Docker Orchestration Workshop
Outline (1/2)

- Pre-requirements
- VM environment
- Our sample application
- Running the whole app on a single node
- Finding bottlenecks
- Scaling workers on a single node
- Scaling HTTP on a single node
- Connecting to containers on other hosts
- Abstracting connection details
Outline (2/2)

- Backups
- Logs
- Security upgrades
- Network traffic analysis
- Dynamic orchestration
- Introducing Mesos, Kubernetes, Swarm
- PAAS and other tools
- Setting up our Swarm cluster
- Running on Swarm
- Network plumbing on Swarm
Pre-requirements

- Computer with network connection and SSH client
  (on Windows, get putty or Git BASH)
- GitHub account
- Docker Hub account
- Basic Docker knowledge

Exercise:

- This is the stuff you're supposed to do!
- Create GitHub and Docker Hub accounts now if needed
- Go to view.dckr.info to view those 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
• 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 have to be done from the first VM, node1.
Brand new versions!

- Docker 1.7
- Compose 1.3.2
- Swarm 0.3
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:

- Fork the repository on GitHub
- Clone your fork on `node1`
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
  - you can see the progress with the webui

Next: we will inspect components independently.
Running components independently

Exercise:

- Go to the dockercoins directory (in the cloned repo)
- Run docker-compose up rng
  (Docker will pull python and build the microservice)

⚠️ The container log says Running on http://0.0.0.0:80/
but that is port 80 in the container. On the host it is 8001.

This is mapped in The docker-compose.yml file:

```yaml
rng:
  ...
  ports:
    - "8001:80"
```
Getting random bytes of data

❖ Exercise:

- Open a second terminal and connect to the same VM
- Check that the service is alive:
  ```bash
curl localhost:8001
  ```
- Get 10 bytes of random data:
  ```bash
curl localhost:8001/10
```
  (the output might confuse your terminal, since this is binary data)
- Test the performance on one big request:
  ```bash
curl -o/dev/null localhost:8001/10000000
```
  (should take ~1s, and show speed of ~10 MB/s)

Next: we'll see how it behaves with many small requests.
Concurrent requests

Exercise:

- Test 100 requests of 1000 bytes each:
  \texttt{ab -n 100 localhost:8001/1000}
- Test 100 requests, 10 requests in parallel:
  \texttt{ab -n 100 -c 10 localhost:8001/1000}
  (look how the latency has increased!)
- Try with 100 requests in parallel:
  \texttt{ab -n 100 -c 100 localhost:8001/1000}

Take note of the number of requests/s.
Save some random data and stop the generator

Before testing the hasher, let's save some random data that we will feed to the hasher later.

Exercise:

- Run `curl localhost:8001/1000000 > /tmp/random`

Now we can stop the generator.

Exercise:

- In the shell where you did `docker-compose up rng`, stop it by hitting `^C`
Running the hasher

Exercise:

- Run `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:

• Run `curl localhost:8002`
  (it will say it's alive)

• Posting binary data requires some extra flags:

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

• Compute the hash locally to verify that it works fine:
  `sha256sum /tmp/random`
  (it should display the same hash)
Benchmarking the hasher

The invocation of `ab` will be slightly more complex as well.

Exercise:

- Execute 100 requests in a row:
  
  ```bash
  ab -n 100 -T application/octet-stream \ 
  -p /tmp/random localhost:8002/
  ```

- Execute 100 requests with 10 requests in parallel:
  
  ```bash
  ab -c 10 -n 100 -T application/octet-stream \ 
  -p /tmp/random localhost:8002/
  ```

Take note of the performance numbers (requests/s).
Benchmarking the hasher on smaller data

Here we hashed 1 meg.

Later we will hash much smaller payloads.

Let's repeat the tests with smaller data.

