Docker Dojo

Doug Davis dug@us.ibm.com

Morgan Bauer mbauer@us.ibm.com

IBM

Open by design™

Agenda

- Introduction to Containers & Docker
- DevOps: Configuration Management
- DevOps: Building Value
- DevOps: Scaling and Reliability

Preface

- Our goal: Learn about Docker

- But, as we learn about Docker, ask yourself:
  - Does Docker make the DevOps end-to-end experience better?
  - How can a customer's existing workflow leverage this technology?
  - Can do we this with VMs?
  - Ask lots of questions!

IBM

Open by design™

Introduction to Containers and Docker

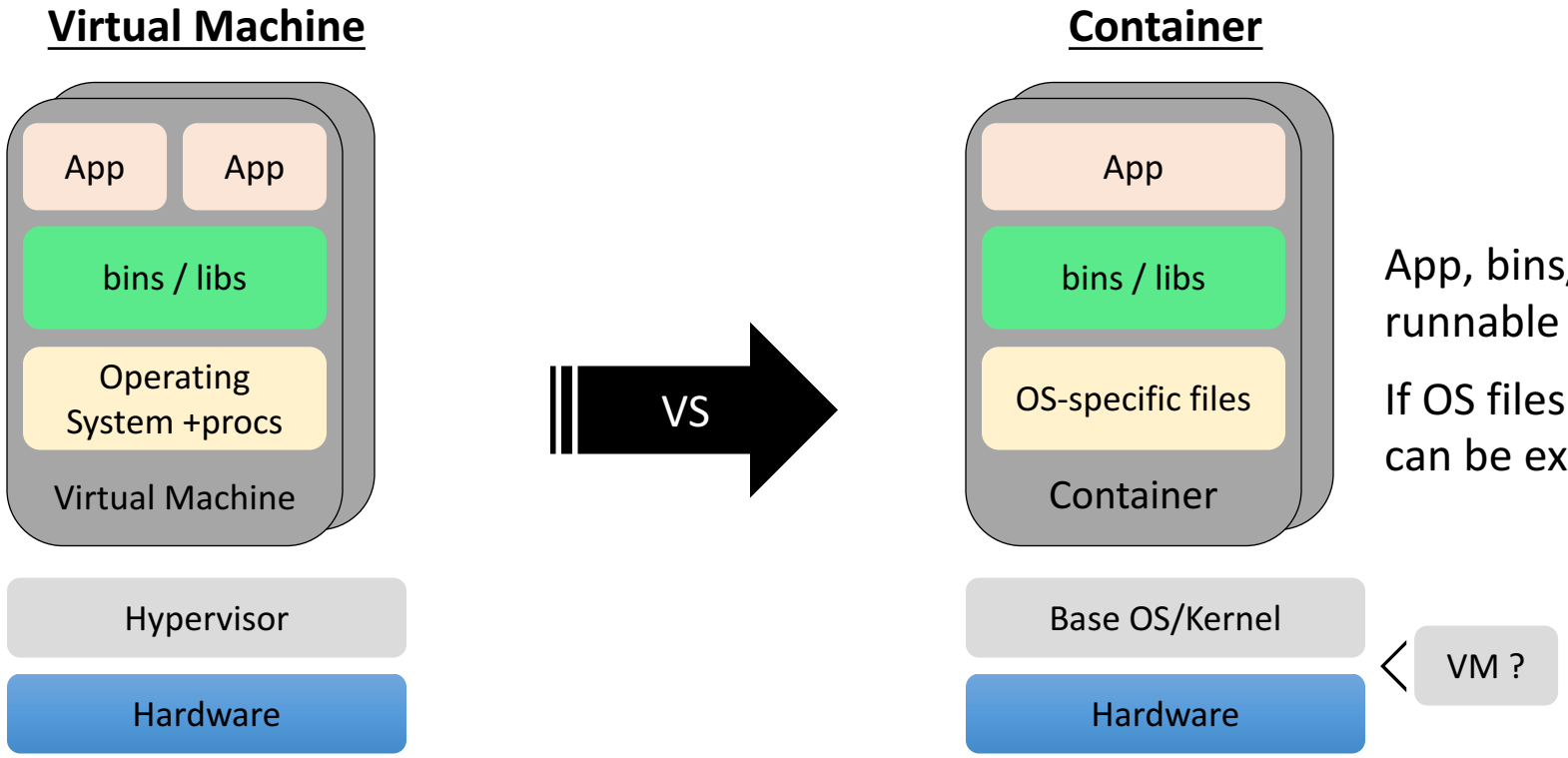
What is Docker?

