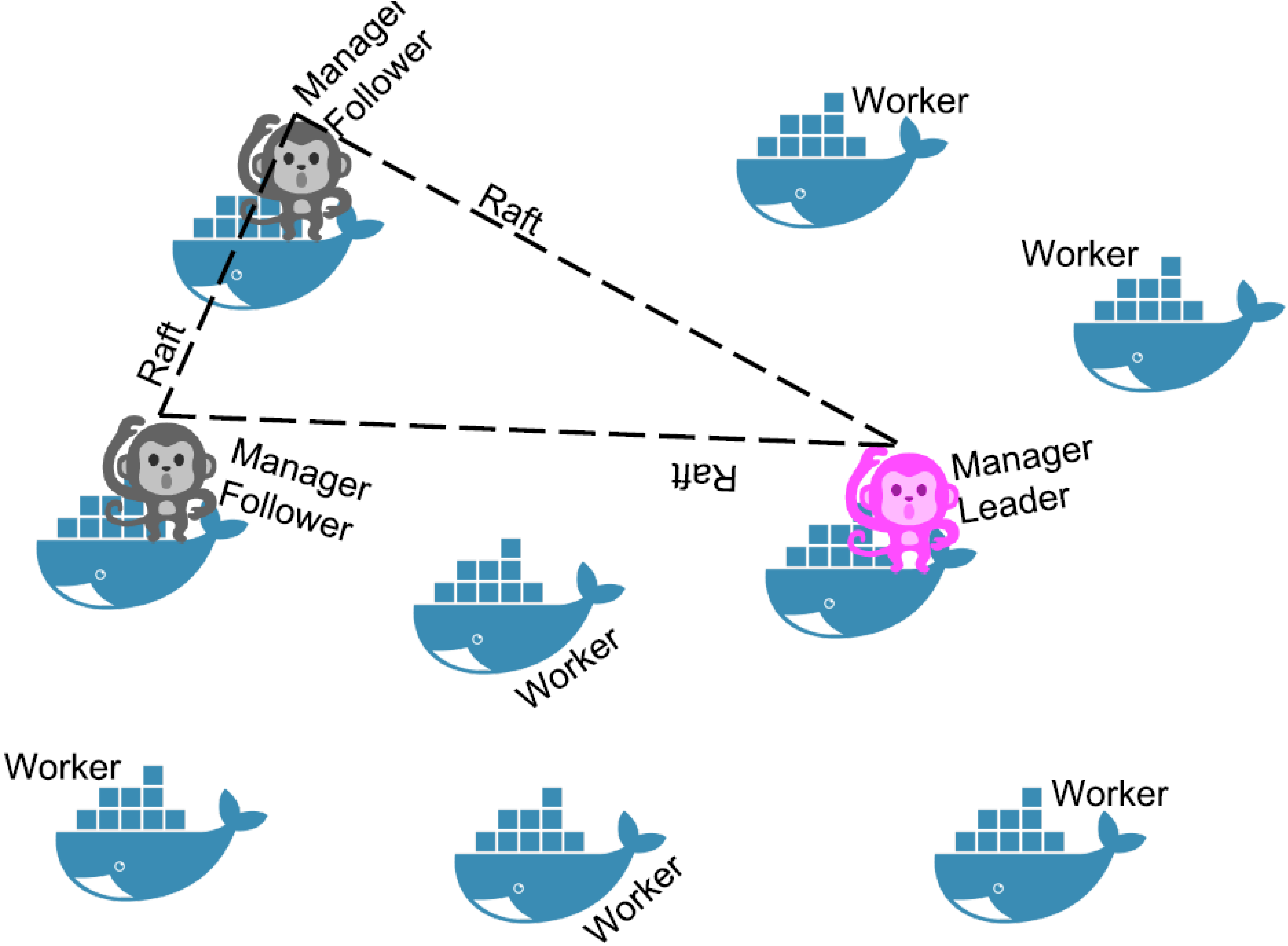
SwarmKit concepts (1/2)

