AWS re:Invent

From Local Docker Development to Production Deployments

Jérôme Petazzoni

Docker Inc.

amazon web services

What to Expect from the Session

We will talk about ...

- Docker Compose for development environments
- taking those environments to production
  - Docker cluster provisioning
  - container image building and deployment
  - service discovery
- Compose, Machine, Swarm, ECS

We expect that you are familiar with Docker fundamentals!

Introductions

- Jérôme Petazzoni (@jpetazzo)
- Since 2010: putting things in containers at dotCloud
  - polyglot PAAS
  - microservices
  - provisioning, metrics, scaling ...
  - massive deployment of LXC
- Since 2013: putting things in containers at Docker (reminder: dotCloud became Docker in 2013...)
- 5 years of experience on a 2 years old technology!

Introductions, take 2

- Hi, I'm Jérôme
- I'm a Software Engineer about to start a new gig!
- Tomorrow for my first day I will work on DockerCoins*
- (It's a cryptocurrency-blockchain-something system)
- My coworkers are using Docker all over the place
- My task will be to deploy their stack at scale

*Fictious project name; you can't buy pizzas or coffee with DockerCoins (yet).

Getting ready

Preparing for my first day

- I just received my new laptop!
- The only instructions where:

“Install the Docker Toolbox.”

- ~180 MB download for Windows and OS X

docker

Docs Support Training Tech Blog Blog Docker Hub Get Started

Products Customers Community Partners Company Careers Open Source

Products | What is Docker? | Pricing | Resources | Use Cases

- Docker Subscription
- Docker Hub
- Docker Trusted Registry
- Docker Engine
- Docker Kitematic
- Docker Toolbox
- Docker Registry
- Docker Machine
- Docker Swarm
- Docker Compose

Docker Toolbox

Getting Started Guide (Mac) | Getting Started Guide (Windows) | Contribute to Toolbox

Download (Mac) Download (Windows)

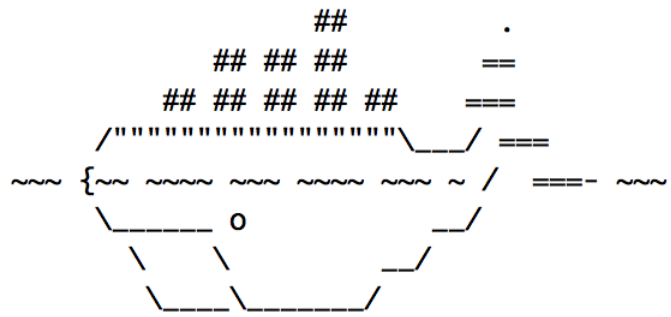
Video: https://www.youtube.com/watch?v=g-g94H_AiOE

Developing with Compose

The Compose on-boarding workflow

- Three simple steps:

1) git clone
2) docker-compose up
3) open app in browser



```
docker is configured to use the default machine with IP 192.168.99.100
For help getting started, check out the docs at https://docs.docker.com

moonshine:~ jp$
```

Video: https://www.youtube.com/watch?v=sk3yYh1MgE0

How does this work?

- “docker-compose up” tells Compose to start the app
- If needed, the app is built first
- How does Compose know what to do?
- It reads the “Compose file” (docker-compose.yml)

docker-compose.yml — simple application

```
web:
  build: .
  ports:
   - "80:5000"
  links:
   - redis

redis:
  image: redis
```

docker-compose.yml — complex application

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

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

redis:
    image: redis
```

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

webui:
    build: webui
    links:
      - redis
    ports:
      - "8000:80"
    volumes:
      - "./webui/files/:/files/"
```

How does this work?

- Application is broken down into services
- Each service is mapped to a container
- Each container can come from:
  - a pre-built image in a library called “registry”
  - a build recipe called “Dockerfile”
- The Compose file defines all those services
  (and their parameters: storage, network, env vars...)

This repository Search Pull requests Issues Gist

jpetazzo / dockercoins

Unwatch 1 Star 0 Fork 0

Description

Short description of this repository

Website

Website for this repository (optional)

Save or Cancel

1 commit 1 branch 0 releases 1 contributor

Branch: **master** dockercoins / +

jpetazzo authored an hour ago latest commit `fc792d5d4d`

hasher	an hour ago
rng	an hour ago
webui	an hour ago
worker	an hour ago
docker-compose.yml	an hour ago
ports.yml	an hour ago

Code
Issues 0
Pull requests 0
Wiki
Pulse
Graphs
Settings

SSH clone URL

git@github.com:jpetaz

You can clone with HTTPS, SSH, or Subversion.

Our sample application

- Microservices architecture
- Different languages and frameworks
  - Ruby + Sinatra
  - Python + Flask
  - Node.js + Express
- Different kinds of services
  - background workers
  - web services with REST API
  - stateful data stores
  - web front-ends

Mandatory plug on microservices

- Advantages of microservices:
  - enables small teams (Jeff Bezos two-pizza rule)
  - enables “right tool for the right job”
  - services can be deployed/scaled independently
  - look for e.g. “Adrian Cockroft Microservices” talks
- Drawbacks to microservices:
  - distributed systems are hard
    (cf. aphyr.com if you have doubts)
  - load balancing, service discovery become essential
  - look for e.g. “Microservices Not A Free Lunch” article

Deploying on a Cloud Instance

- Same workflow:

1) ssh into remote Docker Host
2) git clone
3) docker-compose up
4) open app in browser

- Let's see a real demo!

```
Last login: Sat Oct  3 17:35:36 on ttys000
moonshine:~ jp$ ssh reinvent
```

DEMO

- git clone git://github.com/jpetazzo/dockercoins
- cd dockercoins
- docker-compose up
- open app (instance address, port 8000)
- ^C

The Compose development workflow

- Four simple steps:

1) edit code
2) docker-compose build
3) docker-compose up
4) reload app in browser

DEMO

- edit webui/files/index.html
- change css
- docker-compose build
- docker-compose up
- reload app
- ^C

Video: https://www.youtube.com/watch?v=O3Bps01THBQ

Compose take-aways

- Docker abstracts the environment for us
- Any Docker host is a valid deployment target:
  - local environment
    (with the Docker Toolbox)
  - on-demand cloud instances
    (with Docker Machine)
  - bring-Your-Own-Server
    (for on-prem and hybrid strategies)
- Frictionless on-boarding (and context-switching)
- But how do we deploy to production, at scale?

What's missing

- Cluster provisioning
- Building and deploying code
- Service discovery

(Non-exhaustive list.)

Let's see how to address those points.
We will dive into details — and give more live demos!

Provisioning a cluster

Provisioning

- Manual instance creation (CLI or Console)
- AWS CLI scripting
- Auto Scaling Groups
- CloudFormation templates
- Docker Machine
- ECS CLI

Docker Machine

- Docker Machine comes with the Docker Toolbox
- Can create Docker hosts on:
  - EC2 and other clouds
  - local environments (VirtualBox, OpenStack…)
- Can create clusters using Docker Swarm
- Current limitations (but expect this to improve):
  - one machine at a time
  - centralized credentials

DEMO

```
export TOKEN=$(docker run swarm create)
echo $TOKEN

docker-machine create -d amazonec2 --swarm \
    --swarm-master --swarm-discovery token://$TOKEN node00 &

for N in $(seq 1 4); do
    sleep 3
    docker-machine create -d amazonec2 --swarm \
        --swarm-discovery token://$TOKEN node0$N &
done

wait
```

Video: https://www.youtube.com/watch?v=LFjwusorazs

ECS CLI

- Sneak peek!
- State-of-the-art cluster creation
- Following AWS best practices:
  - CloudFormation template
  - Auto Scaling Group
  - IAM integration

DEMO

- ecs-cli configure
- ecs-cli up --keypair jpetazzo --capability-iam --size 10
- (add elastic load balancer)
- (associate load balancer with auto scaling group)
- (add DNS entry)
- (configure security groups for ELB and ASG)

Video: https://www.youtube.com/watch?v=KqEpIDFxjNc

Building and deploying code

Building and deploying with Docker

- Let's continue to use Compose to build our app images
- And store those images in a Docker Registry
  - Docker Hub
    (SAAS à la GitHub, free for public images)
  - Docker Trusted Registry
    (commercial offering; available e.g. through AWS marketplace)
  - self-hosted, community version

The plan

- Each time we need to deploy:

1) build all containers with Compose
2) tag all images with a unique version number
3) push all images to our Registry
4) generate a new docker-compose.yml file,
   referencing the images that we just built and pushed

- This will be done by a script

You get a script!
And you get a script!
Everybody gets a script!

- All the scripts that we will use here are on GitHub
- Feel free to use them, copy them, adapt them, etc.

URL: https://github.com/jpetazzo/orchestration-workshop

(Don't panic, URL will be shown again at the end of the presentation)

DEMO

- build-tag-push.py
- inspect the resulting YAML file

Those images are now frozen.

They'll stay around “forever” if we need them again.
(e.g. to do a version rollback)

See: https://hub.docker.com/r/jpetazzo/dockercoins_webui/tags/

Service discovery

Why do we need service discovery?

- Service A needs to talk to service B
- How does A know how to talk to B?
  - service A needs: address, port, credentials
- What if there are multiple instances of B?
  - examples: load balancing, replication
- What if B location changes over time?
  - examples: scaling, fail-over
- Service discovery addresses those concerns

Service discovery, seen by devs

Hard-coded service discovery

- Development setup:

```
$db = mysql_connect(“localhost”);

cache = Redis.new(:host => "localhost", :port => 16379)

conn, err := net.Dial("tcp", "localhost:8000”)
```

Hard-coded service discovery

- Other development setup:

```
$db = mysql_connect(“192.168.1.2”);

cache = Redis.new(:host => "192.168.1.3", :port => 6380)

conn, err := net.Dial("tcp", "192.168.1.4:8080”)
```

Hard-coded service discovery

- Production setup:

```
$db = mysql_connect(
        “foo.rds.amazonaws.com”, “produser”, “sesame”);

cache = Redis.new(
           :url => "redis://:p4ssw0rd@redis-as-a-service.io/15”)

conn, err := net.Dial(
                  "tcp", "api-42.elb.amazonaws.com:80”)
```

Hard-coded service discovery

- Requires many code edits to change environment
- Error-prone
- Big, repetitive configuration files often land in the repo
- Adding a new service requires editing all those configs
- Maintenance is expensive
  (S services × E environments)
- 😱

Twelve-factor App

- Environment variables

```
$db = mysql_connect(
        $_ENV[“DB_HOST”],
        $_ENV[“DB_USER”], $_ENV[“DB_PASS”])

cache = Redis.new(
          :url => "redis://:#{ENV[“REDIS_PASS”]}@” +
                  “#{ENV[“REDIS_HOST]}:#{ENV[“REDIS_PORT]}/” +
                  “#{ENV[“REDIS_DB”]}”)

conn, err := net.Dial(
                "tcp", os.ExpandEnv("${API_HOST}:${API_PORT}”))
```

Twelve-factor App

- Separates cleanly code and environment variables (environment is *literally* defined by environment variables)
- Still requires to maintain configuration files (containing lists of environment variables)
- Production parameters are easier to keep out of the repo
- Dramatic errors are less likely to happen
- 😐

Configuration database

- Dynamic lookups (here with Zookeeper)

```
$zk = new Zookeeper('127.0.0.1:2181');
mysql_connect(
    $zk→get('/apps/foo/prod/db/host')
    $zk→get('/apps/foo/prod/db/user')
    $zk→get('/apps/foo/prod/db/pass'))

zk = Zookeeper.new('127.0.0.1:2181')
redis_pass = zk.get(:path => '/apps/foo/prod/redis/pass')
redis_host = zk.get(:path => '/apps/foo/prod/redis/host')
redis_port = zk.get(:path => '/apps/foo/prod/redis/port')
redis_db = zk.get(:path => '/apps/foo/prod/redis/db')
cache = Redis.new(
            :url => "redis://:#{redis_pass}@#{redis_host}:#{redis_port}/#{redis_db}")

c, _, err := zk.Connect([]string{"127.0.0.1"}, time.Second)
api_host, _, err := c.get("/apps/foo/prod/api/host")
api_port, _, err := c.get("/apps/foo/prod/api/port")
conn, err := net.Dial("tcp"", fmt.Sprintf("%s:%s", api_host, api_port))
```

Configuration database

- If you want the same code in dev and prod,
  you need to deploy your config DB in dev too
- Instead of maintaining config files,
  you maintain Zookeeper* clusters and fixtures
- … or have different lookup logic for dev and prod
- 😟

*Or your other favorite config DB, e.g. etcd, Consul...

