Docker Orchestration Workshop

Logistics

- Hello! I'm `jerome at docker dot com`
- Agenda:
    - 2:30pm (now-ish) workshop begins
    - 3:45pm (approx.) short coffee break
    - 5:00pm (supposedly) workshop ends
      (but we'll try to overstay if nobody kicks us out,
      and do some extra fun stuff with those interested)
- This will be FAST PACED, but DON'T PANIC!
- All the content is publicly available
  (slides, code samples, scripts)
- Experimental chat support: #docker-workshop on Freenode
  (you can connect with https://webchat.freenode.net/)

Outline (1/2)

Outline (2/2)

Pre-requirements

- Computer with network connection and SSH client
  (on Windows, get putty or Git BASH)
- GitHub account (recommended; not mandatory)
- Docker Hub account (only for Swarm hands-on section)
- Basic Docker knowledge

Exercise:

- This is the stuff you're supposed to do!
- Create GitHub and Docker Hub accounts now if needed
- Go to lisa.dckr.info to view these slides

VM environment

- Each person gets 5 VMs
- They are *your* VMs
- They'll be up until tomorrow
- You have a little card with login+password+IP addresses
- You can automatically SSH from one VM to another

Exercise:

- Log into the first VM (`node1`)
- Check that you can SSH (without password) to `node2`
- Check the version of docker with `docker version`

Note: from now on, unless instructed, **all commands must be run from the first VM, `node1`**.

Brand new versions!

- Engine 1.9.1
- Compose 1.5.2
- Swarm 1.0.1
- Machine 0.5.6

Our sample application

- Let's look at the general layout of the source code
- Each directory = 1 microservice
  - `rng` = web service generating random bytes
  - `hasher` = web service computing hash of POSTed data
  - `worker` = background process using `rng` and `hasher`
  - `webui` = web interface to watch progress

Exercise:

- Clone the repository on `node1`:

```
git clone git://github.com/jpetazzo/orchestration-workshop
```

(Bonus points for forking on GitHub and cloning your fork!)

What's this application?

- It is a DockerCoin miner!

- No, you can't buy coffee with DockerCoins

- How DockerCoins works:

  - `worker` asks to `rng` to give it random bytes
  - `worker` feeds those random bytes into `hasher`
  - each hash starting with `0` is a DockerCoin
  - DockerCoins are stored in `redis`
  - `redis` is also updated every second to track speed
  - you can see the progress with the `webui`

Next: we will inspect components independently.

Running services independently

First, we will run the random number generator (`rng`).

Exercise:

- Go to the `dockercoins` directory, in the cloned repo:
  ```
  cd orchestration-workshop/dockercoins
  ```
- Use Compose to run the `rng` service:
  ```
  docker-compose up rng
  ```
- Docker will pull `python` and build the microservice

Lies, damn lies, and port numbers

⚠️ Pay attention to the port mapping!

- The container log says:
  `Running on http://0.0.0.0:80/`

- But if you try `curl localhost:80`, you will get:
  `Connection refused`

- Port 80 on the container ≠ port 80 on the Docker host

Understanding port mapping

- `node1`, the Docker host, has only one port 80
- If we give the one and only port 80 to the first container who asks for it, we are in trouble when another container needs it
- Default behavior: containers are not "exposed"
  (only reachable by the Docker host and other containers, through their private address)
- Container network services can be exposed:
  - statically (you decide which host port to use)
  - dynamically (Docker allocates a host port)

Declaring port mapping

- Directly with the Docker Engine:

```
docker run -P redis
docker run -p 6379 redis
docker run -p 1234:6379 redis
```

- With Docker Compose, in the `docker-compose.yml` file:

```
rng:
  …
  ports:
    - "8001:80"
```

→ port 8001 *on the host* maps to port 80 *in the container*

Using the `rng` service

Let's get random bytes of data!

Exercise:

- Open a second terminal and connect to the same VM

- Check that the service is alive:
  `curl localhost:8001`

- Get 10 bytes of random data:
  `curl localhost:8001/10`

- If the binary data output messed up your terminal, fix it:
  `reset`

Running the hasher

Exercise:

- Start the `hasher` service:
  `docker-compose up hasher`

- It will pull `ruby` and do the build

⚠ Again, pay attention to the port mapping!

The container log says that it's listening on port 80, but it's mapped to port 8002 on the host.

You can see the mapping in `docker-compose.yml`.

Testing the hasher

Exercise:

- Open a third terminal window, and SSH to `node1`
- Check that the `hasher` service is alive:
  `curl localhost:8002`
- Posting binary data requires some extra flags:

```
curl \
  -H "Content-type: application/octet-stream" \
  --data-binary hello \
  localhost:8002
```

- Check that it computed the right hash:
  `echo -n hello | sha256sum`

Stopping services

We have multiple options:

- Interrupt `docker-compose up` with `^C`
- Stop individual services with `docker-compose stop rng`
- Stop all services with `docker-compose stop`
- Kill all services with `docker-compose kill` (rude, but faster!)

Exercise:

- Use any of those methods to stop `rng` and `hasher`

Running the whole app on a single node

Exercise:

- Run `docker-compose up` to start all components

- `rng` and `hasher` can be started directly
- Other components are built accordingly
- Aggregate output is shown
- Output is verbose
  (because the worker is constantly hitting other services)

Viewing our application

- The app exposes a Web UI with a realtime progress graph

Exercise:

- Open http://[yourVMaddr]:8000/ (from a browser)

- The app actually has a constant, steady speed (3.33 coins/second)

- The speed seems not-so-steady because:
  - we measure a discrete value over discrete intervals
  - the measurement is done by the browser
  - BREAKING: network latency is a thing

Running in the background

- The logs are very verbose (and won't get better)
- Let's put them in the background for now!

Exercise:

- Stop the app (with `^C`)
- Start it again with `docker-compose up -d`
- Check on the web UI that the app is still making progress

Looking at resource usage

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

Exercise:

- run `top` to see CPU and memory usage
  (you should see idle cycles)
- run `vmstat 3` to see I/O usage (si/so/bi/bo)
  (the 4 numbers should be almost zero,
  except `bo` for logging)

We have available resources.

- Why?
- How can we use them?

Scaling workers on a single node

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

Exercise:

- Start 9 more `worker` containers:
  `docker-compose scale worker=10`

- Check the aggregated logs of those containers:
  `docker-compose logs worker`

- See the impact on CPU load (with top/htop),
  and on compute speed (with web UI)

*With some limitations, as we'll see later.

Identifying bottlenecks

- You should have seen a 3x speed bump (not 10x)
- Adding workers didn't result in linear improvement
- *Something else* is slowing us down

Identifying bottlenecks

- You should have seen a 3x speed bump (not 10x)
- Adding workers didn't result in linear improvement
- *Something else* is slowing us down
- ... But what?

Identifying bottlenecks

- You should have seen a 3x speed bump (not 10x)
- Adding workers didn't result in linear improvement
- *Something else* is slowing us down
- … But what?
- The code doesn't have instrumentation
- Let's use state-of-the-art HTTP performance analysis!
  (i.e. good old tools like `ab`, `httping`…)

Measuring latency under load

We will use `httping`.