- A *cluster* will be at least one *node* (preferably more)
- A *node* can be a *manager* or a *worker*
- A *manager* actively takes part in the Raft consensus, and keeps the Raft log
- You can talk to a *manager* using the SwarmKit API
- One *manager* is elected as the *leader*; other managers merely forward requests to it
- The *workers* get their instructions from the *managers*
- Both *workers* and *managers* can run containers

Illustration

On the next slide:

- whales = nodes (workers and managers)
- monkeys = managers
- purple monkey = leader
- grey monkeys = followers
- dotted triangle = raft protocol


Manager
Follower
Worker
Raft
Worker
Raft
Manager
Follower
Raft
Manager
Leader
Worker
Worker
Worker
Worker

SwarmKit concepts (2/2)

- The *managers* expose the SwarmKit API
- Using the API, you can indicate that you want to run a *service*
- A *service* is specified by its *desired state*: which image, how many instances...
- The *leader* uses different subsystems to break down services into *tasks*:
  - orchestrator, scheduler, allocator, dispatcher
- A *task* corresponds to a specific container, assigned to a specific *node*
- *Nodes* know which *tasks* should be running, and will start or stop containers accordingly (through the Docker Engine API)

You can refer to the NOMENCLATURE in the SwarmKit repo for more details.

Declarative vs imperative

- Our container orchestrator puts a very strong emphasis on being *declarative*
- Declarative:
    - *I would like a cup of tea.*
- Imperative:
    - Boil some water.
    - Pour it in a teapot.
    - Add tea leaves.
    - Steep for a while.
    - Serve in a cup.
- Declarative seems simpler at first ...
    - ... As long as you know how to brew tea

Declarative vs imperative

- What declarative would really be:
  - *I want a cup of tea, obtained by pouring an infusion[1] of tea leaves in a cup.*
  - *[1]An infusion is obtained by letting the object steep a few minutes in hot[2] water.*
  - *[2]Hot liquid is obtained by pouring it in an appropriate container[3] and setting it on a stove.*
  - *[3]Ah, finally, containers! Something we know about. Let's get to work, shall we?*

[Did you know there was an ISO standard specifying how to brew tea?

Declarative vs imperative

- Imperative systems:
  - simpler
  - if a task is interrupted, we have to restart from scratch
- Declarative systems:
  - if a task is interrupted (or if we show up to the party half-way through), we can figure out what's missing and do only what's necessary
  - we need to be able to *observe* the system
  - ... and compute a "diff" between *what we have* and *what we want*

Swarm mode

- Since version 1.12, the Docker Engine embeds SwarmKit
- All the SwarmKit features are "asleep" until you enable "Swarm mode"
- Examples of Swarm Mode commands:
  - `docker swarm` (enable Swarm mode; join a Swarm; adjust cluster parameters)
  - `docker node` (view nodes; promote/demote managers; manage nodes)
  - `docker service` (create and manage services)

Swarm mode needs to be explicitly activated

- By default, all this new code is inactive
- Swarm mode can be enabled, "unlocking" SwarmKit functions
  - (services, out-of-the-box overlay networks, etc.)

> Try a Swarm-specific command:
>
> ```
> docker node ls
> ```

You will get an error message:

```
Error response from daemon: This node is not a swarm manager. [...]
```

Docker Swarms

Creating our first Swarm

- The cluster is initialized with `docker swarm init`
- This should be executed on a first, seed node
- Warning: DO NOT execute `docker swarm init` on multiple nodes!
  - You would have multiple disjoint clusters.

> Create our cluster from node1:

```
docker swarm init
```

If Docker tells you that it `could not choose an IP address to advertise`, see next slide!

IP address to advertise (in case of IP error)

- When running in Swarm mode, each node *advertises* its address to the others (i.e. it tells them *"you can contact me on 10.1.2.3:2377"*)
- If the node has only one IP address, it is used automatically (The addresses of the loopback interface and the Docker bridge are ignored)
- If the node has multiple IP addresses, you **must** specify which one to use (Docker refuses to pick one randomly)
- You can specify an IP address or an interface name (in the latter case, Docker will read the IP address of the interface and use it)
- You can also specify a port number (otherwise, the default port 2377 will be used)

Using a non-default port number

- Changing the *advertised* port does not change the *listening* port
- If you only pass `--advertise-addr eth0:7777`, Swarm will still listen on port 2377
- You will probably need to pass `--listen-addr eth0:7777` as well
- This is to accommodate scenarios where these ports *must* be different (port mapping, load balancers...)