Local load balancing / routing

- Connect to well-known location

```
$db = mysql_connect(“localhost”);

cache = Redis.new(:host => "localhost")

conn, err := net.Dial("tcp", "localhost:8001”)
```

- In dev: all components run locally
- In prod: local load balancer routes the traffic (example: AirBNB's SmartStack)

Local load balancing / routing

- Code can be identical in dev and prod
- Deployment will differ:
  - direct connection in dev
  - proxies, routers, load balancers in prod
- “Configuration” is merely a static port allocation map (indicating which service listens on which port)
- Way easier for devs; however ops still have work to do
- 😒

The ambassador pattern

Our code base with ambassadors

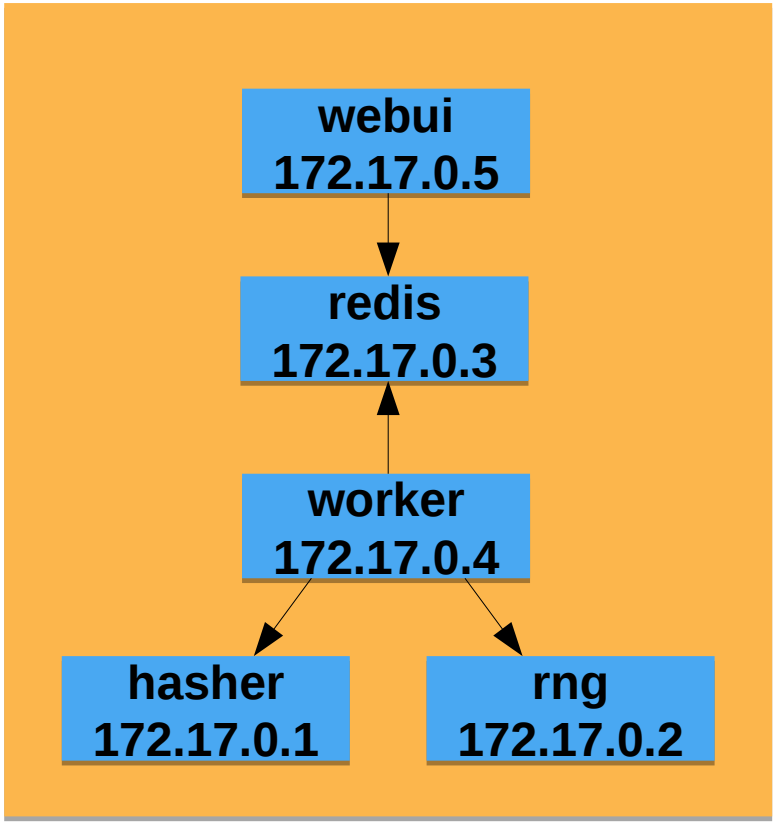
- Use well-known DNS names

```
$db = mysql_connect(“db”);

cache = Redis.new(:host => "redis")

conn, err := net.Dial("tcp", "api:80”)
```

Running in dev



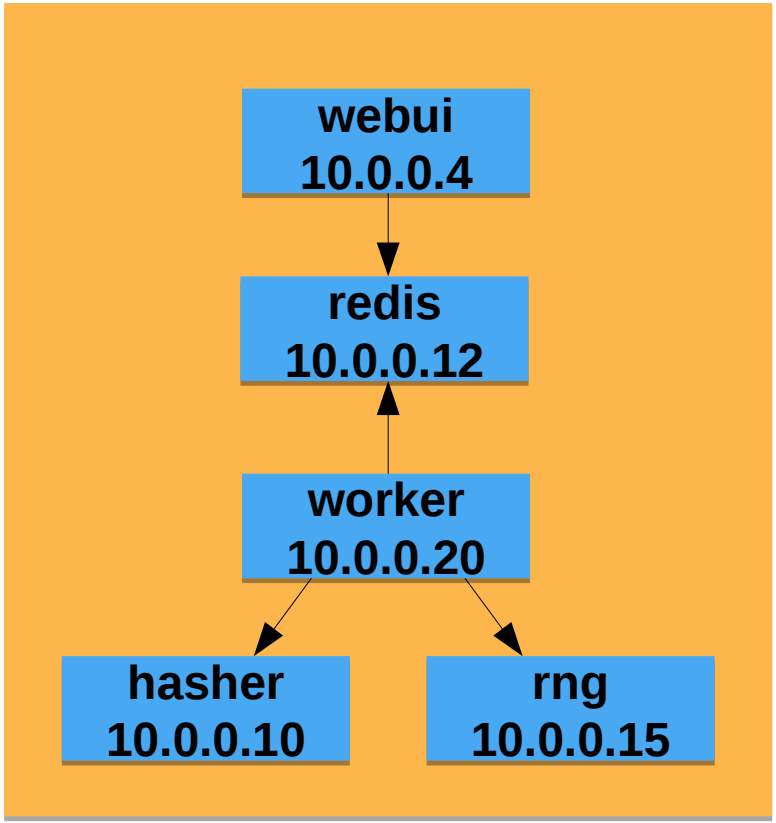
Let's populate a custom /etc/hosts file in each container, referencing the services that it needs to connect to.

e.g. on “worker”:

172.17.0.1 hasher
172.17.0.2 rng
172.17.0.3 redis

Host | Container

Another dev environment



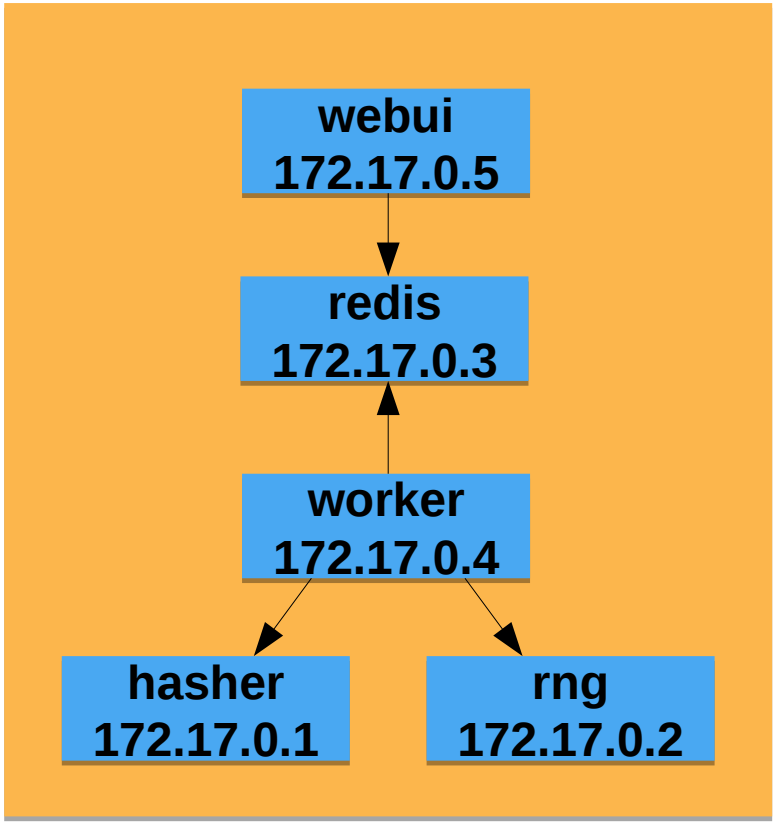
The addressing is different, but the code remains the same.

/etc/hosts on “worker”:

10.0.0.10 hasher
10.0.0.15 rng
10.0.0.12 redis

Host Container

Some good news



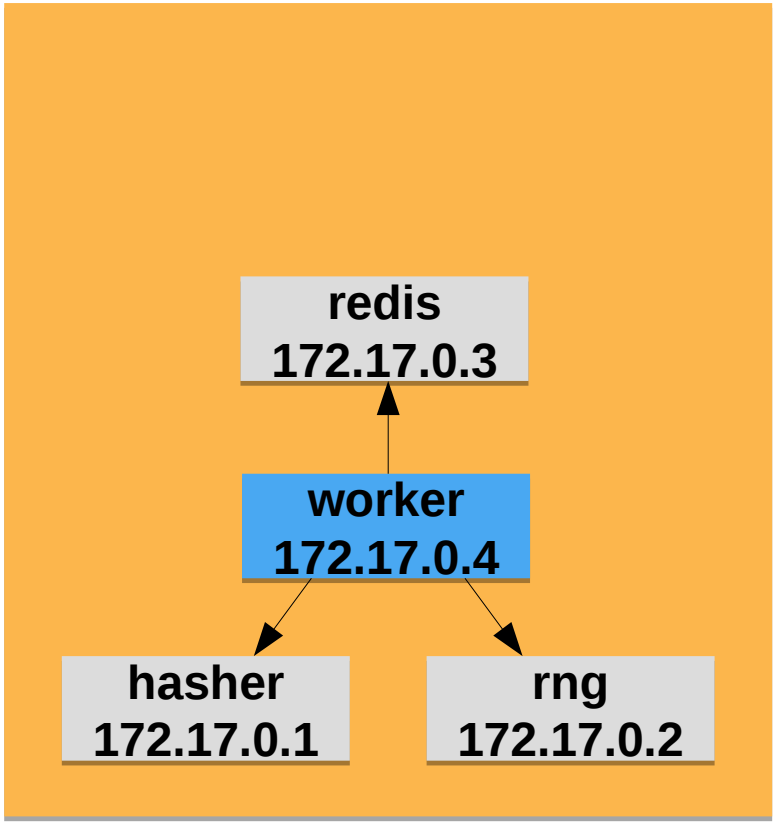
Compose automatically does this for us, using Docker “links.” Links populate /etc/hosts.

Our dev environment is already taken care of!

But what about our production setup on multiple hosts?