Exercise:

- Scale back the `worker` service to zero:
  `docker-compose scale worker=0`

- Open a new SSH connection and check the latency of `rng`:
  `httping localhost:8001`

- Open a new SSH conection and do the same for `hasher`:
  `httping localhost:8002`

- Keep an eye on both connections!

Latency in initial conditions

Latency for both services should be very low (~1ms).

Now add a first worker and see what happens.

Exercise:

- Create the first `worker` instance:
  `docker-compose scale worker=1`

- `hasher` should be very low (~1ms)
- `rng` should be low, with occasional spikes (10-100ms)

Latency when scaling the worker

We will add workers and see what happens.

Exercise:

- Run `docker-compose scale worker=2`
- Check latency
- Increase number of workers and repeat

What happens?

- `hasher` remains low
- `rng` spikes up until it is reaches ~(N-2)*100ms
  (when you have N workers)

Why?

Why does everything take (at least) 100ms?

Why does everything take (at least) 100ms?

`rng` code:

```
@app.route("/<int:how_many_bytes>")
def rng(how_many_bytes):
    # Simulate a little bit of delay
    time.sleep(0.1)
    return Response(
```

Why does everything take (at least) 100ms?

`rng` code:

```
@app.route("/<int:how_many_bytes>")
def rng(how_many_bytes):
    # Simulate a little bit of delay
    time.sleep(0.1)
    return Response(
```

`hasher` code:

```
post '/' do
    # Simulate a bit of delay
    sleep 0.1
    content_type 'text/plain'
```

But …

WHY?!?

Why did we sprinkle this sample app with sleeps?

- Deterministic performance
  (regardless of instance speed, CPUs, I/O…)

Why did we sprinkle this sample app with sleeps?

- Deterministic performance
  (regardless of instance speed, CPUs, I/O...)

- Actual code sleeps all the time anyway

Why did we sprinkle this sample app with sleeps?

- Deterministic performance
  (regardless of instance speed, CPUs, I/O...)

- Actual code sleeps all the time anyway

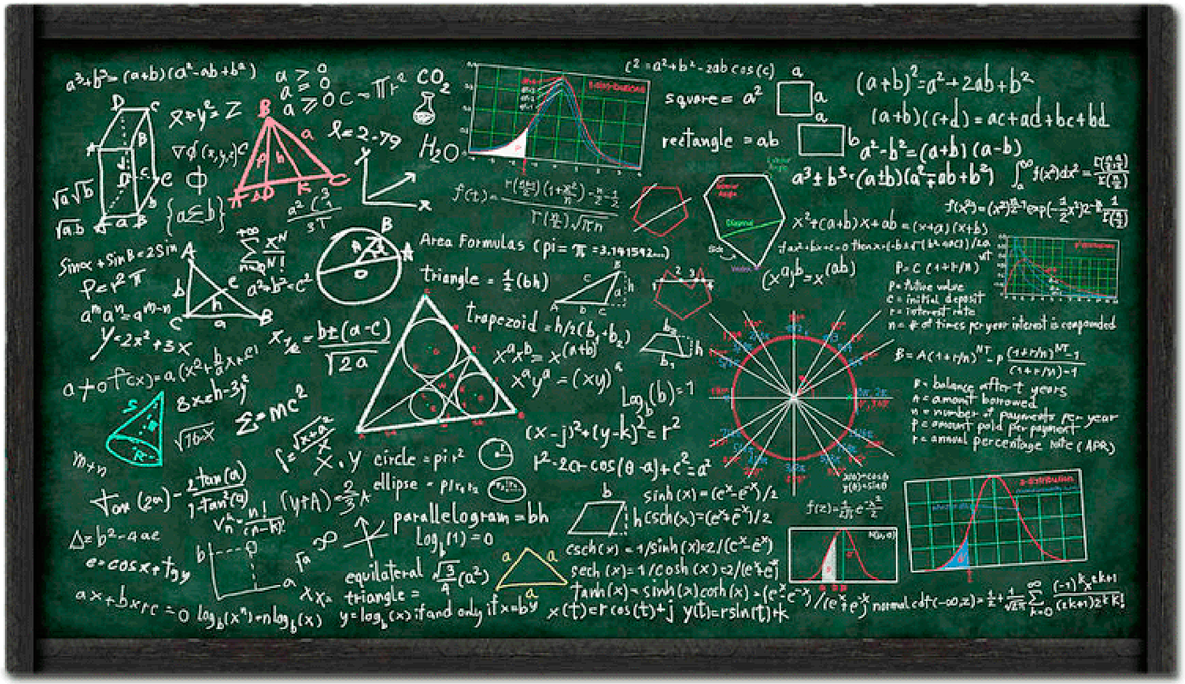
- When your code makes a remote API call:

  - it sends a request;

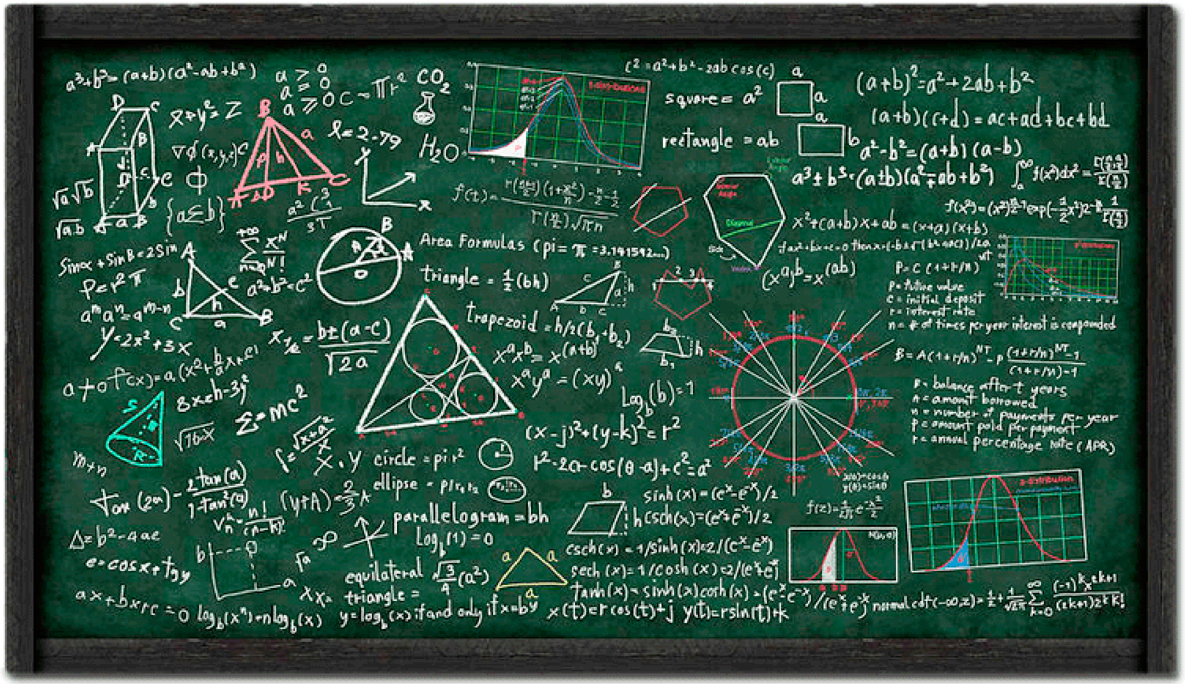
  - it sleeps until it gets the response;

  - it processes the response.