- At its core, Docker is tooling to manage containers
  - Docker is not a technology, it’s a tool or platform
  - Simplified existing technology to enable it for the masses
- But, let's first discuss containers...

What are Containers?

- A group of processes run in isolation
  - Similar to VMs but managed at the process level
  - All processes MUST be able to run on the shared kernel
- Each container has its own set of "namespaces" (isolated view)
  - **PID** - process IDs
  - **USER** - user and group IDs
  - **UTS** - hostname and domain name
  - **NS** - mount points
  - **NET** - Network devices, stacks, ports
  - **IPC** - inter-process communications, message queues
  - **cgroups** - controls limits and monitoring of resources
- Docker gives it its own root filesystem

VM vs Container



Each VM has its own OS

Containers share the same base Kernel

App, bins/libs/OS must all be runnable on the shared kernel

If OS files aren't needed they can be excluded.

Why Containers?

- Fast startup time - only takes milliseconds to:
  - Create a new directory
  - Lay-down the container's filesystem
  - Setup the networks, mounts, ...
  - Start the process
- Better resource utilization
  - Can fit far more containers than VMs into a host

What is Docker again?

- **Tooling** to manage containers
  - Containers are not new
  - Docker just made them easy to use

- Docker creates and manages the lifecycle of containers
  - Setup filesystem
  - CRUD container
    - Setup networks
    - Setup volumes / mounts
    - Create: start new process telling OS to run it in isolation

Our First Container

```
$ docker run ubuntu echo Hello World
Hello World
```

- What happened?
  - Docker created a directory with a "ubuntu" filesystem (image)
  - Docker created a new set of namespaces
  - Ran a new process: `echo Hello World`
    - Using those namespaces to isolate it from other processes
    - Using that new directory as the "root" of the filesystem (`chroot`)
  - That's it!
    - Notice as a user I never installed "ubuntu"
  - Run it again - notice how quickly it ran

ssh-ing into a container - fake it...

```
$ docker run -ti ubuntu bash
root@62deec4411da:/# pwd
/
root@62deec4411da:/# exit
$
```

- Now the process is "bash" instead of "echo"
- But its still just a process
- Look around, mess around, its totally isolated
  - rm /etc/passwd – no worries!
  - MAKE SURE YOU'RE IN A CONTAINER!