Host Container

Running in prod



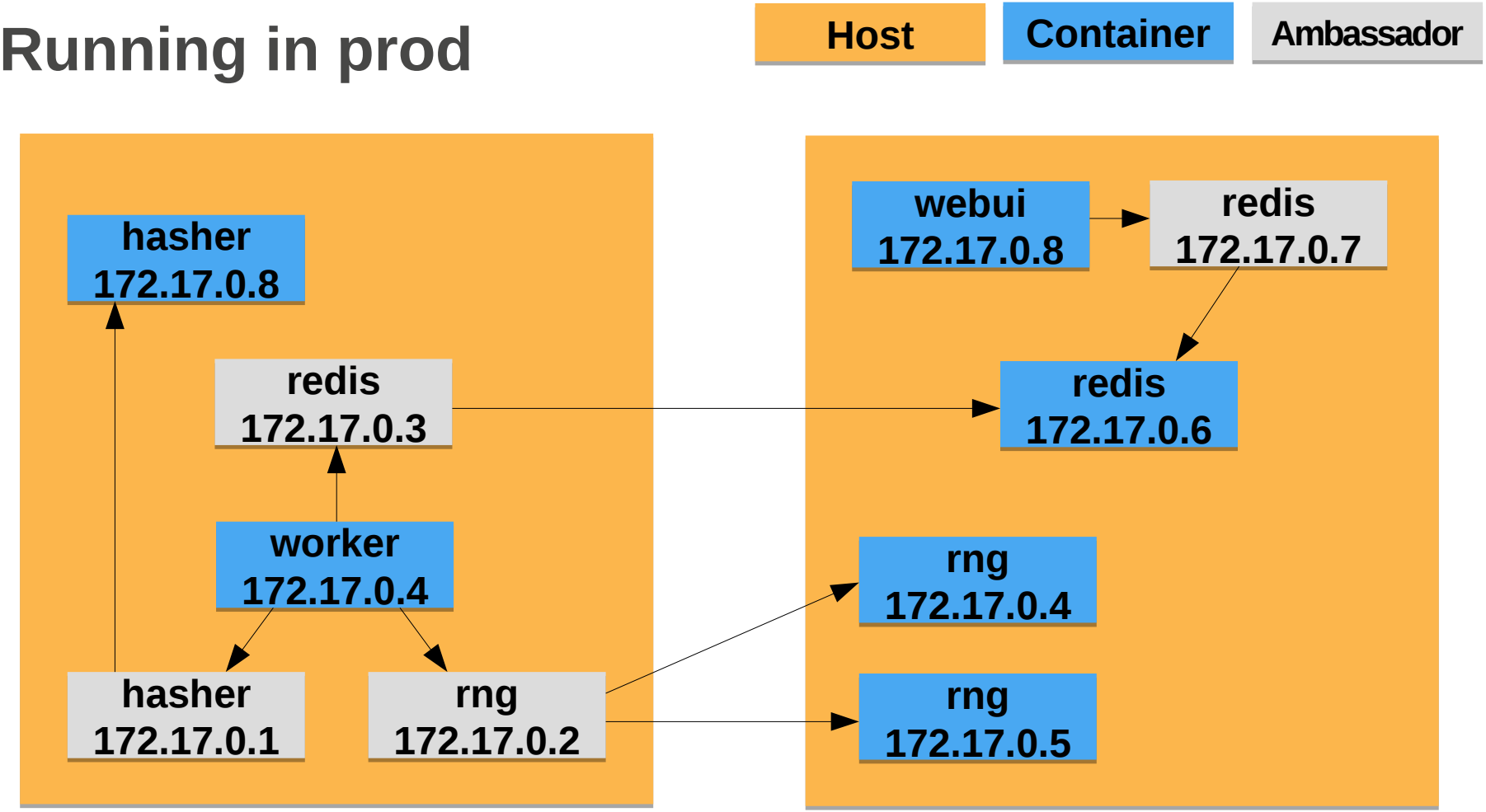
Worker doesn't talk to actual instances of redis, hasher, and rng, but to *ambassadors.*

Ambassadors will route* the traffic to the destination.

*Or forward, load-balance, proxy...

Host | Container | Ambassador

Running in prod



Using ambassadors

- Code remains readable and clean
- Plumbing (service discovery, routing, load balancing, etc.) is abstracted away (somebody still has to do it, though!)
- Plumbing doesn't encumber our dev environment
- Changes in plumbing won't impact the code base
- 😊

Service discovery, seen by ops

How fast are we moving?

Moving slowly

- Code deployment is infrequent:
  - every week, on a regular schedule
  - a bit of downtime is OK (a few minutes, maybe one hour)
- Failures are rare (less than 1/year)
  and/or don't have critical impact
- Reconfigurations are not urgent:
  - we bake them in the deployment process
  - it's OK if they disrupt service or cause downtime

Strategy for apps moving slowly

- Bake configuration and parameters with the deployment (reconfiguration = rebuild, repush, redeploy)
- Or configure manually after deployment (!)
- In case of emergency: SSH+vi (!)

Results

- Advantages
  - zero cost upfront
  - easy to understand*
- Drawbacks
  - each deployment, each change = risk
  - expensive in the long run

*Except for your boss when your app is down and it takes a while to bring it back up

Moving mildly

- Code deployment:
  - happens every day
  - downtime is not OK (except maybe very short glitches)
- Failures happen regularly;
  they must be resolved quickly
- Reconfigurations are frequent:
  - scaling up/down; moving workloads; changing databases
  - altering application parameters for A/B testing

Strategy for apps moving mildly

- Inject configuration after the deployment
- When you just want to change a parameter: reconfigure (without redeploying everything)
- Automate the process with a “push button” script

Results

- Advantages
  - easy to understand and to implement
  - no extra moving part
    (just this extra “push button” script/process)
- Drawbacks
  - services must allow reconfiguration
  - reconfiguration has to be triggered after each change
  - risk of meta-failure (bug in the deployment system)

Moving wildly*

- Code deployment:
  - happens continuously (10, 100, 1000+ times a day)
  - downtime is not OK, even if it's just a few sporadic failed requests
- Failures happen all the time;
  repair actions must be fully automated
- Reconfigurations are part of the app lifecycle:
  - automatic scaling, following planned and unplanned patterns
  - generalized blue/green deployment, canary testing, etc.

*a.k.a “move fast and break things”

Strategy for apps moving wildly

- Requirement: detect changes as they happen
- Use a combination of:
  - monitoring
  - live stream of events that we can subscribe to
  - services that register themselves
  - fast polling
- After deployment, scaling, outage, metric threshold…: automatic reconfiguration

Results

- Advantages
  - everything happens automatically
  - no extra step to run when you deploy
  - more modular
    (different processes can take care of different service types)
- Drawbacks
  - extra moving parts and services to maintain
  - meta-failures are even more dangerous

Recap table

	How fast should we move?			How much work is it for ...		How do we handle ...	
	Slowly	Mildly	Wildly	Devs	Ops	Scaling	Failures
Hard-coded	✅	🚫	🚫	😀	😀	😠	😰
12-Factor	✅	🔃	🚫	😀	😀	😕	😕
Config Database	✅	✅	✅	😩	😠	😎	😎
Local LB/routers	✅	✅	✅	😐	😐	😎	😎
Ambassadors	✅	✅	✅	😀	😐	😎	😎

Recap table (subtitles)

	How fast should we move?			How much work is it for ...		How do we handle ...	
	Slowly	Mildly	Wildly	Devs	Ops	Scaling	Failures
Hard-coded	OK	NO	NO	easy	easy	painfully	horribly
12-Factor	OK	OK WITH RESTARTS	NO	easy	easy	meh	meh
Config Database	OK	OK	OK	hard	hard	cool	cool
Local LB/routers	OK	OK	OK	medium	medium /hard	cool	cool
Ambassadors	OK	OK	OK	easy	medium /hard	cool	cool

Ambassadors in action

The plan

- Deploy a simple application (trainingwheels)
  - on ECS
  - on Swarm
- Deploy a complex application (dockercoins)
  - on ECS
  - on Swarm

Our simple application, “trainingwheels”

- Two service:
  - web server
  - redis data store
- Tells you which web server served your request
- Counts how many requests were served
- Keeps separate counters for each server

DEMO

- cd ~
- git clone git://github.com/jpetazzo/trainingwheels
- cd trainingwheels
- docker-compose up
- open app
- ^C

Deploying on ECS

- On ECS, a container is created as a member of a *task*
- Tasks are created from *task definitions*
- Task definitions are conceptually similar to Compose files (but in a different format)
- ECS CLI to the rescue!

Deploying on ECS

- ECS CLI will:
  - create a *task definition* from our Compose file
  - register that task definition with ECS
  - run a *task instance* from that task definition
- ECS CLI will not:
  - work if your Compose file has a “build” section
    (it only accepts “image” sections)
- Let's use the “build-tag-push” script shown earlier!

DEMO

- build-tag-push.py
- set COMPOSE_FILE
- fixup-yaml.sh

Scaling “trainingwheels” on ECS

At this point, if we deploy and scale, we will end up with multiple copies of the app, each with its own Redis.

To avoid this, we need to deploy our first ambassador!

Here is the plan:

- Create a new Compose file for our Redis service
- Use ECS CLI to run redis, and note its location
- Update the main Compose file so that the “redis” service is now an ambassador pointing to the actual Redis

Introducting jpetazzo/hamba

- Easy ambassadoring for the masses!
- In a shell:

  ```
  docker run jpetazzo/hamba <frontend-port> \
      [backend1-addr] [backend1-port] \
      [backend2-addr] [backend2-port] …
  ```

- In a Compose file:

  ```
  redis:
      image: jpetazzo/hamba
      command: <front-port> [backend-addr] [backend-port] ...
  ```

DEMO (1/2)

- mkdir ~/myredis
- cp $COMPOSE_FILE ~/myredis
- cd ~/myredis
- edit $COMPOSE_FILE
  - expose port 6379
  - remove www service
- ecs-cli compose up
- ecs-cli compose ps
- note host+port

DEMO (2/2)

- cd ~/trainingwheels
- edit $COMPOSE_FILE
  - replace redis image with jpetazzo/hamba
  - add “command: 6379 <redishost> <redisport>”
- ecs-cli compose up
- ecs-cli compose scale 4
- watch ecs-cli compose ps
- open a couple of apps
- open the load balancer

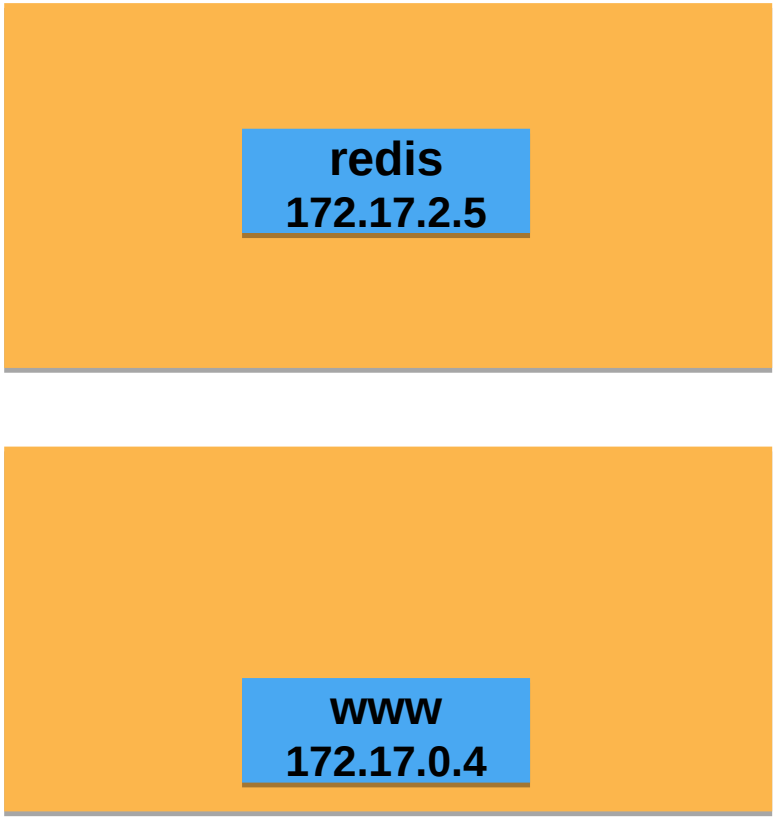
CLEANUP

- ecs-cli compose down
- Let redis running (we'll re-use it later)