Why do `rng` and `hasher` behave differently?



Why do `rng` and `hasher` behave differently?



(Synchronous vs. asynchronous event processing)

How to make `rng` go faster

- Obvious solution: comment out the `sleep` instruction

How to make `rng` go faster

- Obvious solution: comment out the `sleep` instruction

- Real-world solution: use an asynchronous framework
  (e.g. use gunicorn with gevent)

How to make `rng` go faster

- Obvious solution: comment out the `sleep` instruction

- Real-world solution: use an asynchronous framework
  (e.g. use gunicorn with gevent)

- New rule: we can't change the code!

How to make `rng` go faster

- Obvious solution: comment out the `sleep` instruction

- Real-world solution: use an asynchronous framework
  (e.g. use gunicorn with gevent)

- New rule: we can't change the code!

- Solution: scale out `rng`
  (dispatch `rng` requests on multiple instances)

Scaling HTTP on a single node

- We could try to scale with Compose:

```
docker-compose scale rng=3
```

- Compose doesn't deal with load balancing
- We would get 3 instances ...
- ... But only the first one would serve traffic

The plan

- Stop the `rng` service first
- Create multiple identical `rng` containers
- Put a load balancer in front of them
- Point other services to the load balancer

Stopping `rng`

- That's the easy part!

Exercise:

- Use `docker-compose` to stop `rng`:

```
docker-compose stop rng
```

Note: we do this first because we are about to remove `rng` from the Docker Compose file.

If we don't stop `rng` now, it will remain up and running, with Compose being unaware of its existence!

Scaling `rng`

Exercise:

- Replace the `rng` service with multiple copies of it:

```
rng1:
  build: rng

rng2:
  build: rng

rng3:
  build: rng
```

That's all!

Shortcut: `docker-compose.yml-scaled-rng`

Introduction to `jpetazzo/hamba`

- Public image on the Docker Hub

- Load balancer based on HAProxy

- Expects the following arguments:
  `FE-port BE1-addr BE1-port BE2-addr BE2-port ...`
  or
  `FE-addr:FE-port BE1-addr BE1-port BE2-addr BE2-port ...`

  - FE=frontend (the thing other services connect to)

  - BE=backend (the multiple copies of your scaled service)

Example: listen to port 80 and balance traffic on www1:1234 + www2:2345

```
docker run -d -p 80 jpetazzo/hamba 80 www1 1234 www2 2345
```

Put a load balancer on it

Let's add our load balancer to the Compose file.

Exercise:

- Add the following section to the Compose file:

```
rng0:
    image: jpetazzo/hamba
    links:
      - rng1
      - rng2
      - rng3
    command: 80 rng1 80 rng2 80 rng3 80
    ports:
      - "8001:80"
```

Shortcut: `docker-compose.yml-scaled-rng`

Point other services to the load balancer

- The only affected service is `worker`
- We have to replace the `rng` link with a link to `rng0`, but it should still be named `rng` (so we don't change the code)

Exercise:

- Update the `worker` section as follows:

```
worker:
  build: worker
  links:
    - rng0:rng
    - hasher
    - redis
```

Shortcut: `docker-compose.yml-scaled-rng`

Start the whole stack

Exercise:

- Start the new services:
  `docker-compose up -d`

- Check worker logs:
  `docker-compose logs worker`

- Check load balancer logs:
  `docker-compose logs rng0`

If you get errors about port 8001, make sure that `rng` was stopped correctly and try again.

Results

- Check the latency of `rng`
  (it should have improved significantly!)

- Check the application performance in the Web UI
  (it should improve if you have enough workers)

Note: if `worker` was scaled when you did `docker-compose up`, it probably took a while, because `worker` doesn't handle signals properly and Docker patiently waits 10 seconds for each `worker` instance to terminate. This would be much faster for a well-behaved application.

The good, the bad, the ugly

- The good

  We scaled a service, added a load balancer -
  without changing a single line of code.

- The bad

  We manually copy-pasted sections in `docker-compose.yml`.

  Improvement: write scripts to transform the YAML file.

- The ugly

  If we scale up/down, we have to restart everything.

  Improvement: reconfigure the load balancer dynamically.

Connecting to containers on other hosts

- So far, our whole stack is on a single machine
- We want to scale out (across multiple nodes)
- We will deploy the same stack multiple times
- But we want every stack to use the same Redis
  (in other words: Redis is our only *stateful* service here)

Connecting to containers on other hosts

- So far, our whole stack is on a single machine
- We want to scale out (across multiple nodes)
- We will deploy the same stack multiple times
- But we want every stack to use the same Redis
  (in other words: Redis is our only *stateful* service here)
- And remember: we're not allowed to change the code!
  - the code connects to host `redis`
  - `redis` must resolve to the address of our Redis service
  - the Redis service must listen on the default port (6379)

Using host name injection to abstract service dependencies

- It is possible to add host entries to a container
- With the CLI:

```
docker run --add-host redis:192.168.1.2 myservice...
```

- In a Compose file:

```
myservice:
  image: myservice
  extra_host:
    redis: 192.168.1.2
```

- This creates entries in `/etc/hosts` in the container
(in Engine 1.10, a local DNS server is used instead)

Abstracting remote services with ambassadors

- What if we can't/won't run Redis on its default port?
- What if we want to be able to move it easily?

Abstracting remote services with ambassadors

- What if we can't/won't run Redis on its default port?
- What if we want to be able to move it easily?
- We will use an ambassador
- Redis will be started independently of our stack
- It will run at an arbitrary location (host+port)
- In our stack, we replace `redis` with an ambassador
- The ambassador will connect to Redis
- The ambassador will "act as" Redis in the stack

Start redis

- Start a standalone Redis container
- Let Docker expose it on a random port

Exercise:

- Run redis with a random public port:
```
docker run -d -P --name myredis redis
```
- Check which port was allocated:
```
docker port myredis 6379
```

- Note the IP address of the machine, and this port

Update `docker-compose.yml`

Exercise:

- Replace `redis` with an ambassador using `jpetazzo/hamba`:

```
redis:
  image: jpetazzo/hamba
  command: 6379 AA.BB.CC.DD EEEEE
```

- Comment out the `volumes` section in `webui`:

```
#volumes:
#  - "./webui/files/:/files/"
```

Shortcut: `docker-compose.yml-ambassador`
(But you still have to update `AA.BB.CC.DD EEEE`!)

Why did we comment out the `volumes` section?

- Volumes have multiple uses:
  - storing persistent stuff (database files...)
  - sharing files between containers (logs, configuration...)
  - sharing files between host and containers (source...)
- The `volumes` directive expands to an host path
  (e.g. `/home/docker/orchestration-workshop/dockercoins/webui/files`)
- This host path exists on the local machine
  (not on the others)
- This specific volume is used in development
  (not in production)

Start the stack on the first machine

- Compose will detect the change in the `redis` service
- It will replace `redis` with a `jpetazzo/hamba` instance

Exercise:

- Just tell Compose to to its thing:

```
docker-compose up -d
```

- Check that the stack is up and running:

```
docker-compose ps
```

- Look at the Web UI to make sure that it works fine

Start the stack on another machine

- We will set the `DOCKER_HOST` variable
- `docker-compose` will detect and use it
- Our Docker hosts are listening on port 55555

Exercise:

- Set the environment variable:
  `export DOCKER_HOST=tcp://node2:55555`
- Start the stack:
  `docker-compose up -d`
- Check that it's running:
  `docker-compose ps`

Scale!