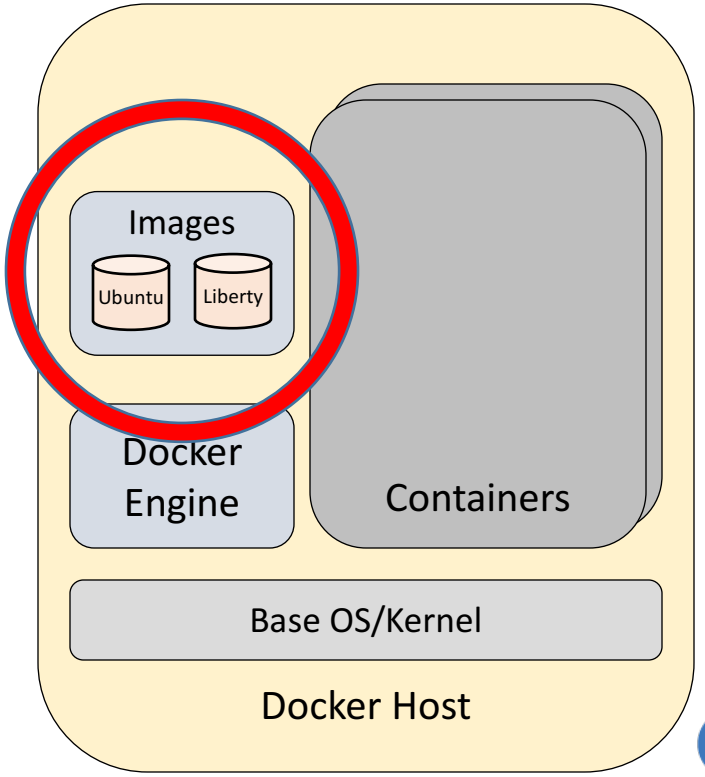
A look under the covers

```
$ docker run ubuntu ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 14:33 ?        00:00:00 ps -ef
```

- Things to notice with these examples
  - Each container only sees its own process(es)