Scaling “trainingwheels” on Swarm

- Slightly different idea!
- We keep a single Compose file for our app
- We replace links with ambassadors:
  - using a local address (127.X.Y.Z)
  - sharing the client container's namespace
- Each container that needs to connect to another service, gets its own private load balancer for this exact service
- That's a lot of load balancers, but don't worry, they're cheap

Network namespace ambassadors



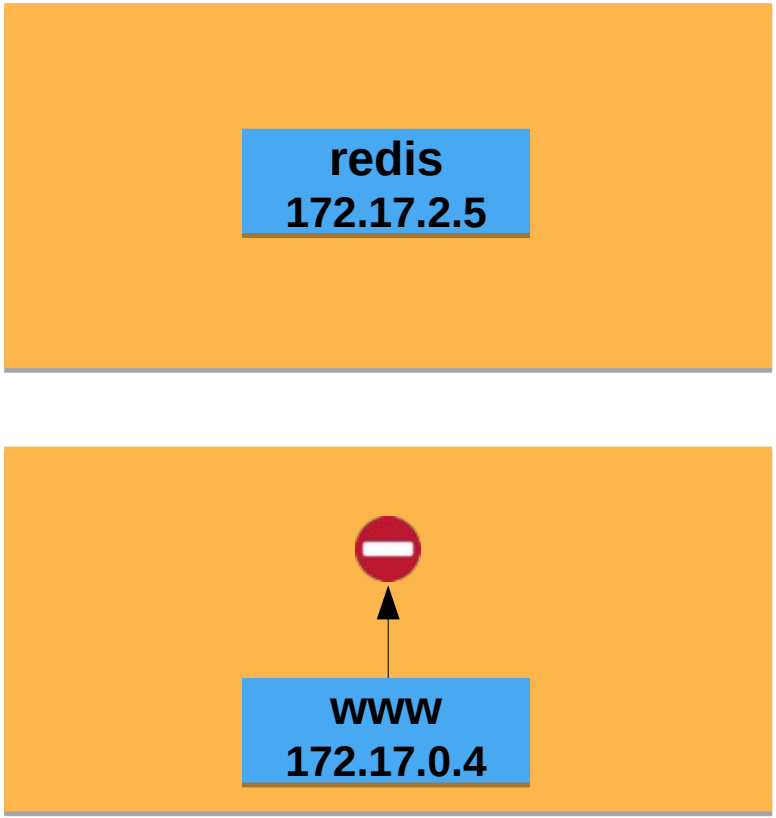
"redis" and "www" containers are created by Compose, and placed by Swarm, potentially on different hosts.

In "www", /etc/hosts has the following entry:

127.127.0.2 redis

Host | Container | Ambassador

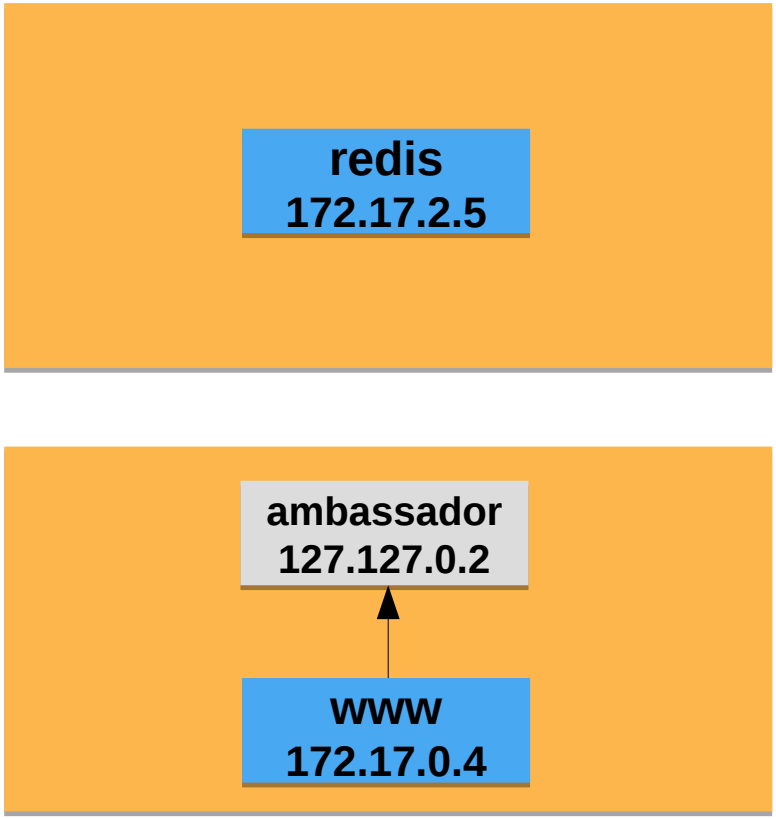
Network namespace ambassadors



At this stage, connection attempts from “www” to “redis” fail with “connection refused.”

Host Container Ambassador

Network namespace ambassadors

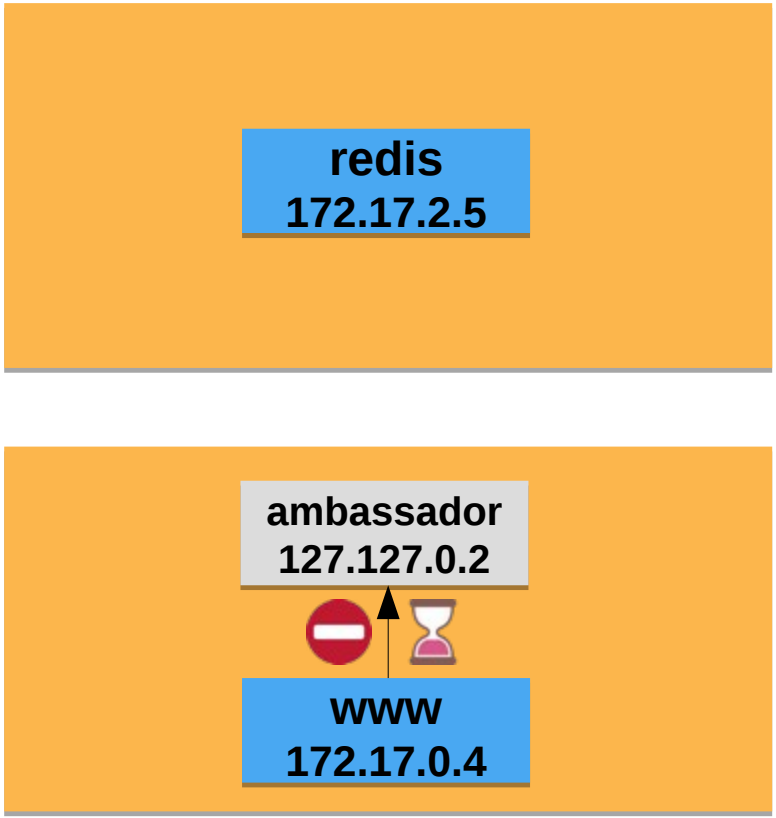


The ambassador is created.

It's sharing the network namespace of the “www” container, meaning that they have the same loopback interface (they can talk over localhost).

Host | Container | Ambassador

Network namespace ambassadors



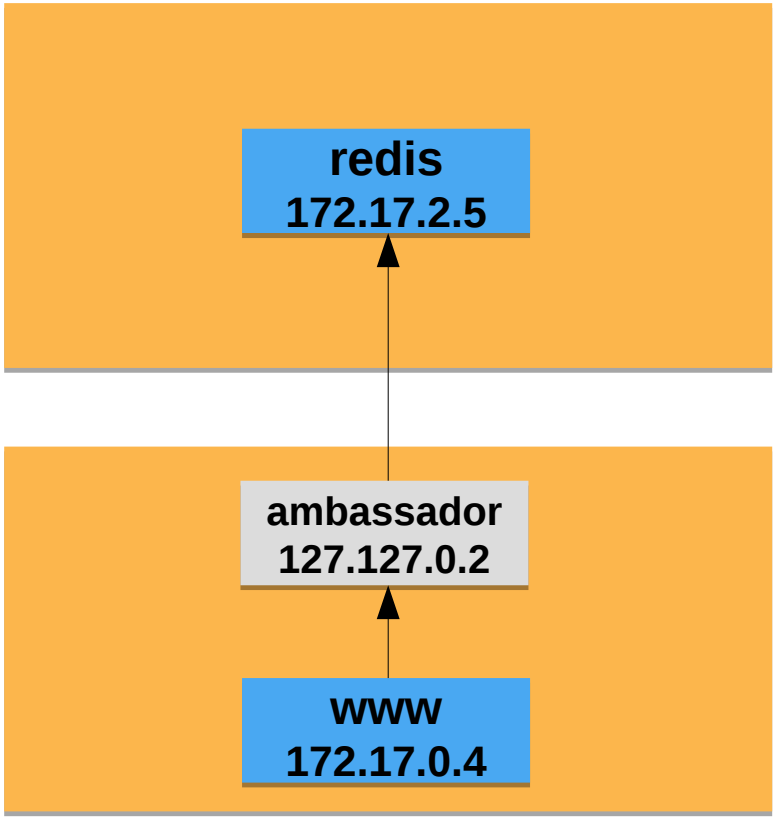
At this stage, connections are still failing (with either “connection refused” or a timeout, depending on the load balancer settings.)

The application has to handle this gracefully.

(Crashing and being restarted is graceful enough.)

Host | Container | Ambassador

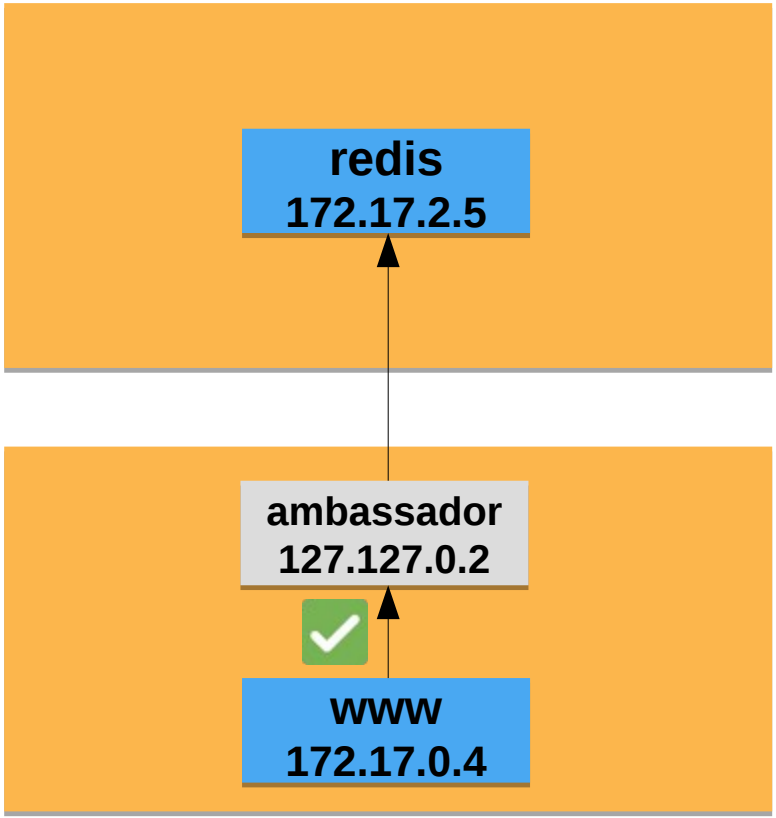
Network namespace ambassadors



The ambassador receives its configuration, containing the public address of the “redis” container.