Exercise:

- Deploy one instance of the stack on each node:

```
for N in 3 4 5; do
  DOCKER_HOST=tcp://node$N:55555 docker-compose up -d &
done
```

- Add a bunch of workers all over the place:

```
for N in 1 2 3 4 5; do
  DOCKER_HOST=tcp://node$N:55555 docker-compose scale worker=10
done
```

- Admire the result in the Web UI!

Social Media Moment

Let's celebrate our success!

(And the fact that we're just 24983498938492839489 82 DockerCoins away from being able to afford a cup of coffee!)

Exercise:

- If you have a Twitter account, tweet about your awesome Docker deployment skills!

Various considerations about ambassadors

- "But, ambassadors are adding an extra hop!"

Various considerations about ambassadors

- "But, ambassadors are adding an extra hop!"

- Yes, but if you need load balancing, you need that hop

- Ambassadors actually *save* one hop
  (they act as local load balancers)

  - traditional load balancer:
    client ⇒ external LB ⇒ server (2 physical hops)

  - ambassadors:
    client → ambassador ⇒ server (1 physical hop)

Various considerations about ambassadors

- "But, ambassadors are adding an extra hop!"
- Yes, but if you need load balancing, you need that hop
- Ambassadors actually *save* one hop
  (they act as local load balancers)
  - traditional load balancer:
    client ⇒ external LB ⇒ server (2 physical hops)
  - ambassadors:
    client → ambassador ⇒ server (1 physical hop)
- Ambassadors are more reliable than traditional LBs
  (they are colocated with their clients)

Inconvenients of ambassadors

- Generic issues
  (shared with any kind of load balancing / HA setup)
  - extra logical hop (not transparent to the client)
  - must assess backend health
  - one more thing to worry about (!)
- Specific issues
  - load balancing fairness

High-end load balancing solutions will rely on back pressure
from the backends. This addresses the fairness issue.

There are many ways to deploy ambassadors

"Ambassador" is a design pattern.

There are many ways to implement it.

We will present three increasingly complex (but also powerful) ways to deploy ambassadors.

Single-tier ambassador deployment

- One-shot configuration process
- Must be executed manually after each scaling operation
- Scans current state, updates load balancer configuration
- Pros:
  - simple, robust, no extra moving part
  - easy to customize (thanks to simple design)
  - can deal efficiently with large changes
- Cons:
  - must be executed after each scaling operation
  - harder to compose different strategies
- Example: this workshop

Two-tier ambassador deployment

- Daemon listens to Docker events API
- Reacts to container start/stop events
- Adds/removes back-ends to load balancers configuration
- Pros:
  - no extra step required when scaling up/down
- Cons:
  - extra process to run and maintain
  - deals with one event at a time (ordering matters)
- Hidden gotcha: load balancer creation
- Example: interlock

Three-tier ambassador deployment

- Daemon listens to Docker events API
- Reacts to container start/stop events
- Adds/removes scaled services in distributed config DB
  (zookeeper, etcd, consul…)
- Another daemon listens to config DB events
- Adds/removes backends to load balancers configuration
- Pros:
  - more flexibility
- Cons:
  - three extra services to run and maintain
- Example: registrator

Other multi-host communication mechanisms

- Overlay networks
    - weave, flannel, pipework ...
- Network plugins
    - available since Engine 1.9
- Allow a flat network for your containers
- Often requires an extra service to deal with BUM packets (broadcast/unknown/multicast)
    - e.g. a key/value store (Consul, Etcd, Zookeeper ...)
- Load balancers and/or failover mechanisms still needed

Interlude

Docker for ops

Backups

- Redis is still running (with name `myredis`)
- We want to enable backups without touching it
- We will use a special backup container:
  - sharing the same volumes
  - linked to it (to connect to it easily)
  - possibly containing our backup tools
- This works because the `redis` container image stores its data on a volume

Starting the backup container

Exercise:

- Make sure you're talking to the initial host:

```
unset DOCKER_HOST
```

- Start the container:

```
docker run --link myredis:redis \
           --volumes-from myredis \
           -v /tmp/myredis:/output \
           -ti alpine sh
```

- Look in `/data` in the container
  (That's where Redis puts its data dumps)

Connecting to Redis

- We need to tell Redis to perform a data dump *now*

Exercise:

- Connect to Redis:
  `telnet redis 6379`
- Issue commands `SAVE` then `QUIT`
- Look at `/data` again

- There should be a recent dump file now!

Getting the dump out of the container

- We could use many things:
  - s3cmd to copy to S3
  - SSH to copy to a remote host
  - gzip/bzip/etc before copying
- We'll just copy it to the Docker host

Exercise:

- Copy the file from `/data` to `/output`
- Exit the container
- Look into `/tmp/myredis` (on the host)

Logs

- Sorry, this part won't be hands-on
- Two strategies:
  - log to plain files on volumes
  - log to stdout
    (and use a logging driver)

Logging to plain files on volumes

- Start a container with `-v /logs`
- Make sure that all log files are in `/logs`
- To check logs, run e.g.

```
docker run --volumes-from ... ubuntu sh -c \
       "grep WARN /logs/*.log"
```

- Or just go interactive:

```
docker run --volumes-from ... -ti ubuntu
```

- You can (should) start a log shipper that way

Logging to stdout

- All containers should write to stdout/stderr
- Docker will collect logs and pass them to a logging driver
- Available drivers:
  json-file (default), syslog, journald, gelf, fluentd
- Change driver by passing `--log-driver` option to daemon
  (Better use Machine `engine-opt` for that!)
- For now, only json-files supports logs retrieval
  (i.e. `docker logs`)
- Warning: json-file doesn't rotate logs by default
  (but this can be changed with `--log-opt`)

See: https://docs.docker.com/reference/logging/overview/

Security upgrades

- This section is not hands-on
- Public Service Announcement
- We'll discuss:
    - how to upgrade the Docker daemon
    - how to upgrade container images

Upgrading the Docker daemon

- Stop all containers cleanly
  (`docker ps -q | xargs docker stop`)
- Stop the Docker daemon
- Upgrade the Docker daemon
- Start the Docker daemon
- Start all containers
- This is like upgrading your Linux kernel,
  but it will get better

Upgrading container images

- When a vulnerability is announced:
  - if it affects your base images,
    make sure they are fixed first
  - if it affects downloaded packages,
    make sure they are fixed first
  - re-pull base images
  - rebuild
  - restart containers

(The procedure is simple and plain, just follow it!)

Network traffic analysis

- We still have `myredis` running
- We will use *shared network namespaces*
  to perform network analysis
- Two containers sharing the same network namespace...
  - have the same IP addresses
  - have the same network interfaces
- `eth0` is therefore the same in both containers

Install and start `ngrep`

Ngrep uses libpcap (like tcpdump) to sniff network traffic.

Exercise:

- Start a container with the same network namespace:
```
docker run --net container:myredis -ti alpine sh
```
- Install ngrep:
```
apk update && apk add ngrep
```
- Run ngrep:
```
ngrep -tpd eth0 -Wbyline . tcp
```

You should see a stream of Redis requests and responses.