  - Running as "root"
  - Running as PID 1

Docker Images

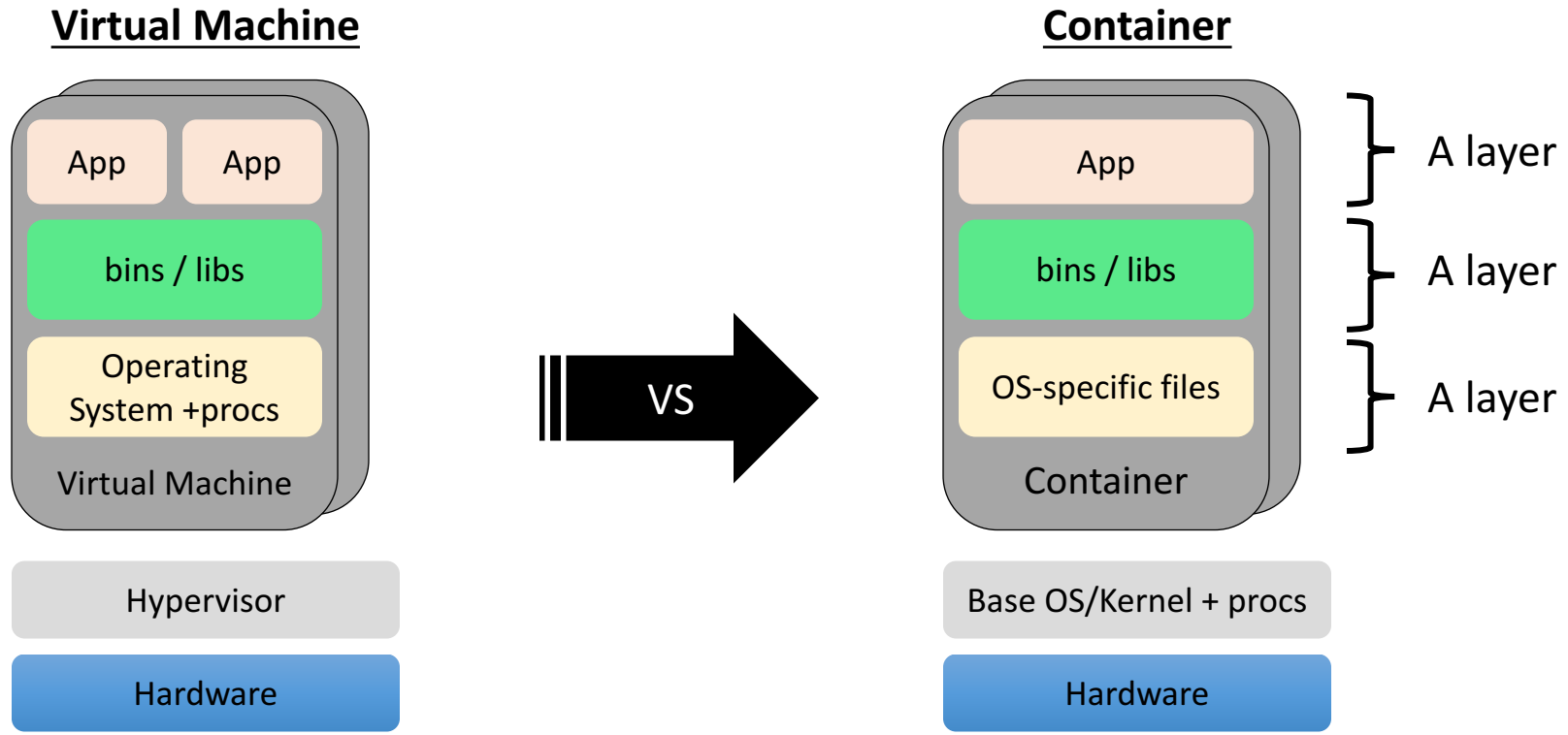
- Tar file containing a container's filesystem + metadata
- For sharing and redistribution
  - Global/public registry for sharing: DockerHub
- Similar, in concept, to a VM image