Host | Container | Ambassador

Network namespace ambassadors



Traffic can now flow normally from “www” to “redis”.

Host | Container | Ambassador

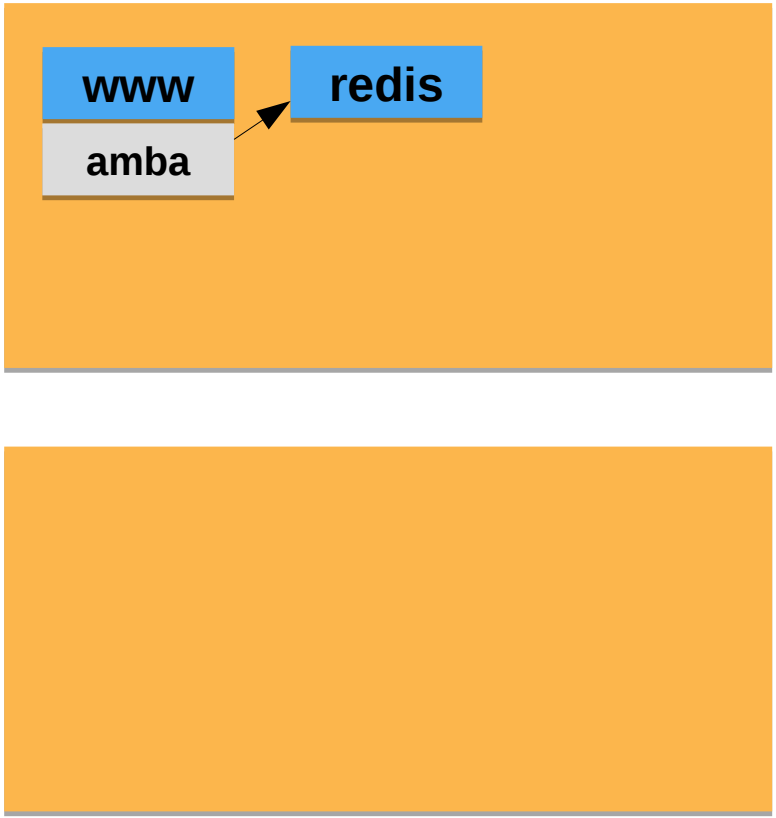
DEMO (1/2)

- eval $(docker-machine env node00 --swarm)
- edit $COMPOSE_FILE
  - revert “redis” to use “image: redis”
  - remove “command:”
- link-to-ambassadors.py
- docker-compose up -d
- docker-compose ps
- open app
- (It doesn't work — yet)

DEMO (2/2)

- create-ambassadors.py
- configure-ambassadors.py
- open app
- docker-compose scale www=4
- create-ambassadors.py
- configure-ambassadors.py
- open app

Scaling with ambassadors

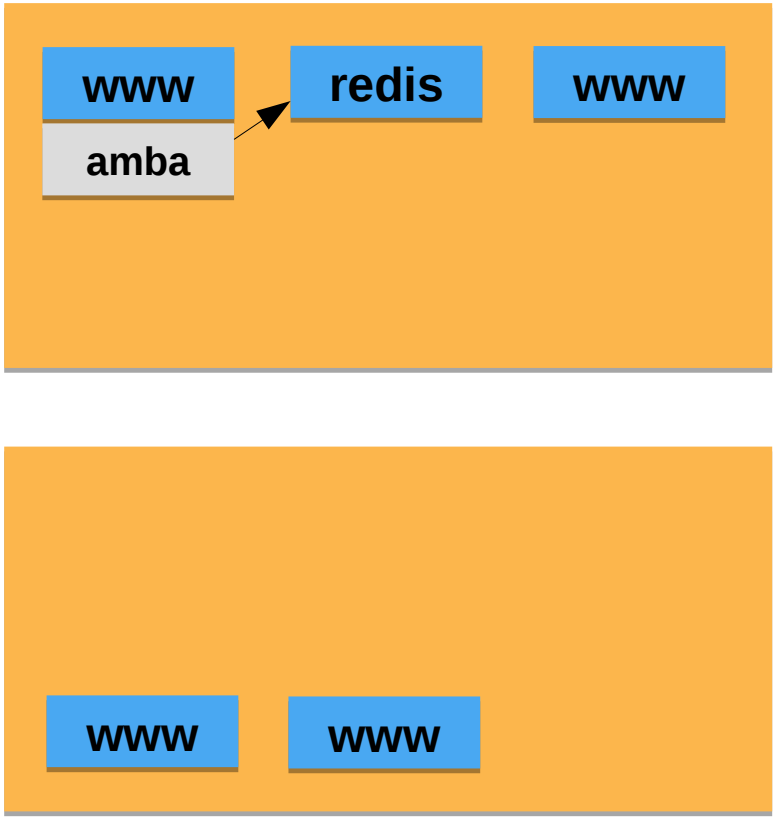


Before scaling our app, we have one single “www” instance, coupled with its ambassador.

(In this example, we have placed the first “www” and “redis” together for clarity.)

Host | Container | Ambassador

Scaling with ambassadors

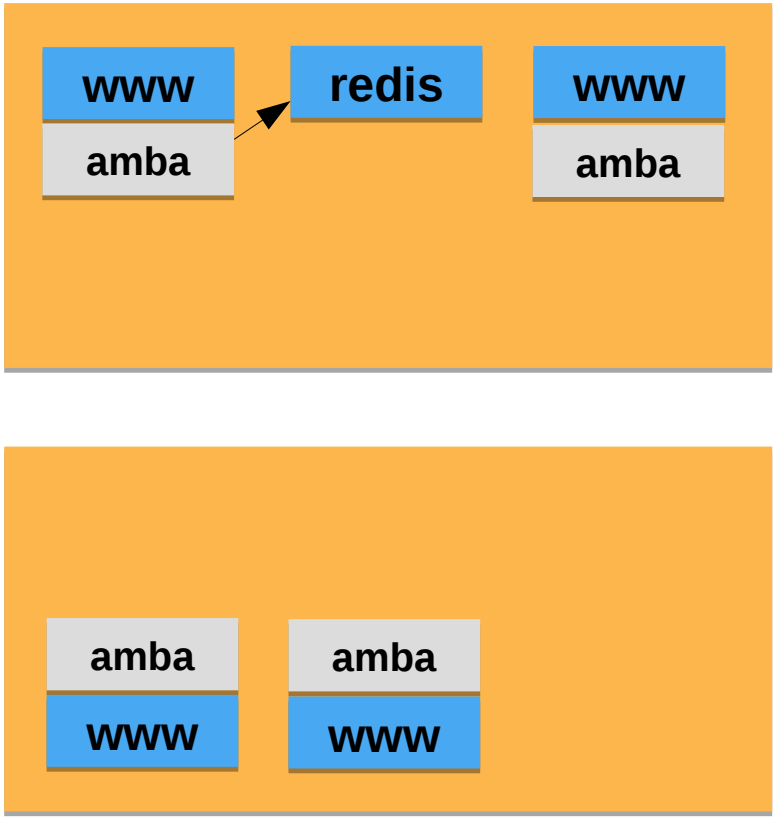


“docker-compose scale www=4”

We now have 4 instances of “www” but 3 of them can't communicate with “redis” yet.

Host | Container | Ambassador

Scaling with ambassadors

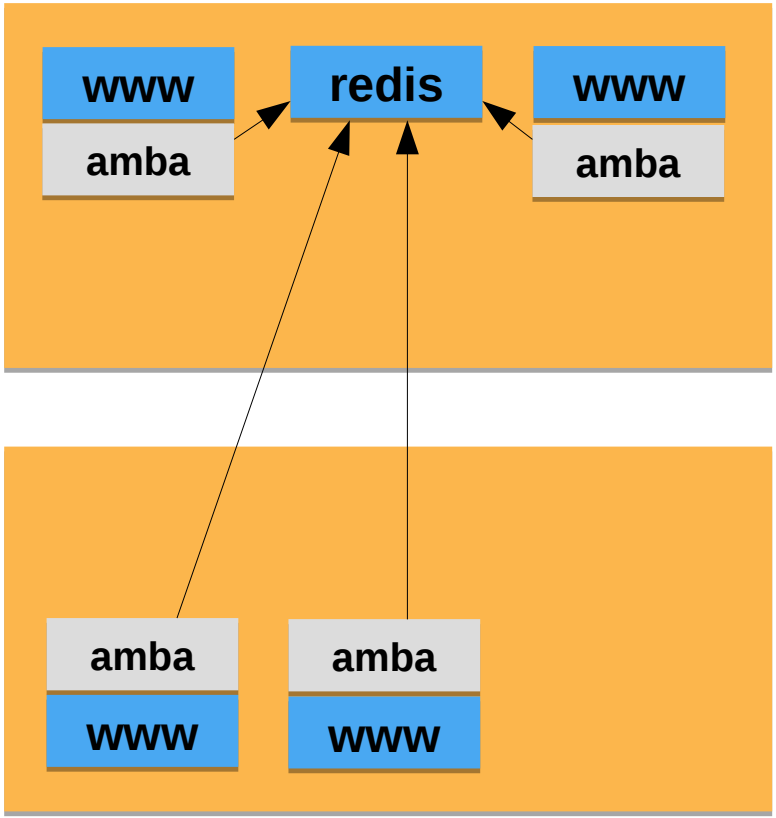


"create-ambassadors.py"

Each "www" instance now has its own ambassador, but 3 of them are still unconfigured.

Host Container Ambassador

Scaling with ambassadors



“configure-ambassadors.py”

The 3 new ambassadors receive their configuration and can now route traffic to the “redis” service.

Host | Container | Ambassador

CLEANUP

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

Scaling “dockercoins” on ECS

- Let's apply the same technique as before
- Separate the Redis service
- Replace “redis” with an ambassador in the Compose file
- Let ECS do the rest!

DEMO (1/2)

- Get our Redis host+port again:
  - cd ~/myredis
  - ecs-cli compose ps
- cd ~/dockercoins
- set COMPOSE_FILE
- edit $COMPOSE_FILE
  - change “image: redis” to “image: jpetazzo/hamba”
  - add “command: 6379 <redishost> <redisport>
  - add “mem_limit: 100000000” everywhere
  - remove volumes
- fixup-yaml.sh

DEMO (2/2)

- ecs-cli compose up
- watch ecs-cli compose ps
- open webui
- ecs-cli compose scale 4
- watch ecs-cli compose ps
- open webui
- repeat!