Example to run Swarm on a different port:

```
docker swarm init --advertise-addr eth0:7777 --listen-addr eth0:7777
```

Which IP address should be advertised?

- If your nodes have only one IP address, it's safe to let autodetection do the job
- If your nodes have multiple IP addresses, pick an address which is reachable *by every other node* of the Swarm

```
docker swarm init --advertise-addr 172.24.0.2
docker swarm init --advertise-addr eth0
```

Token generation

- In the output of `docker swarm init`, we have a message confirming that our node is now the (single) manager:

  ```
  Swarm initialized: current node (8jud...) is now a manager.
  ```

- Docker generated two security tokens (like passphrases or passwords) for our cluster

- The CLI shows us the command to use on other nodes to add them to the cluster using the "worker" security token:

  ```
    To add a worker to this swarm, run the following command:
      docker swarm join --token SWMTKN-1-59fl4ak4nqjmao1ofttrc4eprhrola2l87...
  ```

Checking that Swarm mode is enabled

> Run the traditional `docker info` command:

```
docker info
```

The output should include:

```
Swarm: active
 NodeID: 8jud7o8dax3zxbags3f8yox4b
 Is Manager: true
 ClusterID: 2vcw2oa9rjps3a24m91xhvv0c
 ...
```

Running our first Swarm mode command

- Let's retry the exact same command as earlier
- List the nodes (well, the only node) of our cluster:

```
docker node ls
```

The output should look like the following:

```
ID                HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
8jud...ox4b *     node1     Ready   Active        Leader
```

Adding nodes to the Swarm

- A cluster with one node is not a lot of fun
- Let's add `node2` !

> Run the following to spin up a suitable node
>
> ```
> docker run --privileged --name node2 docker:dind
> ```

- We now need the token that was shown earlier
  - You wrote it down, right?
  - Don't panic, we can easily see it again 😏

Adding nodes to the Swarm

On `node1`

> Show the token again:

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

> Log into `node2` using the following (or the VSCode Docker extension)

```
docker exec -it node2 sh
```

> Copy-paste the `docker swarm join ...` command (that was displayed just before) in the `node2` terminal

Check that the node was added correctly

- Stay on `node2` for now!
- We can still use `docker info` to verify that the node is part of the Swarm:
  ```
  docker info | grep Swarm
  ```
- However, Swarm commands will not work; try, for instance:
  ```
  docker node ls
  ```
- This is because the node that we added is currently a *worker*
- Only *managers* can accept Swarm-specific commands

View our two-node cluster

- Let's go back to `node1` and see what our cluster looks like

> Switch back to `node1` (with `exit`, `Ctrl-D` ...)

> View the cluster from `node1`, which is a manager:

```
docker node ls
```

The output should be similar to the following:

```
ID                HOSTNAME   STATUS   AVAILABILITY   MANAGER STATUS
8jud...ox4b *     node1      Ready    Active         Leader
ehb0...4fvx       node2      Ready    Active
```

Under the hood: docker swarm init

When we do `docker swarm init`:

- a keypair is created for the root CA of our Swarm
- a keypair is created for the first node
- a certificate is issued for this node
- the join tokens are created

Under the hood: join tokens

There is one token to *join as a worker*, and another to *join as a manager*.

The join tokens have two parts:

- a secret key (preventing unauthorized nodes from joining)
- a fingerprint of the root CA certificate (preventing MITM attacks)

If a token is compromised, it can be rotated instantly with:

```
docker swarm join-token --rotate <worker|manager>
```

Under the hood: docker swarm join

When a node joins the Swarm:

- it is issued its own keypair, signed by the root CA
- if the node is a manager:
  - it joins the Raft consensus
  - it connects to the current leader
  - it accepts connections from worker nodes
- if the node is a worker:
  - it connects to one of the managers (leader or follower)

Under the hood: cluster communication

- The *control plane* is encrypted with AES-GCM; keys are rotated every 12 hours
- Authentication is done with mutual TLS; certificates are rotated every 90 days

  (`docker swarm update` allows to change this delay or to use an external CA)
- The *data plane* (communication between containers) is not encrypted by default

  (but this can be activated on a by-network basis, using IPSEC,
  leveraging hardware crypto if available)

Under the hood: I want to know more!