Docker special sauce: Layers

- But first, let's compare VMs and Containers one more time...

VM vs Container: Notice the layers!

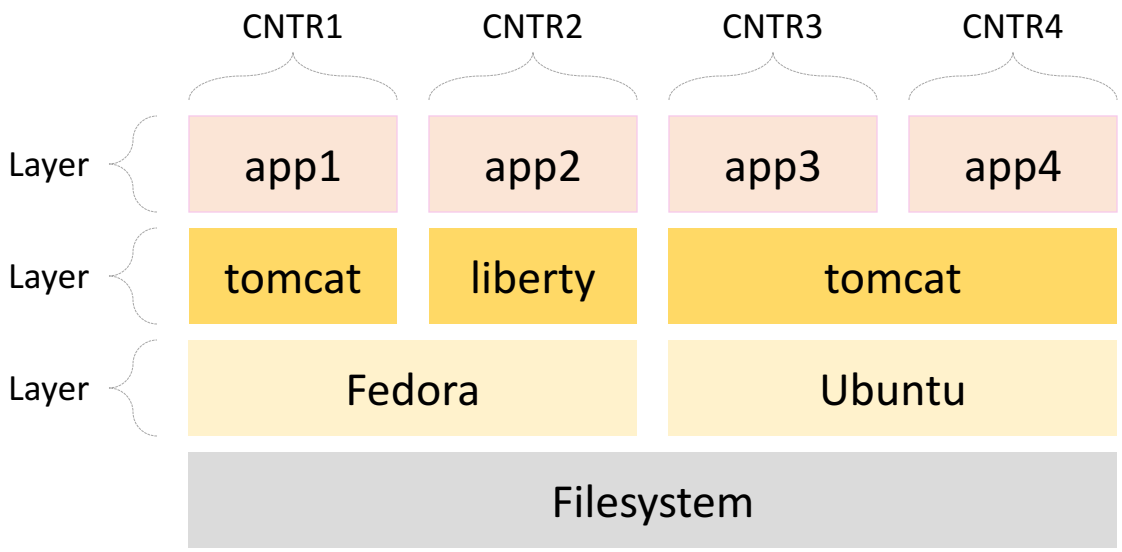


Each VM has its own OS

Containers share the same base Kernel

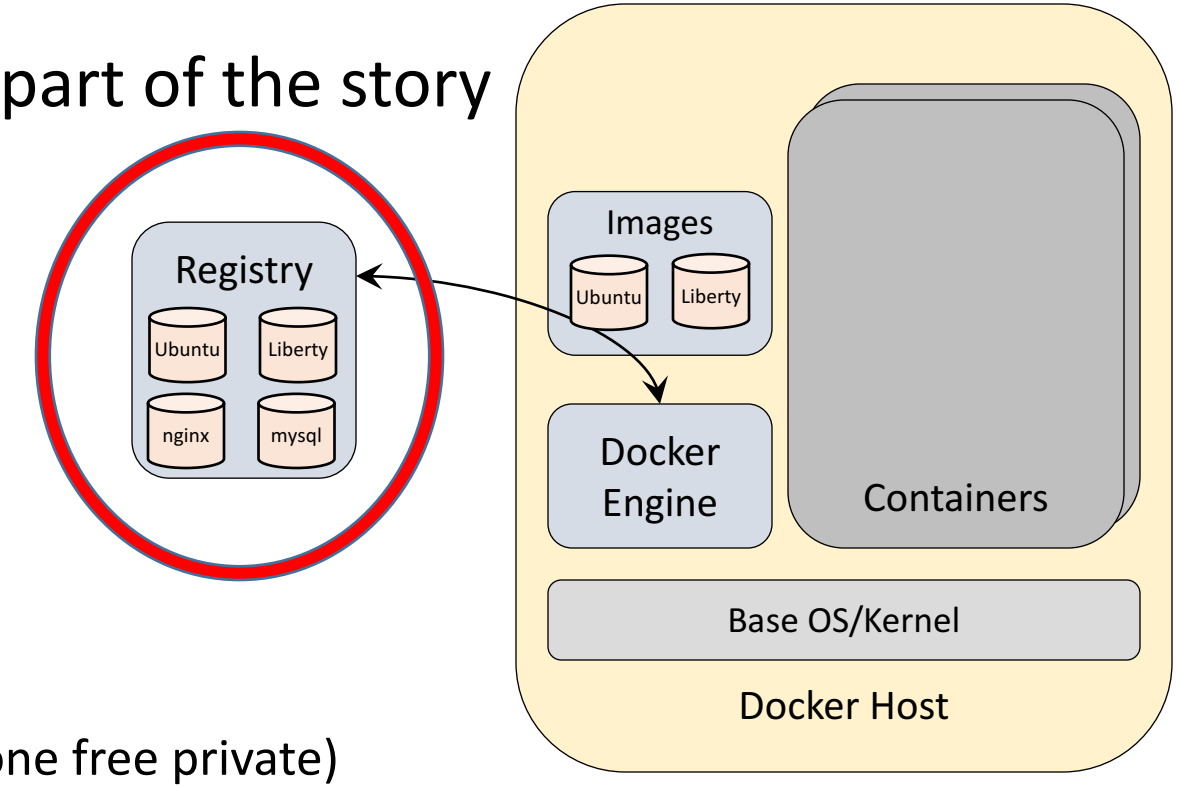
Shared / Layered / Union Filesystems

- Docker uses a copy-on-write (union) filesystem
- New files(& edits) are only visible to current/above layers
- Layers allow for reuse
  - More containers per host
  - Faster start-up/download time
- Images
  - Tarball of layers
- Think: Transparencies on projector



Docker Registry

- Creating and reusing images is only part of the story
- Sharing them is the other