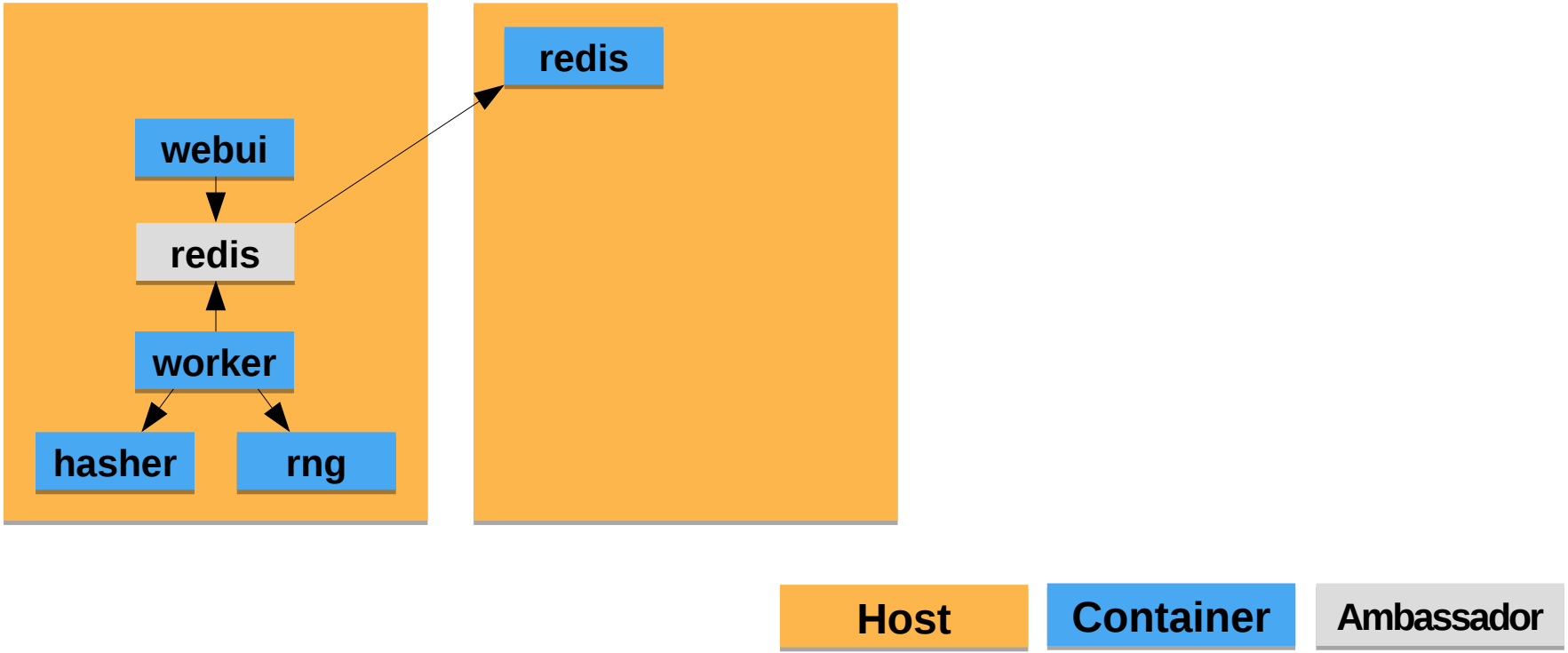
Scaling “dockercoins” on ECS

- We started with our “redis” service...

Host Container Ambassador

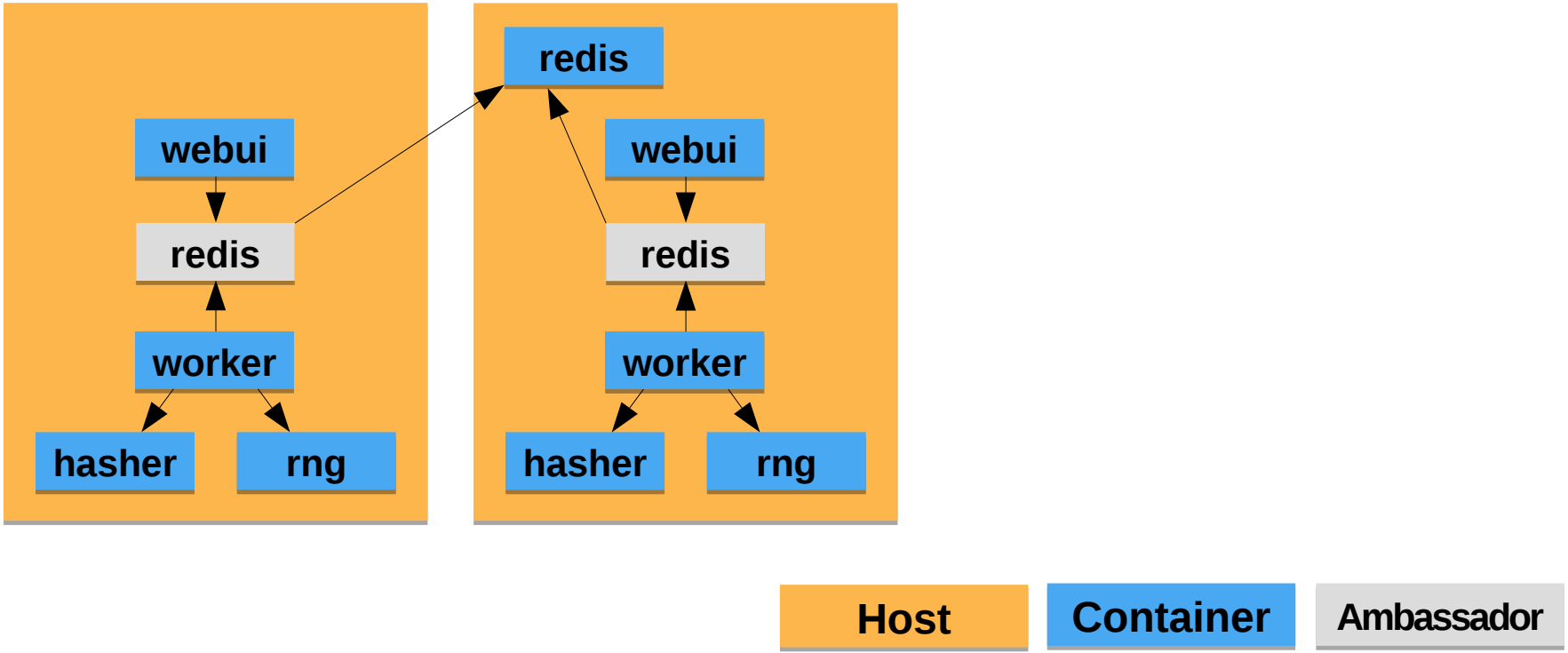
Scaling “dockercoins” on ECS

- Created one instance of the stack with an ambassador...



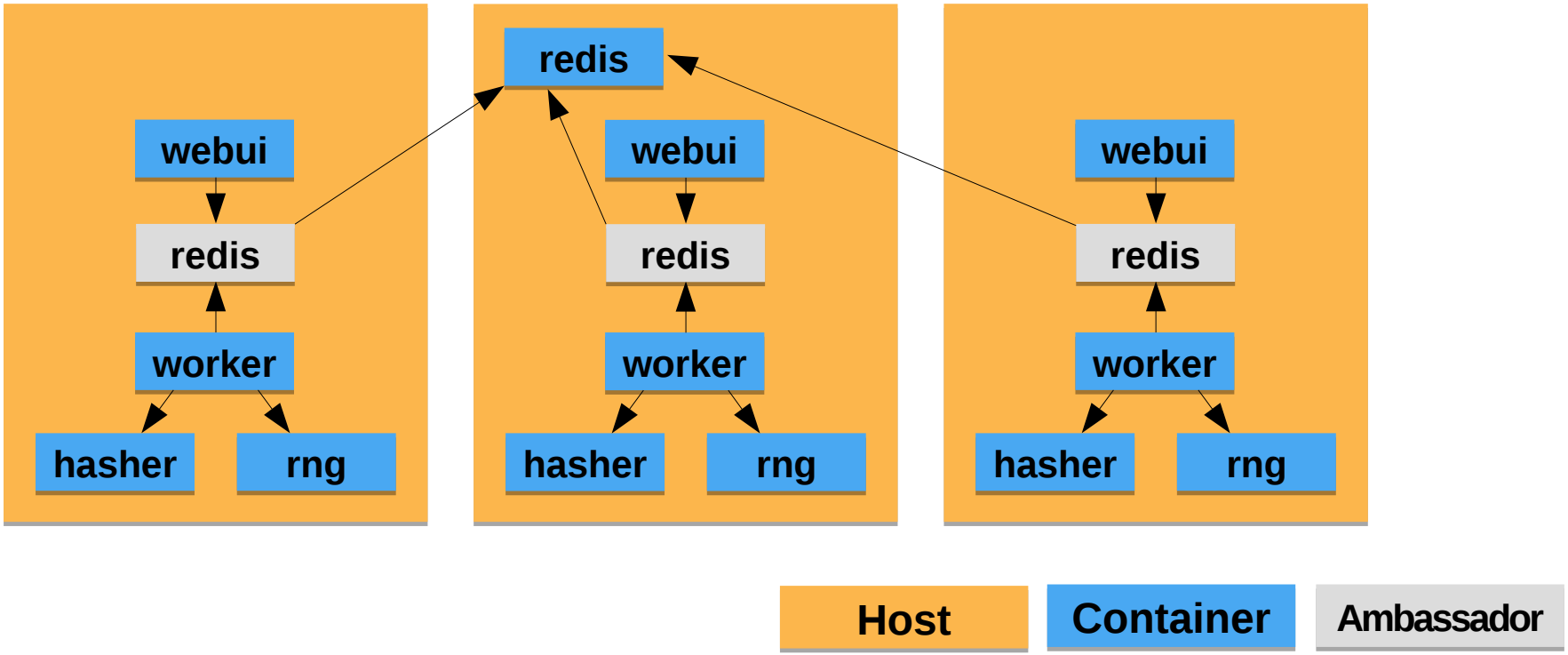
Scaling “dockercoins” on ECS

- Added a second instance of the full stack...



Scaling “dockercoins” on ECS

- And another one… etc.