Exercise:

- Run `truncate --size=10 /tmp/random`
- Repeat the ab tests
Why do rng and hasher behave differently?
Running the whole app on a single node

Exercise:

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

- Aggregate output is shown

- Output is verbose because the worker is constantly hitting other services

- Now let's use the little web UI to see realtime progress

Exercise:

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

- 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?
Measuring performance

- The code doesn't have instrumentation
- Let's use `ab` and `httping` to view latency of microservices

Exercise:
- Start two new SSH connections
- In the first one, let run `httping localhost:8001`
- In the other one, let run `httping localhost:8002`
Scaling workers on a single node

- Docker Compose supports scaling
- It doesn't deal with load balancing
- For services that *do not* accept connections, that's OK
- Let's scale `worker` and see what happens!

Exercise:

- In one SSH session, run `docker-compose logs worker`
- In another, run `docker-compose scale worker=4`
- See the impact on CPU load (with `top/htop`), and on compute speed (with web UI)
Scaling HTTP on a single node

The plan:

- Stop the rng service first
- Scale rng to multiple containers
- Put a load balancer in front of it
- Point other services to the load balancer

Note: Compose does not support that kind of scaling yet. We will have to do it manually for now.
Stopping \texttt{rng}

- That's the easy part!

Exercise:

- Use \texttt{docker-compose} to stop \texttt{rng}:

  \begin{verbatim}
  docker-compose stop rng
  \end{verbatim}

Note: we do this first because we are about to remove \texttt{rng} from the Docker Compose file.

If we don't stop \texttt{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!
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

```bash
docker run -d -p 80 jpetazzo/hamba 80 www1 1234 www2 2345
```
Add our load balancer to the Compose file

Exercise:

- Add the following section to the Compose file:

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

```yaml
worker:
  build: worker
  links:
    - rng0:rng
    - hasher
    - redis
```
Start the whole stack

- The new \texttt{rng0} load balancer also ties up port 8001
- We have to stop the old \texttt{rng} service first (Compose doesn't do it for us)

Exercise:

- Run \texttt{docker-compose stop rng}

Now (re-)start the whole stack

Exercise:

- Run \texttt{docker-compose up -d}
- Check worker logs with \texttt{docker-compose logs worker}
- Check load balancer logs with \texttt{docker-compose logs rng0}
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

• The ugly

  If we scale up/down, we have to restart everything
Ideas to improve the situation

- Parse `docker-compose.yml` to automatically replace services with their scaled counterparts
- Replace Docker Links with network namespace sharing
- More on this later
Connecting to containers on other hosts

- We want to scale 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)
The plan

- Deploy our Redis service separately
  - use the same redis image
  - make sure that Redis server port (6379) is publicly accessible, using port 6379 on the Docker host
- Update our Docker Compose YAML file
  - remove the redis section
  - in the links section, remove redis
  - instead, put a redis entry in extra_hosts
Deploy Redis

Exercise:

- Start a new redis container, mapping port 6379 to 6379:
  
  ```
  docker run -d -p 6379:6379 redis
  ```

- Check that it's running with `docker ps`

- Note the IP address of this Docker host

- Try to connect to it (from anywhere):

  ```
  telnet ip.ad.dr.ess 6379
  ```

To exit a telnet session: `Ctrl-] c` ENTER
Update docker-compose.yml (1/3)

Exercise:

- Comment out redis:

```yaml
# redis:
#   image: redis
```
Update docker-compose.yml (2/3)

Exercise:

• Update worker:

```yaml
worker:
  build: worker
  extra_hosts:
    - redis: A.B.C.D
  links:
    - rng
    - hasher
```

(Replace A.B.C.D with the IP address noted earlier)
Update docker-compose.yml (3/3)

Exercise:

- Update webui:

```yaml
webui:
  build: webui
  extra_hosts:
    - redis: A.B.C.D
  ports:
    - "8000:80"
  # volumes:
  #   - "webui/files/::files/"
```

(Replace A.B.C.D with the IP address noted earlier)

Note: we also commented out the volumes section since those files are only present locally, not on the remote nodes.
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://X.Y.Z:55555
  ```
- Start the stack:
  ```
  docker-compose up -d
  ```
- Check that it's running:
  ```
  docker-compose ps
  ```
Scale!

Exercise:

- Open the Web UI (on a node where it's deployed)
- Deploy one instance of the stack on each node
Cleanup

- Let's remove what we did

Exercise:

- You can use the following scriptlet:

```bash
for N in $(seq 1 5); do
    export DOCKER_HOST=tcp://node$N:55555
    docker ps -qa | xargs docker rm -f
done
unset DOCKER_HOST
```
Abstracting connection details

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

- What if we can't/won't run Redis on its default port?
- What if we want to be able to move it more easily?
- We will use an ambassador
- Redis will run at an arbitrary location (host+port)
- The ambassador will be part of the scaled stack
- The ambassador will connect to Redis
- The ambassador will "act as" Redis in the stack
Start redis

- This time, we will let Docker pick the port for Redis

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 this IP address and port
Update `docker-compose.yml`

Exercise:

- Restore links as they were before in `webui` and `worker`
- Replace `redis` with an ambassador using `jpetazzo/hamba`:

```yaml
redis:
  image: jpetazzo/hamba
  command: 6379 AA.BB.CC.DD EEEEE
```
Start the new stack

Exercise:

- Run `docker-compose up -d`
- Go to the web UI
- Start the stack on another node as previously, and confirm on the web UI that it's picking up
Discussion

- **jpetazzo/hamba** is simple and stupid
- It could be replaced with something dynamic:
  - looking up the host+port in consul/etcd/zk
  - reconfiguring itself when consul/etcd/zk is updated
  - dealing with failover
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:

- Start the container:

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

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

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

Exercise:

- `apt-get install telnet`
- `telnet redis 6379`
- `issue 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 (and a half) strategies:
  - log to plain files on volumes
  - log to stdout with the syslog driver
  - log to stdout with the JSON driver
- The last one doesn't really count
  (but it's the default)
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 syslog

- All containers should write to stdout/stderr
- Change Docker start options to add `--log-driver syslog` (On Ubuntu, tweak `DOCKER_OPTS` in `/etc/default/docker`)
- When you do that, you can't use `docker logs` anymore
Logging to JSON files

- That's the default option
- All containers should write to stdout/stderr
- You can use `docker logs`
- But those local JSON files are, well, local
- ... And they will eventually use up all the space
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

Exercise:

- Start a container with the same network namespace:
  docker run --net container:myredis -ti ubuntu

- Install ngrep:
  apt-get update && apt-get install -y ngrep

- Run ngrep:
  ngrep -tpd eth0 -Wbyline . tcp
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)

- Great deployment platform ...
  but no developer experience yet
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)

- Not stable yet (version 0.3 right now)
- OK for some scenarios (Jenkins, grid...)
- Not OK (yet) for Compose build, links...
- We'll see it (briefly) 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

- Flocker
  - manage/migrate stateful containers
- Powerstrip
  - sits in front of the Docker API
  - great to implement your own experiments
- Weave
  - overlay network so that containers can ping each other

... And many more!
Here be dragons!
Warning: 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
Introducing Swarm
Setting up our Swarm cluster

- This is usually done by Docker Machine (or by custom deployment scripts)
- We will do a simplified version here (without TLS), to give you an idea of what's involved
- Components involved:
  - service discovery mechanism (we'll use Docker's hosted system)
  - swarm agent (runs on each node, registers it with service discovery)
  - swarm manager (runs on node1, exposes Docker API)
Service discovery

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

- We will use the token mechanism

Exercise:

- Run `docker run swarm create`
- Save the output carefully: it's your token
  (it's the unique identifier for your cluster)
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"
- The node continues to work even if the agent dies
Join the cluster

Exercise:

- Connect to node2
- Start the swarm agent:
  docker run -d swarm join \
  --advertise A.B.C.D:55555 token://XXX
  (A.B.C.D is the IP address of node2, XXX is the token generated earlier)
- Check that the node registered successfully:
  docker swarm list token://XXX
- Repeat on nodes 3, 4, 5