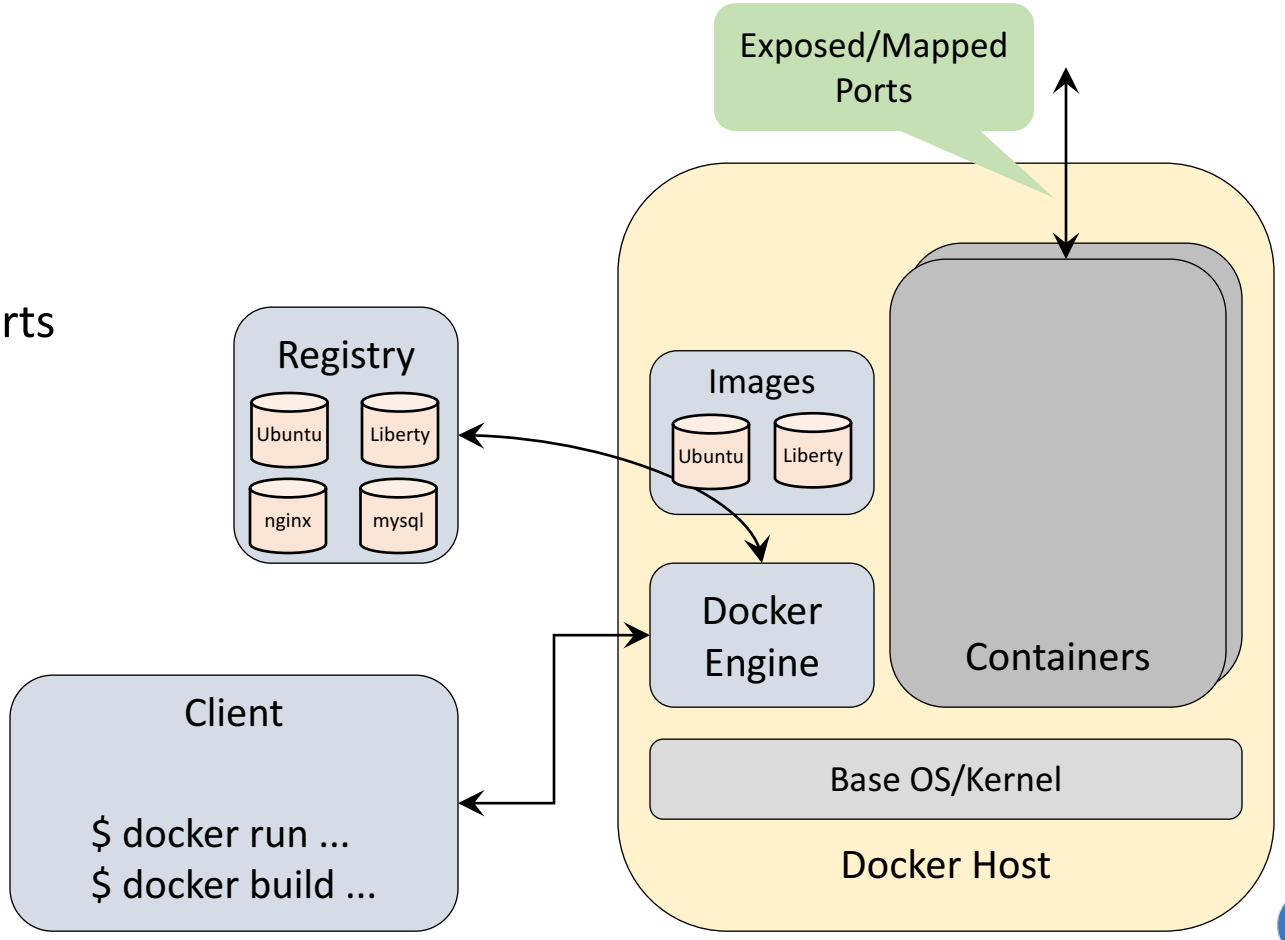
- DockerHub - http://hub.docker.com
  - Public registry of Docker Images
  - Hosted by Docker Inc.
  - Free for public images, pay for private ones (one free private)
  - By default docker engines will look in DockerHub for images
  - Browser interface for searching, descriptions of images

Multi-Architecture Support

- Before: **Docker runs everywhere!** (as long as its x86/Linux)
- Now: Docker daemon has multi-architecture support
  - Docker builds for Power, Z, ARM - Linux
  - Windows CLI built in community, Windows daemon built by Microsoft
- Registry Multi-architecture support is available
  - Engine and OSS Registry code and DockerHub supports it
  - Docker CLI doesn't provide a nice UX yet, but there are tools available
- Engine when pulling down an image:
  - Sends host's arch & OS along with the image tag
  - Registry will find image+arch+OS