Scaling “dockercoins” on Swarm

- Let's apply the same technique as before
- Replace links with ambassadors
- Start containers
- Add ambassadors
- Inject ambassador configuration

DEMO (1/2)

- edit COMPOSE_FILE
  - restore “image: redis”
  - remove “command:” from the redis section
- link-to-ambassadors.py
- docker-compose up -d
- create-ambassadors.py
- configure-ambassadors.py
- docker-compose ps webui
- open webui

DEMO (2/2)

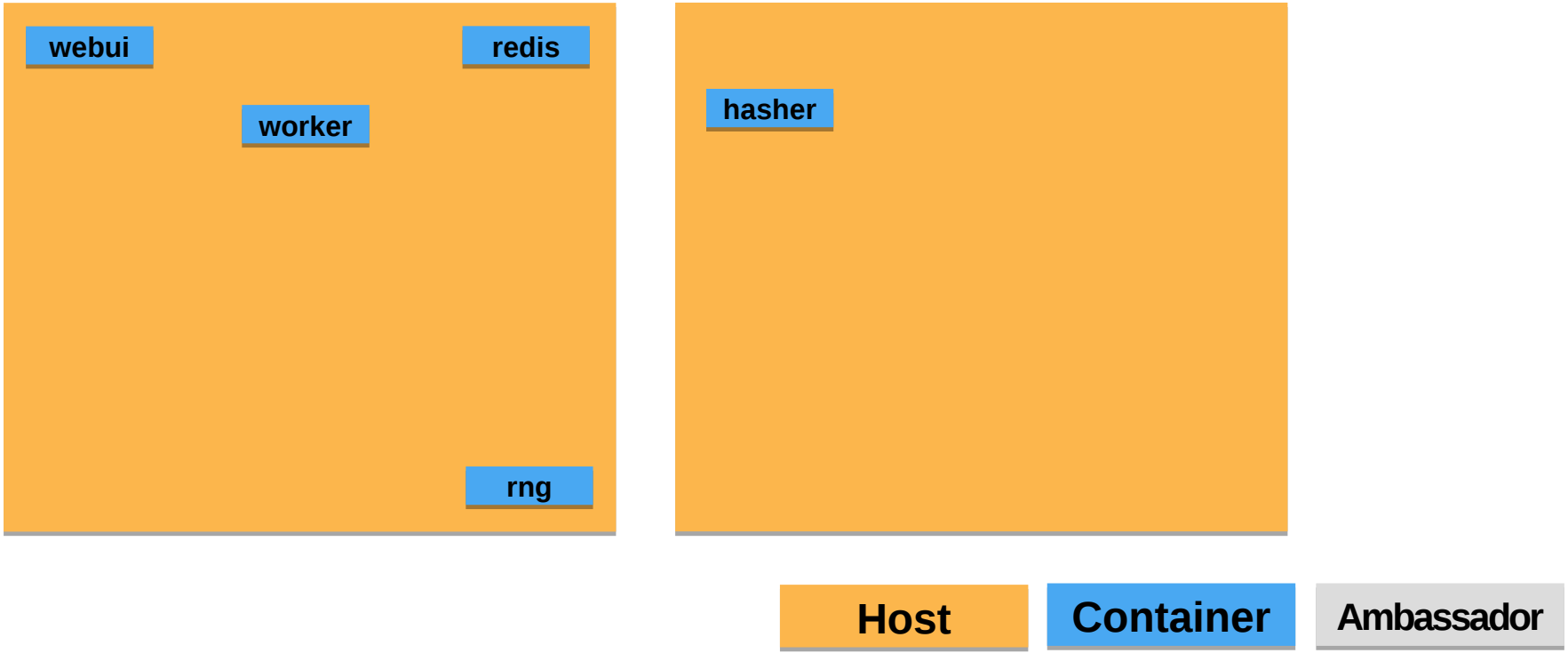
- docker-compose scale \
  webui=2 worker=10 rng=20 hasher=5
- create-ambassadors.py
- configure-ambassadors.py