Dynamic orchestration

Static vs Dynamic

- Static
  - you decide what goes where
  - simple to describe and implement
  - seems easy at first but doesn't scale efficiently
- Dynamic
  - the system decides what goes where
  - requires extra components (HA KV...)
  - scaling can be finer-grained, more efficient

Mesos (overview)

- First presented in 2009
- Initial goal: resource scheduler
  (two-level/pessimistic)
  - top-level "master" knows the global cluster state
  - "slave" nodes report status and resources to master
  - master allocates resources to "frameworks"
- Container support added recently
  (had to fit existing model)
- Network and service discovery is complex

Mesos (in practice)

- Easy to setup a test cluster (in containers!)
- Great to accommodate mixed workloads
  (see Marathon, Chronos, Aurora, and many more)
- "Meh" if you only want to run Docker containers
- In production on clusters of thousands of nodes
- Open source project; commercial support available

Kubernetes (overview)

- 1 year old
- Designed specifically as a platform for containers
  ("greenfield" design)
  - "pods" = groups of containers sharing network/storage
  - Scaling and HA managed by "replication controllers"
  - extensive use of "tags" instead of e.g. tree hierarchy
- Initially designed around Docker,
  but doesn't hesitate to diverge in a few places

Kubernetes (in practice)

- Network and service discovery is powerful, but complex
  (different mechanisms within pod, between pods, for inbound traffic…)
- Initially designed around GCE
  (currently relies on "native" features for fast networking and persistence)
- Adaptation is needed when it differs from Docker
  (need to learn new API, new tooling, new concepts)
- Tends to be loved by ops more than devs
  (but keep in mind that it's evolving quite as fast as Docker)

Swarm (in theory)

- Consolidates multiple Docker hosts into a single one
- "Looks like" a Docker daemon, but it dispatches (schedules) your containers on multiple daemons
- Talks the Docker API front and back (leverages the Docker API and ecosystem)
- Open source and written in Go (like Docker)
- Started by two of the original Docker authors (@aluzzardi and @vieux)

Swarm (in practice)

- Stable since November 2015
- Tested with 1000 nodes + 50000 containers
  (without particular tuning; see DockerCon EU opening keynotes!)
- Perfect for some scenarios (Jenkins, grid...)
- Requires extra effort for Compose build, links...
- Requires a key/value store to achieve high availability
- We'll see it in action!

PAAS on Docker

- The PAAS workflow: *just push code*
  (inspired by Heroku, dotCloud...)

- TL,DR: easier for devs, harder for ops,
  some very opinionated choices

- A few examples:
  (Non-exhaustive list!!!)

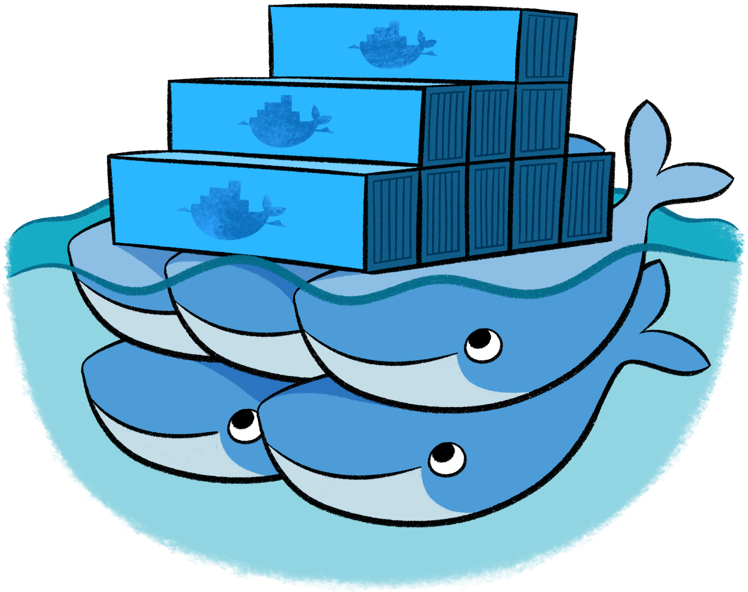
  - Cloud Foundry
  - Deis
  - Dokku
  - Flynn
  - Tsuru

A few other tools

- Volume plugins (Convoy, Flocker…)
    - manage/migrate stateful containers (and more)
- Network plugins (Contiv, Weave…)
    - overlay network so that containers can ping each other
- Powerstrip
    - sits in front of the Docker API; great for experiments
- Tutum, Docker UCP (Universal Control Plane)
    - dashboards to manage fleets of Docker hosts

… And many more!

Hands-on Swarm



Setting up our Swarm cluster

- This can be done manually or with **Docker Machine**
- Manual deployment:
    - with TLS: certificate generation is painful
      (needs dual-use certs)
    - without TLS: easier, but insecure
      (unless you run on your internal/private network)
- Docker Machine deployment:
    - generates keys, certificates, and deploys them for you
    - can also create VMs

The Way Of The Machine

- Install `docker-machine` (single binary download)
- Set a few environment variables (cloud credentials)
- Create one or more machines:
  `docker-machine create -d digitalocean node42`
- List machines and their status:
  `docker-machine ls`
- Select a machine for use:
  `eval $(docker-machine env node42)`
  (this will set a few environment variables)
- Execute regular commands with Docker, Compose, etc.
  (they will pick up remote host address from environment)

Docker Machine `generic` driver

- Most drivers work the same way:
  - use cloud API to create instance
  - connect to instance over SSH
  - install Docker
- The `generic` driver skips the first step
- It can install Docker on any machine,
  as long as you have SSH access
- We will use that!

Deploying Swarm

- Components involved:
    - service discovery mechanism
      (we'll use Docker's hosted system)
    - swarm manager
      (runs on `node1`, exposes Docker API)
    - swarm agent
      (runs on each node, registers it with service discovery)

Cluster discovery

- Possible backends:
  - dynamic, self-hosted (zk, etcd, consul)
  - static (command-line or file)
  - hosted by Docker (token)
- We will use the token mechanism

Generating our Swarm discovery token

The token is a unique identifier, corresponding to a bucket in the discovery service hosted by Docker Inc.

(You can consider it as a rendez-vous point for your cluster.)

Exercise:

- Create your token, saving it preciusly to disk as well:

```
TOKEN=$(docker run swarm create | tee token)
```

Swarm agent

- Used only for dynamic discovery (zk, etcd, consul, token)
- Must run on each node
- Every 20s (by default), tells to the discovery system:
  "Hello, there is a Swarm node at A.B.C.D:EFGH"
- Must know the node's IP address
  (sorry, it can't figure it out by itself, because
  it doesn't know whether to use public or private addresses)
- The node continues to work even if the agent dies
- Automatically started by Docker Machine
  (when the `--swarm` option is passed)

Swarm manager

- Today: must run on the leader node
- Later: can run on multiple nodes, with leader election
- Automatically started by Docker Machine
  (when the `--swarm-master` option is passed)

Exercise:

- Connect to `node1`
- "Create" a node with Docker Machine

```
docker-machine create --driver generic \
  --swarm --swarm-master --swarm-discovery token://$TOKEN \
  --generic-ssh-user docker --generic-ip-address 1.2.3.4 node1
```

(Don't forget to replace 1.2.3.4 with the node IP address!)