Revisit SwarmKit concepts:

- Docker 1.12 Swarm Mode Deep Dive Part 1: Topology (video)
- Docker 1.12 Swarm Mode Deep Dive Part 2: Orchestration (video)

Adding more manager nodes

- Right now, we have only one manager (node1)
- If we lose it, we lose quorum - and that's *very bad!*
- Containers running on other nodes will be fine ...
- But we won't be able to get or set anything related to the cluster
- If the manager is permanently gone, we will have to do a manual repair!
- Nobody wants to do that ... so let's make our cluster highly available

Adding more managers

1. Get the token-based joining command

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

2. Launch a new node

> ```
> docker run -d --privileged --name node3 docker:dind
> ```

3. Use the command on the node

> ```
> docker exec node3 <command_here>
> ```

4. Repeat for as many managers as needed
   - (e.g. write a script to produce many nodes).
5. Similarly use `docker swarm join-token worker` and follow the same process to add additional workers if you need some.

Controlling the Swarm from other nodes

> Try the following command on a few different nodes:

```
docker node ls
```

On manager nodes:

- you will see the list of nodes, with a `*` denoting the node you're talking to.

On non-manager nodes:

- you will get an error message telling you that the node is not a manager.

As we saw earlier, you can only control the Swarm through a manager node.

Dynamically changing the role of a node

- We can change the role of a node on the fly:

  `docker node promote nodeX` → make nodeX a manager
  `docker node demote nodeX` → make nodeX a worker

> See the current list of nodes:

```
docker node ls
```

> Promote any worker node to be a manager:

```
docker node promote <node_name_or_id>
```

How many managers do we need?

- 2N+1 nodes can (and will) tolerate N failures

  (you can have an even number of managers, but there is no point)

- 1 manager = no failure

- 3 managers = 1 failure

- 5 managers = 2 failures (or 1 failure during 1 maintenance)

- 7 managers and more = now you might be overdoing it for most designs

See Docker's admin guide on node failure and datacenter redundancy

Why not have *all* nodes be managers?

- With Raft, writes have to go to (and be acknowledged by) all nodes
- Thus, it's harder to reach consensus in larger groups
- Only one manager is Leader (writable), so more managers ≠ more capacity
- Managers should be < 10ms latency from each other
- These design parameters lead us to recommended designs

What would McGyver do?

- Keep managers in one region (multi-zone/datacenter/rack)
- Groups of 3 or 5 nodes: all are managers. Beyond 5, separate out managers and workers
- Groups of 10-100 nodes: pick 5 "stable" nodes to be managers
- Groups of more than 100 nodes: watch your managers' CPU and RAM
    - 16GB memory or more, 4 CPU's or more, SSD's for Raft I/O
    - otherwise, break down your nodes in multiple smaller clusters

What's the upper limit?

- Don't know!
- Internal testing at Docker Inc.: 1000-10000 nodes is fine
  - deployed to a single cloud region
  - one of the main take-aways was *"you're gonna need a bigger manager"*
- it just works, assuming nodes have the resources
  - more nodes require manager CPU and networking; more containers require RAM
  - scheduling of large jobs (>70,000 containers) is slow

Real-life deployment methods

- Running commands manually over SSH?
  - (NO - not ideal when you have >100 nodes!)
- Using your favorite configuration management tool
- Docker for AWS
- Docker for Azure

Tasks

- Ensure you are able to launch a cluster with, say, 3 manager nodes and 5 worker nodes.
- Access the management terminals to check that the manager nodes have access to the `docker swarm` manager functionality and that they can communicate with the other nodes.
- Log in to a worker node, double check that it does not have manager permissions, then promote it to a manager and check again.