Scaling “dockercoins” on Swarm

- Two (for simplicity) empty Docker hosts

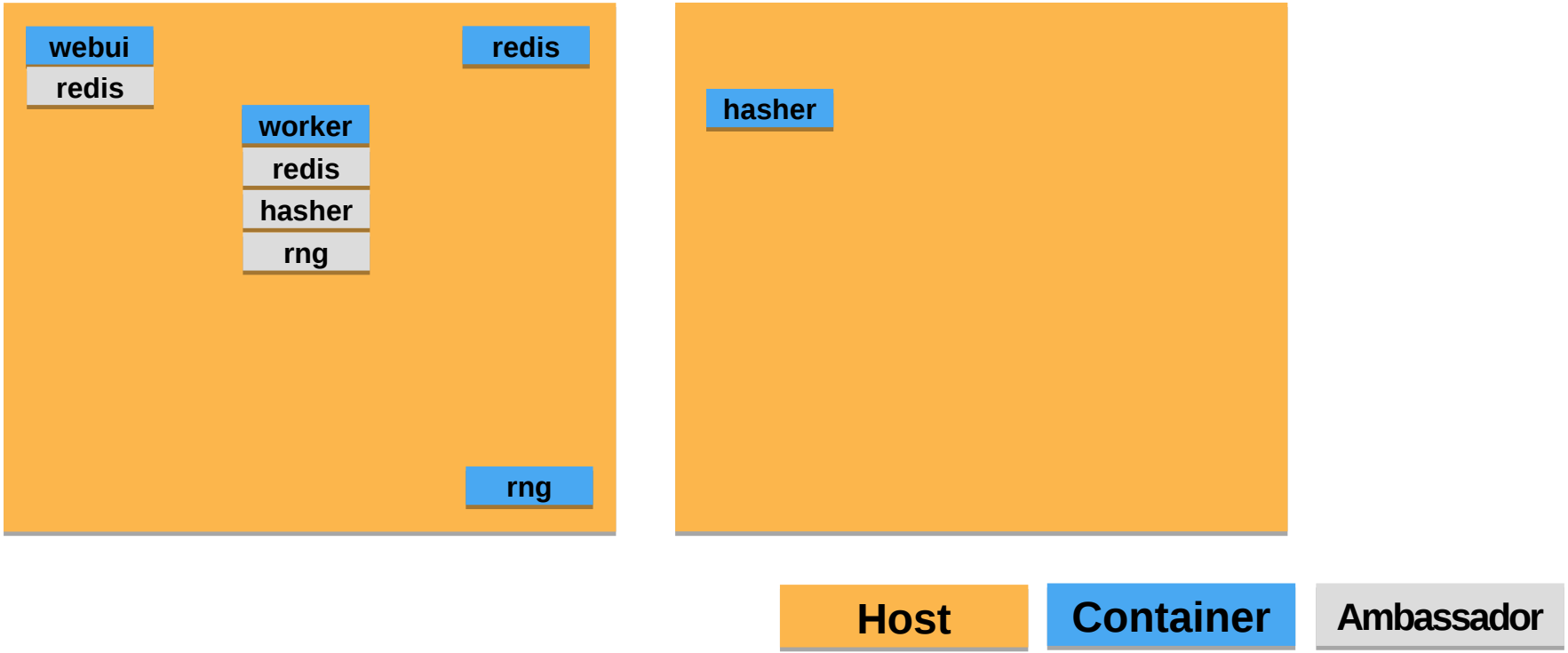
Scaling “dockercoins” on Swarm

- “docker-compose up” — containers are unwired



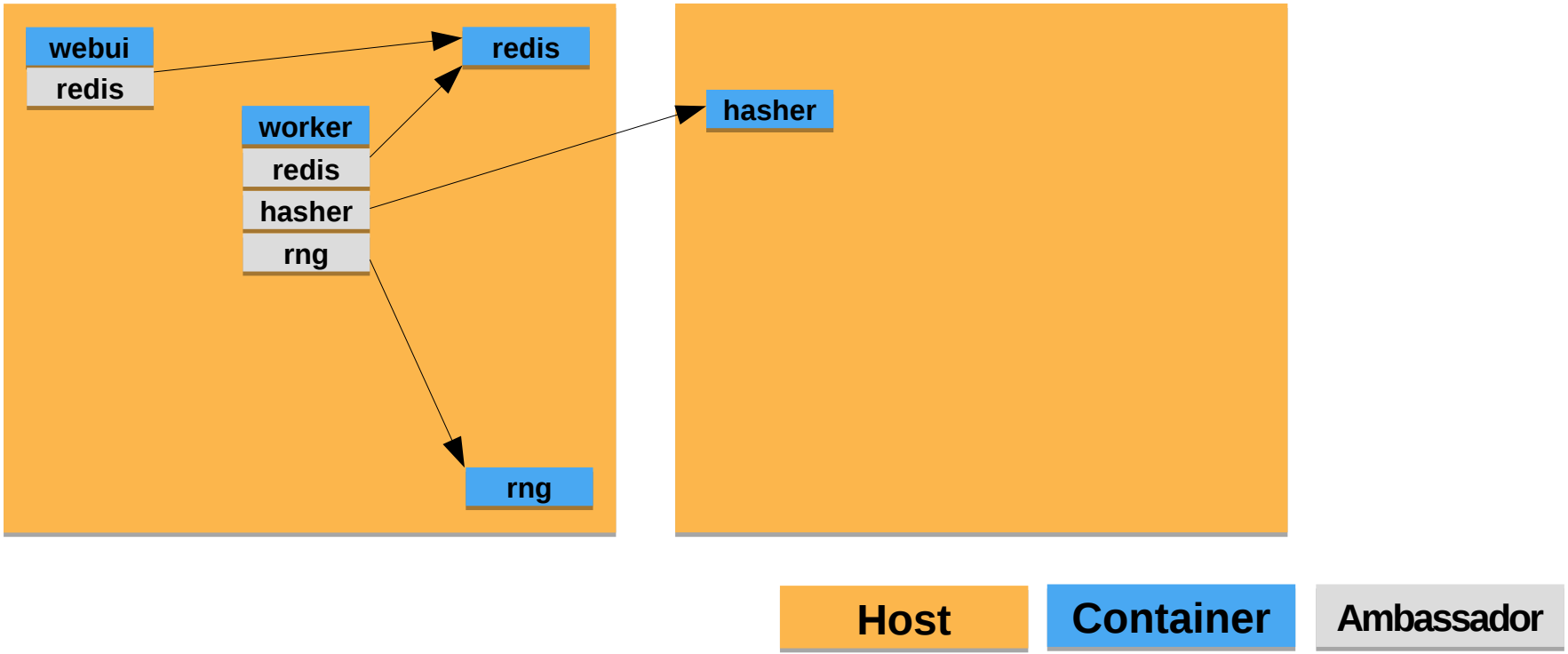
Scaling “dockercoins” on Swarm

- Create ambassadors for all containers needing them



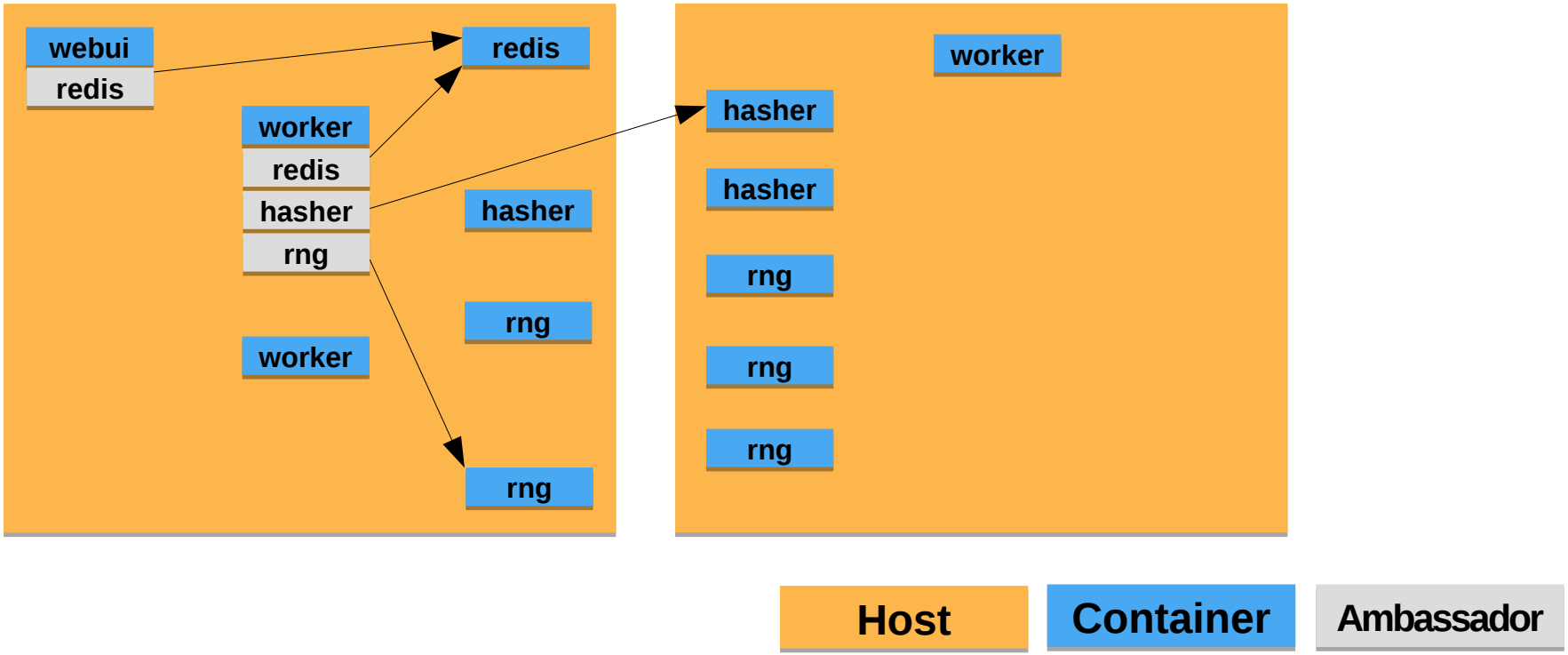
Scaling “dockercoins” on Swarm

- Configure ambassadors: the app is up and running



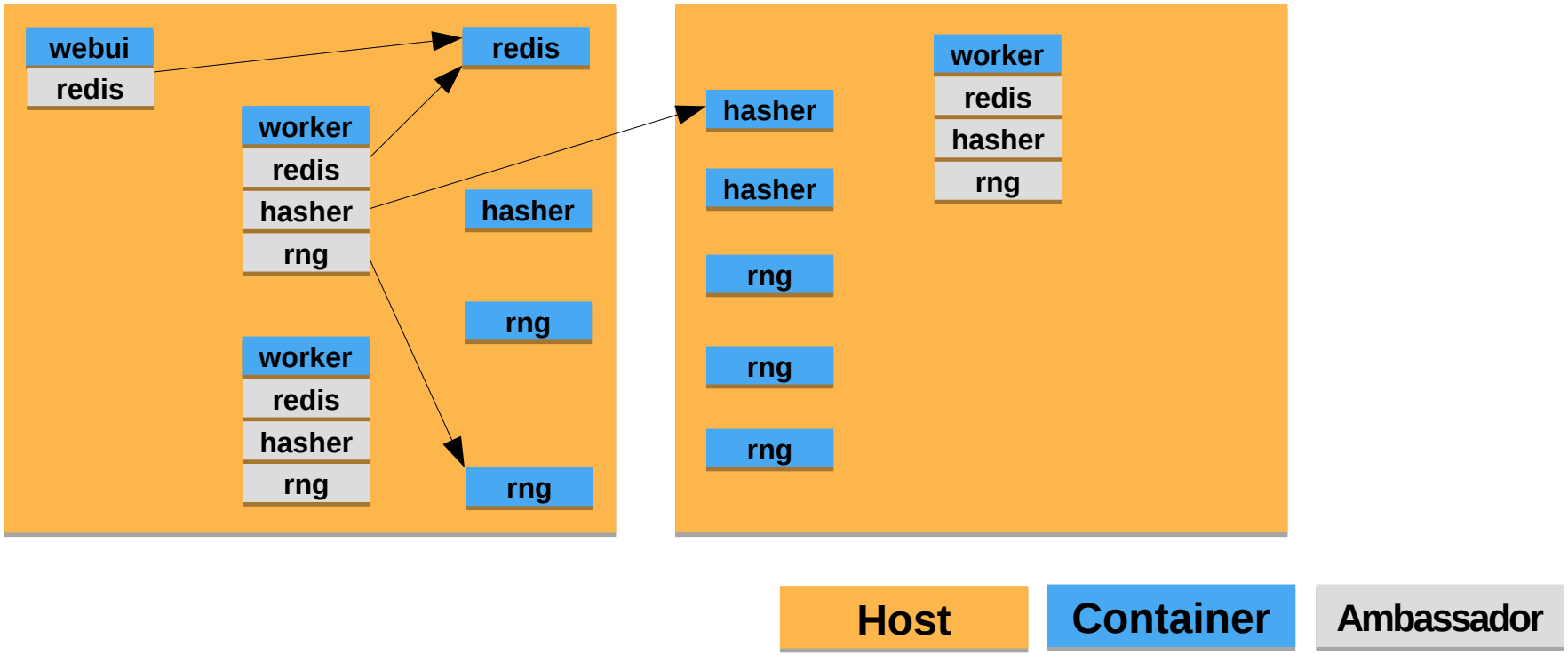
Scaling “dockercoins” on Swarm

- “docker-compose scale”



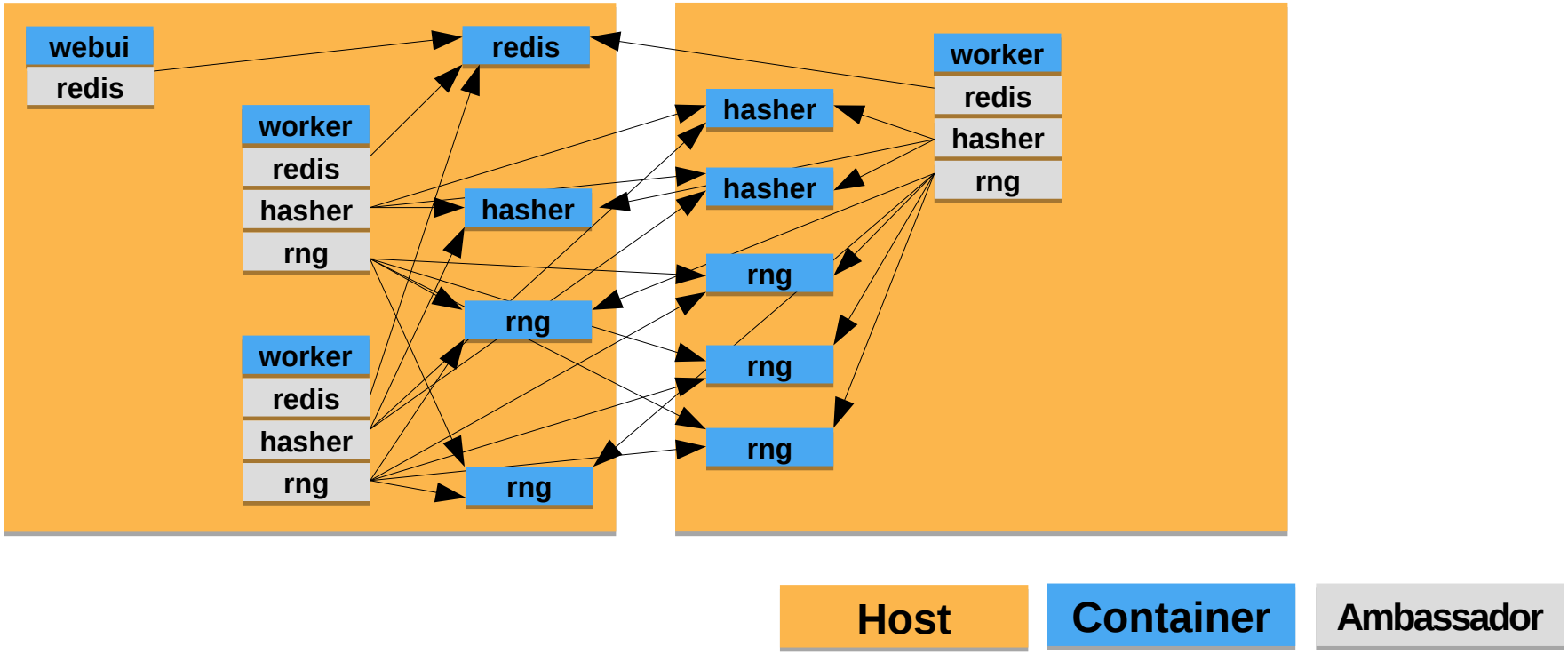
Scaling “dockercoins” on Swarm

- Creation of new ambassadors



Scaling “dockercoins” on Swarm

- Configuration of new ambassadors



Remarks

- Yes, that's a lot of ambassadors
- They are very lightweight, though (~1 MB)
  docker stats $(docker ps | grep hamba | awk '{print $1}')
- Ambassadors *do not* add an extra hop
  - they are local to their client (virtually zero latency)
  - better efficiency than external load balancer
  - if the ambassador is down, the client is probably down as well

Recap

The Docker Compose workflow

- Application portability between:
  - Docker Compose + Docker Toolbox
  - Docker Compose + Docker Swarm
  - ECS CLI + ECS
- Interface points:
  - Compose file
  - Docker Registry

ECS and Swarm highlights

- Both offer easy provisioning tools
- ECS = AWS ecosystem
  - integrates with offerings like IAM, ELB…
  - provides health-checks and self-healing
- Swarm = Docker ecosystem
  - offers parity with local development environments
  - exposes real-time events stream through Docker API
- Both require additional tooling for builds
  (Swarm has preliminary build support)
- Both require extra work for plumbing / service discovery

Future directions, ideas ...

- We would love your feedback!
- App-specific ambassadors
  (SQL bouncers, credential injectors...)
- Automatically replace services using official images:
  - redis, memcached → elasticache
  - mysql, postgresql → RDS
  - etc.

Other improvements

- Listen to Docker API events stream,
  detect containers start/stop events
  - automatically configure load balancers (ehazlett/interlock)
  - insert containers into a config database (gliderlabs/registrator)
- Overlay networks
  (offers direct container-to-container communication)
  - 3rd party: weave, flannel, pipework
  - Docker network plugins (experimental.docker.com)

Thank you!

Questions?

Remember to complete your evaluations!

Related sessions

- **CMP302** - Amazon EC2 Container Service: Distributed Applications at Scale
- **CMP406** - Amazon ECS at Coursera: Powering a general-purpose near-line execution microservice, while defending against untrusted code
- **DVO305** - Turbocharge Your Continuous Deployment Pipeline with Containers
- **DVO308** - Docker & ECS in Production: How We Migrated Our Infrastructure from Heroku to AWS (Remind)
- **DVO313** - Building Next-Generation Applications with Amazon ECS (Meteor)

Code repositories:

https://github.com/aws/amazon-ecs-cli
https://github.com/jpetazzo/dockercoins
https://github.com/jpetazzo/trainingwheels
https://github.com/jpetazzo/orchestration-workshop

Videos:

https://www.youtube.com/watch?v=g-g94H_AiOE
https://www.youtube.com/watch?v=sk3yYh1MgE0
https://www.youtube.com/watch?v=O3Bps01THBQ
https://www.youtube.com/watch?v=LFjwusorazs
https://www.youtube.com/watch?v=KqEpIDFxjNc

Note: videos just include the installation and deployment processes.
I'll make videos of the other demos if there's enough demand for it!

Follow us on Twitter:

@docker @jpetazzo