Check our node

Let's connect to the node *individually*.

Exercise:

- Select the node with Machine

```
eval $(docker-machine env node1)
```

- Execute some Docker commands

```
docker version
docker info
docker ps
```

Two containers should show up: the agent and the manager.

Check our (single-node) Swarm cluster

Let's connect to the manager instead.

Exercise:

- Select the Swarm manager with Machine

```
eval $(docker-machine env node1 --swarm)
```

- Execute some Docker commands

```
docker version
docker info
docker ps
```

The output is different! Let's review this.

`docker version`

Swarm identifies itself clearly:

```
Client:
 Version:      1.9.1
 API version:  1.21
 Go version:   go1.4.2
 Git commit:   a34a1d5
 Built:        Fri Nov 20 13:20:08 UTC 2015
 OS/Arch:      linux/amd64

Server:
 Version:      swarm/1.0.1
 API version:  1.21
 Go version:   go1.5.2
 Git commit:   744e3a3
 Built:
 OS/Arch:      linux/amd64
```

docker info

Swarm gives cluster information, showing all nodes:

```
Containers: 3
Images: 6
Role: primary
Strategy: spread
Filters: affinity, health, constraint, port, dependency
Nodes: 1
 node: 52.89.117.68:2376
   └ Containers: 3
   └ Reserved CPUs: 0 / 2
   └ Reserved Memory: 0 B / 3.86 GiB
   └ Labels: executiondriver=native-0.2,
            kernelversion=3.13.0-53-generic,
            operatingsystem=Ubuntu 14.04.2 LTS,
            provider=generic, storagedriver=aufs
CPUs: 2
Total Memory: 3.86 GiB
Name: 2ec2e6c4054e
```

`docker ps`

- This one should show nothing at this point.
- The Swarm containers are hidden.
- This avoids unneeded pollution.
- This also avoids killing them by mistake.

Add other nodes to the cluster

- Let's use *almost* the same command line
  (but without `--swarm-master`)

Exercise:

- Stay on `node1` (it has keys and certificates now!)
- Add another node with Docker Machine

```
docker-machine create --driver generic \
  --swarm --swarm-discovery token://$TOKEN \
  --generic-ssh-user docker --generic-ip-address 1.2.3.4 node2
```

Remember to update the IP address correctly.

Repeat for all 4 nodes.

Pro tip: look for name/address mapping in `/etc/hosts`.

Scripting

To help you a little bit:

```
grep node[2345] /etc/hosts | grep -v ^127 |
while read IPADDR NODENAME
do docker-machine create --driver generic \
   --swarm --swarm-discovery token://$TOKEN \
   --generic-ssh-user docker \
   --generic-ip-address $IPADDR $NODENAME
done
```

Running containers on Swarm

Try to run a few `busybox` containers.

Then, let's get serious:

Exercise:

- Start a Redis service:
  ```
  docker run -dP redis
  ```
- See the service address:
  ```
  docker port $(docker ps -lq) 6379
  ```

This can be any of your five nodes.

Building our app on Swarm

Before trying to build our app, we will remove previous images.

Exercise:

- Delete all images with "dockercoins" in the name:

```
docker images |
  grep dockercoins |
  awk '{print $3}' |
  xargs -r docker rmi -f
```

Building our app on Swarm

- Swarm has partial support for builds
- ⚠ Older versions of Compose would crash on builds

Exercise:

- Run `docker-compose build` multiple times
  (until you get it to build twice)
- Loudly complain that caching doesn't work as expected!
- Run one container multiple times with a resource limit:
  `docker run -d -m 1G dockercoins_rng`
- Check where the containers are running with `docker ps`

Caveats when building with Swarm

- Caching doesn't work all the time
  - cause: build nodes can be picked randomly
  - solution: always pin builds to the same node
- Containers are only scheduled on a few nodes
  - cause: images are not present on all nodes
  - solution: distribute images through a registry
    (e.g. Docker Hub)

Why can't Swarm do this automatically for us?

- Let's step back and think for a minute ...
- What should `docker build` do on Swarm?
    - build on one machine
    - build everywhere ($$$)
- After the build, what should `docker run` do?
    - run where we built (how do we know where it is?)
    - run on any machine that has the image
- Could Compose+Swarm solve this automatically?

A few words about "sane defaults"

- *It would be nice if Swarm could pick a node, and build there!*
  - but which node should it pick?
  - what if the build is very expensive?
  - what if we want to distribute the build across nodes?
  - what if we want to tag some builder nodes?
  - ok but what if no node has been tagged?
- *It would be nice if Swarm could automatically push images!*
  - using the Docker Hub is an easy choice
    (you just need an account)
  - but some of us can't/won't use Docker Hub
    (for compliance reasons or because no network access)

("Sane" defaults are nice only if we agree on the definition of "sane")

The plan

- Build locally

- Tag images

- Upload them to the hub
  (Note: this part requires a Docker Hub account!)

- Update the Compose file to use those images

That's the purpose of the `build-tag-push.py` script!

Docker Hub account

- You need a Docker Hub account for that part
- If you don't have one, create it

Exercise:

- Set the following environment variable:

```
export DOCKERHUB_USER=jpetazzo
```

- (Use *your* Docker Hub login, of course!)
- Log into the Docker Hub:

```
docker login
```

Build, Tag, And Push

Let's inspect the source code of `build-tag-push.py` and run it.

⚠ It is better to run it against a single node!

(There are some race conditions within Swarm when building+pushing too fast.)

Exercise:

- Point to a single node:
  ```
  eval $(docker-machine env node1)
  ```

- Run the script (from the `dockercoins` directory):
  ```
  ../build-tag-push.py
  ```

- Inspect the `docker-compose.yml-XXX` file that it created

Can we run this now?

Let's try!

Exercise:

- Switch back to the Swarm cluster:
  `eval $(docker-machine env node1 --swarm)`

- Protip - set the `COMPOSE_FILE` variable:
  `export COMPOSE_FILE=docker-compose.yml-XXX`

- Bring up the application:
  `docker-compose up`

Can we run this now?

Let's try!

Exercise:

- Switch back to the Swarm cluster:
  `eval $(docker-machine env node1 --swarm)`
- Protip - set the `COMPOSE_FILE` variable:
  `export COMPOSE_FILE=docker-compose.yml-XXX`
- Bring up the application:
  `docker-compose up`

It won't work, because Compose and Swarm do not collaborate to establish *placement constraints*.

Can we run this now?

Let's try!

Exercise:

- Switch back to the Swarm cluster:
  ```
  eval $(docker-machine env node1 --swarm)
  ```
- Protip - set the `COMPOSE_FILE` variable:
  ```
  export COMPOSE_FILE=docker-compose.yml-XXX
  ```
- Bring up the application:
  ```
  docker-compose up
  ```

It won't work, because Compose and Swarm do not collaborate to establish *placement constraints*.

(╯°□°)╯︵ ┻━┻

Simple container dependencies

- Container A has a link to container B
- Compose starts B first, then A
- Swarm translates the link into a placement constraint:
  - *"put A on the same node as B"*
- Alles gut

Complex container dependencies

- Container A has a link to containers B and C
- Compose starts B and C first
  (but that can be on different nodes!)
- Compose starts A
- Swarm translates the links into placements contraints
  - *"put A on the same node as B"*
  - *"put A on the same node as C"*
- If B and C are on different nodes, that's impossible

So, what do?