Note: the Docker daemon on your VMs listens on port 55555
Swarm manager

- Today: must run on the "master" node
- Later: can run on multiple nodes, with master election

Exercise:

- Connect to node1
- Start the swarm manager:
  docker run -d -p 10000:2375 swarm manager token://XXX

- Remember to replace XXX with your token!
- The Swarm manager listens on port 2375
- We're telling Docker to expose that on port 10000
First contact with Swarm

- We must setup our CLI to talk to the Swarm master

Exercise:

- From any machine, set the environment variable:
  ```
  export DOCKER_HOST=tcp://node1:10000
  ```

- Check the output of `docker version` and `docker info`

- Remember to set the environment variable if you open another SSH session!

- With Docker Machine, you would do a command like:
  ```
  eval $(docker-machine env my-swarm-master)
  ```
Running on Swarm

- Swarm doesn't support builds (yet)
- ⚠️ Our version of Compose will crash on builds
- Try it!

💻 Exercise:
- Run `docker-compose build` and see the traceback!
Building with Swarm

- 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
- What do, what do?
The plan

- Build locally
- Tag images
- Upload them to the hub
- Update the Compose file to use those images

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

- Look at build-tag-push.py
- Run it!

- Run it in the directory containing docker-compose.yml (suggestion: ../build-tag-push.py)
- It will create a new docker-compose.yml file (named docker-compose.yml-XXX where XXX is a timestamp)
Can we run this now?

Let's try!

Exercise:

- Run docker-compose -f docker-compose.yml-XXX up
Can we run this now?

Let's try!

Exercise:

- Run `docker-compose -f docker-compose.yml XXX up`

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

So, what do?
Network plumbing on Swarm

- We will share *network namespaces* (as seen before)
- Other available options:
  - injecting service addresses in environment variables
  - implementing service discovery in the application
  - using an overlay network like Weave or Pipework
Another use of network namespaces

- Two (or more) containers can share a network stack
- They will have the same IP address
- They will be able to connect over localhost
- Other containers can be added later
Connecting over localhost

Exercise:

- Start a container running redis:
  `docker run -d --name myredis redis`

- Start another container in the same network namespace:
  `docker run -ti --net container:myredis ubuntu`

- In the 2nd container, install telnet:
  `apt-get update && apt-get install telnet`

- In the 2nd container, connect to redis on localhost:
  `telnet localhost 6379`

Some Redis commands: "SET key value" "GET key"
Same IP address

- Let's confirm that our containers share the same IP address

Exercise:

- Run a couple of times:
  
  ```bash
docker run ubuntu ip addr ls
  ```

- Now run a couple of times:
  
  ```bash
docker run --net container:myredis ubuntu ip addr ls
  ```
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

⚠️ Services should try to reconnect!
• Ideally, we would use \texttt{--add-host} (and Docker Compose counterpart, \texttt{extra_hosts}) to populate /etc/hosts

• Unfortunately, this does not work yet (See \texttt{Swarm issue #908} for details)

• We'll populate /etc/hosts manually instead (with \texttt{docker exec})
Our tools

- *unlink-services.py*
  - replaces all links with extra_hosts entries

- *connect-services.py*
  - scans running containers
  - generate commands to patch /etc/hosts
  - generate commands to start ambassadors
Putting it together

Exercise:

• Generate the new Compose YAML file:
  ```
  ../unlink-services.py docker-compose.yml-XXX deploy.yml
  ```

• Start our services:
  ```
  docker-compose -f deploy.yml up -d
  docker-compose -f deploy.yml scale worker=8
  docker-compose -f deploy.yml scale rng=4
  ```

• Generate plumbing commands:
  ```
  ../connect-services.py deploy.yml
  ```

• Execute plumbing commands:
  ```
  ../connect-services.py deploy.yml | sh
  ```
Notes

- If you want to scale up or down, you have to re-do the whole plumbing
- This is not a design issue; just an implementation detail of the connect-services.py script
- Possible improvements:
  - get list of containers using a single API call
  - use labels to tag ambassador containers
  - update script to do fine-grained updates of /etc/hosts
  - update script to add/remove ambassadors carefully (instead of destroying+recreating them all)
Thanks!

Questions?

@jpetazzo
@docker