A word on placement constraints

- Swarm supports constraints
- We could tell swarm to put all our containers together
- Linking would work
- But all containers would end up on the same node

A word on placement constraints

- Swarm supports constraints
- We could tell swarm to put all our containers together
- Linking would work
- But all containers would end up on the same node
- So having a cluster would be pointless!

Network plumbing on Swarm

- We will use one-tier, dynamic ambassadors
  (as seen before)

- Other available options:

  - injecting service addresses in environment variables

  - implementing service discovery in the application

  - use an overlay network

Revisiting `jpetazzo/hamba`

- Configuration is stored in a *volume*
- A watcher process looks for configuration updates,
  and restarts HAProxy when needed
- It can be started without configuration:

```
docker run --name amba jpetazzo/hamba run
```

- There is a helper to inject a new configuration:

```
docker run --rm --volumes-from amba jpetazzo/hamba \
       reconfigure 80 backend1 port1 backend2 port2 ...
```

Note: configuration validation and error messages will be logged by the ambassador, not the `reconfigure` container.

Should we use `links` for our ambassadors?

Technically, we could use links.

- Before starting an app container:

  start the ambassador(s) it needs

- When starting an app container:

  link it to its ambassador(s)

But we wouldn't be able to use `docker-compose scale` anymore.

Network namespaces and `extra_hosts`

This is our plan:

- Replace each `link` with an `extra_host`,
  pointing to the `127.127.X.X` address space

- Start app containers normally
  (`docker-compose up`, `docker-compose scale`)

- Start ambassadors after app containers are up:

  - ambassadors bind to `127.127.X.X`

  - they share their client's network namespace

- Reconfigure ambassadors each time something changes

Our plan for service discovery

- Replace all `links` with static `/etc/hosts` entries
- Those entries will map to `127.127.0.X`
  (with different `X` for each service)
- Example: `redis` will point to `127.127.0.2`
  (instead of a container address)
- Start all services; scale them if we want
  (at this point, they will all fail to connect)
- Start ambassadors in the services' namespace;
  each ambassador will listen on the right `127.127.0.X`
- Gather all backend addresses and configure ambassadors

⚠ Services should try to reconnect!

"Design for failure," they said

- When the containers are started, the network is not ready
- First connection attempts **will fail**
- App should try to reconnect
- It is OK to crash and restart
- Exponential back-off is nice

Our tools

- `link-to-ambassadors.py`
  - replaces all `links` with `extra_hosts` entries
- `create-ambassadors.py`
  - scans running containers
  - allocates `127.127.X.X` addresses
  - starts (unconfigured) ambassadors
- `configure-ambassadors.py`
  - scans running containers
  - gathers backend addresses
  - sends configuration to ambassadors

Convert links to ambassadors

- When we ran `build-tag-push.py` earlier,
  it generated a new `docker-compose.yml-XXX` file.

Exercise:

- Run the first script to create a new YAML file:

```
../link-to-ambassadors.py $COMPOSE_FILE new.yml
```

- Look how the file was modified:

```
diff $COMPOSE_FILE new.yml
```

Change `$COMPOSE_FILE` in place

The script can take zero, one, or two file name arguments:

- two arguments indicate input and output files to use;
- with one argument, the file will be modified in place;
- with zero agument, it will act on `$COMPOSE_FILE`.

For convenience, let's avoid having a bazillion files around.

Exercise:

- Remove the temporary Compose file we just created:
```
rm -f new.yml
```

- Update `$COMPOSE_FILE` in place:
```
../link-to-ambassadors.py
```

Bring up the application

The application can now be started and scaled.

Exercise:

- Start the application:
  `docker-compose up -d`

- Scale the application:
  `docker-compose scale worker=5 rng=10`

Note: you can scale everything as you like, *except Redis*, because it is stateful.

Create the ambassadors

This has to be executed each time you create new services or scale up existing ones.

After reading `$COMPOSE_FILE`, it will scan running containers, and compare:

- the list of app containers,
- the list of ambassadors.

It will create missing ambassadors.

Exercise:

- Run the script!
  ```
  ../create-ambassadors.py
  ```

Configure the ambassadors

All ambassadors are created but they still need configuration.

That's the purpose of the last script.

It will read `$COMPOSE_FILE` and gather:

- the list of app backends,
- the list of ambassadors.

Then it configures all ambassadors with all found backends.

Exercise:

- Run it!

```
../configure-ambassadors.py
```

Check what we did

Exercise:

- Find out the address of the web UI:
  `docker-compose ps webui`
- Point your browser to it
- Check the logs:
  `docker-compose logs`

Going further

Scaling the application (difficulty: easy)

- Run `docker-compose scale`
- Re-create ambassadors
- Re-configure ambassadors
- No downtime

Going further

Deploying a new version (difficulty: easy)

- Just re-run all the steps!
- However, Compose will re-create the containers
- You will have to re-create ambassadors
  (and configure them)
- You will have to cleanup old ambassadors
  (left as an exercise for the reader)
- You will experience a little bit of downtime

Going further

Zero-downtime deployment (difficulty: medium)

- Isolate stateful services
  (like we did earlier for Redis)

- Do blue/green deployment:

  - deploy and scale version N

  - point a "top-level" load balancer to the app

  - deploy and scale version N+1

  - put both apps in the "top-level" balancer

  - slowly switch traffic over to app version N+1

Going further

Use the new networking features (difficulty: medium)

- Create a key/value store (e.g. Consul cluster)
- Reconfigure all Engines to use the key/value store
- Load balancers can use DNS for backend discovery

Note: this is really easy to do with a 1-node Consul cluster.

Going further

Harder projects:

- Two-tier or three-tier ambassador deployments
- Deploy to Mesos or Kubernetes

Here be dragons!

Here be dragons

- So far, we've used stable products (versions 1.X)
- We're going to explore experimental software
- **Use at your own risk**

Setting up Consul and overlay networks

- We will reconfigure our Swarm cluster to enable overlays
- We will deploy a Consul cluster
- We will connect containers running on different machines

First, let's Clean All The Things!

- We need to remove the old containers
  (in particular the `swarm` agents and managers)

Exercise:

- The following snippet will nuke all containers on all hosts:

```
for N in 1 2 3 4 5
do
  ssh node$N "docker ps -qa | xargs -r docker rm -f"
done
```

(If it asks you to confirm SSH keys, just do it!)

Note: our Swarm cluster is now broken.

Remove old Machine information

- We will use `docker-machine rm`
- With the `generic` driver, this doesn't do anything
  (it just deletes local configuration)
- With cloud/VM drivers, this would actually delete VMs

Exercise:

- Remove our nodes from Docker Machine config database:

```
for N in 1 2 3 4 5
do
  docker-machine rm -f node$N
done
```

Add extra options to our Engines

- We need two new options for our engines:
  - `cluster-store` (to indicate which key/value store to use)
  - `cluster-advertise` (to indicate which IP address to register)
- `cluster-store` will be `consul://localhost:8500` (we will run one Consul node on each machine)
- `cluster-advertise` will be `eth0:2376` (Engine will automatically pick up eth0's IP address)

Reconfiguring Swarm clusters, the Docker way

- The traditional way to reconfigure a service is to edit its configuration (or init script), then restart
- We can use Machine to make that easier
- Re-deploying with Machine's `generic` driver will reconfigure Engines with the new parameters

Exercise:

- Re-provision the manager node:

```
docker-machine create --driver generic \
  --engine-opt cluster-store=consul://localhost:8500 \
  --engine-opt cluster-advertise=eth0:2376 \
  --swarm --swarm-master --swarm-discovery consul://localhost:8500 \
  --generic-ssh-user docker --generic-ip-address 52.32.216.30 node1
```

Reconfigure the other nodes

- Once again, scripting to the rescue!

Exercise:

```
grep node[2345] /etc/hosts | grep -v ^127 |
while read IPADDR NODENAME
do docker-machine create --driver generic \
   --engine-opt cluster-store=consul://localhost:8500 \
   --engine-opt cluster-advertise=eth0:2376 \
   --swarm --swarm-discovery consul://localhost:8500 \
   --generic-ssh-user docker \
   --generic-ip-address $IPADDR $NODENAME
done
```

Checking what we did

Exercise:

- Directly point the CLI to a node and check configuration:

```
eval $(docker-machine env node1)
docker info
```

  (should show `Cluster store` and `Cluster advertise`)

- Try to talk to the Swarm cluster:

```
eval $(docker-machine env node1 --swarm)
docker info
```

  (should show zero node)

Why zero node?

- We haven't started Consul yet
- Swarm discovery is not operationl
- Swarm can't discover the nodes

Note: good guy ~~Stevedore~~ Docker will start without K/V

(This lets us run Consul itself in a container!)

Adding Consul

- We will run Consul in containers
- We will use awesome Jeff Linday's awesome consul image
- We will tell Docker to automatically restart it on reboots
- To simplify network setup, we will use `host` networking

Starting the first Consul node

Exercise:

- Log into `node1`
- The first node must be started with the `-bootstrap` flag:

```
CID=$(docker run --name consul_node1 \
      -d --restart=always --net host \
      progrium/consul -server -bootstrap)
```

- Find the internal IP address of that node
  With This One Weird Trick:

```
IPADDR=$(docker run --rm --net container:$CID alpine \
         ip a ls dev eth0 |
         sed -n 's,.*inet \(.*\)/.*,\1,p')
```

Starting the other Consul nodes

Exercise:

- The other nodes have to be startd with the `-join IP.AD.DR.ESS` flag:

```
for N in 2 3 4 5; do
ssh node$N docker run --name consul_node$N \
             -d --restart=always --net host \
             progrium/consul -server -join $IPADDR
done
```

- With your browser, navigate to any instance on port 8500 (in "NODES" you should see the five nodes)

Check that our Swarm cluster is up

Exercise:

- Try again the `docker info` from earlier:

```
eval $(docker-machine env --swarm node1)
docker info
```

- Now all nodes should be visible
  (Give them a minute or two to register)

Multi-host networking

- Docker 1.9 has the concept of *networks*
- By default, containers are on the default "bridge" network
- You can create additional networks
- Containers can be on multiple networks
- Containers can dynamically join/leave networks
- The "overlay" driver lets networks span multiple hosts
- Let's see that in action!

Create a few networks and containers

Exercise:

```
docker network create --driver overlay jedi
docker network create --driver overlay darkside
docker network ls
```

Create a few networks and containers

Exercise:

```
docker network create --driver overlay jedi
docker network create --driver overlay darkside
docker network ls
```

(Don't worry, there won't be any spoiler here, I have been so busy preparing this workshop that I haven't seen the new movie yet!)

Create a few networks and containers

Exercise:

```
docker network create --driver overlay jedi
docker network create --driver overlay darkside
docker network ls
```

(Don't worry, there won't be any spoiler here, I have been so busy preparing this workshop that I haven't seen the new movie yet!)

Exercise:

```
docker run -d --name luke --net jedi -m 3G redis
docker run -d --name vador --net jedi -m 3G redis
docker run -d --name palpatine --net darkside -m 3G redis
```

Check connectivity within networks

Exercise:

- Check that our containers are on different networks:

```
docker ps
```

- This will work:

```
docker exec -ti vador ping luke
```

- This will not:

```
docker exec -ti vador ping palpatine
```

Dynamically connect containers

Exercise:

- ~~Connect `vador` to the `darkside`:~~
- To the `darkside`, connect `vador` we must:

```
docker network connect darkside vador
```

- Now this will work:

```
docker exec -ti vador ping palpatine
```

- Take a peek inside `vador`:

```
docker exec -ti vador ip addr ls
```

Dynamically disconnecting containers

Exercise:

- This works, right:

```
docker exec -ti vador ping luke
```

- Let's disconnect `vador` from the `jedi` ~~order~~ network:

```
docker network disconnect jedi vador
```

- And now:

```
docker exec -ti vador ping luke
```

Cleaning up

Exercise:

- Destroy containers:

```
docker rm -f luke vador palpatine
```

- Destroy networks:

```
docker network rm jedi
docker network rm darkside
```

Compose and multi-host networking

⚠ Here be 7-headed flame-throwing hydras!

- This is super experimental
- Your cluster is likely to blow up to bits
- Situation is much better in Engine 1.10 and Compose 1.6 (currently in RC; to be released circa February 2016!)

Revisiting DockerCoins

Exercise:

- Go back to the `dockercoins` app:

```
cd ~/orchestration-workshop/dockercoins
```

- Re-execute `build-tag-push` to get a fresh Compose file:

```
eval $(docker-machine env -u)
../build-tag-push.py
export COMPOSE_FILE=docker-compose.yml-XXX
```

Add `container_name` to Compose file

Exercise:

- Edit the Compose file
- In the `hasher`, `rng`, and `redis` sections, add:
  `container_name: XXX`
  (where XXX is the name of the section)
- Also, comment out the `volumes` section

Note: by default, containers will be named `dockercoins_XXX_1` (instead of `XXX`) and links will not work.

This is no longer necessary with Compose 1.6!

Run the app

Exercise:

- Add two custom experimental flags:

```
docker-compose \
    --x-networking --x-network-driver=overlay \
    up -d
```

- Check the `webui` endpoint address:

```
docker-compose ps webui
```

- Go to the webui with your browser!

Scale the app

Exercise:

- Don't forget the custom experimental flags:

```
docker-compose \
    --x-networking --x-network-driver=overlay \
    scale worker=2
```

- Look at the graph in your browser

Note: with Compose 1.6 and Engine 1.10, you can have multiple containers with the same DNS name, thus achieving "natural" load balancing through DNS round robin.

Cleaning up

Exercise:

- Terminate containers and remove them:

```
docker-compose kill
docker-compose rm -f
```

Note: Compose 1.5 doesn't support changes to an existing app (except basic scaling).

When trying to do `docker-compose -x-... up` on existing apps, you might get errors like this one:

```
ERROR: unable to find a node that satisfies container==38aac...
```

If that happens, just kill+rm the app and try again.

A new hope

- Compose 1.5 + Engine 1.9 =
  first release with multi-host networking

- Compose 1.6 + Engine 1.10 =
  HUGE improvements

- I will deliver this workshop about twice a month

- Check out the GitHub repo for updated content!

Thanks!
Questions?

@jpetazzo
@docker