Docker Component Overview

- Docker Engine
  - Manages containers on a host
  - Accepts requests from clients
    - REST API
  - Maps container ports to host ports
    - E.g. 80 → 3582
- Images
- Docker Client
  - Drives engine
  - Drives "builder" of Images
- Docker Registry
  - Image DB



Summary

- Docker is just a tool to manage containers
  - Key concepts: Containers, Docker Engine, Images, Registries
- Docker value-add:
  - An excellent User Experience
  - Image Layers
  - Easily shared images via DockerHub
- Why? When compared to VMs:
  - Better resource utilization - CPU, Memory, Disk
  - Faster start-up times
  - Easier tooling/scripting via Docker CLI
- Discussion / Questions?

Quiz!

- What's the difference between a container and an image?

- Answer:
  - An image is a tar of a filesystem
  - A container is a filesystem + a set of processing running in isolation

DevOps Enablement: Configuration Management

Topics

- Configuration Management

- Ensuring developers and stages of the CI pipeline have the correct environment can be a challenge
  - Install variants based on machines
  - Wrong version of products installed

Scenario

- A new developer has joined the team

- They already have:
  - Ubuntu VM with Docker installed
  - "git clone" of the source code for the project:

    `~/myapp` in the provided VM

- We need to get them up and running as quickly as possible
  - Without installing anything else!

The setup

```
$ cd myapp
```

```
$ cat Makefile
myapp: myapp.go
    go build -tags netgo -installsuffix netgo -o myapp myapp.go
```

- There's nothing here about Docker, just a normal compile step

Verify we're missing our dev environment

```
$ make
go build -tags netgo -installsuffix netgo -o myapp myapp.go
make: go: Command not found
Makefile:2: recipe for target 'myapp' failed
make: *** [myapp] Error 127
```

Solution

- Our IT department has provided a Docker image called "golang"
- This image has the go compiler installed
- Let's use this image to do our build

Using the "golang" image

- Abstractly:
  - Create a new container using the "golang" image
  - Make our source code available inside of the container
  - Build our application in the container
  - Make the executable available outside of the container
    - Otherwise the results will be lost when the container is deleted

Using the "golang" image

- Technically the IT department would setup the Makefile like this **(do not type this)**:

```
docker run golang -v $(PWD):/src  -w /src
   go build -tags netgo -installsuffix netgo -o myapp myapp.go
```

- Summary:
  - **`docker run golang`** # Creates a container based on "golang"
  - **`-v $(PWD):/src`** # Mounts current directory into container at /src
  - **`-w /src`** # Docker will "cd" to /src before starting process
  - Notice that we didn't modify the normal developer's process, we just wrapped it with Docker

- Quiz: where does the output go?

In action

```
$ make -f Makefile1
docker run -v /home/user/myapp:/src -w /src golang \
  go build -tags netgo -installsuffix netgo -o myapp myapp.go
```

- Built "myapp" in the container
- But also stored in current directory due to the mount

Test the Build

```
$ ./myapp 8080
Will show:
<pre><b>v1.0 Host: docker  Date: 2016-09-04
05:27:42.582058185 -0700 PDT</b>
127.0.0.1
192.168.59.147
172.17.0.1
172.19.0.1
172.18.0.1
172.20.0.1

Listening on: 0.0.0.0:8080


$ curl localhost:8080     # Test it from another window
ctrl-c                    # To stop it
```

Summary

- Developer prerequisites:
  - VM + Docker
  - Source code: `git clone ...`
- Concepts:
  - Configuration management/headaches are lessened or removed
  - Developer never had to "install" any dev tooling
  - IT department provided a "standard development environment"
- Discussion / Questions?

Discussion Point

```
$ ls -l
total 5584
-rw-r--r-- 1 user user      44 Sep  3 11:27 Makefile
-rwxr-xr-x 1 root root 5706234 Sep  4 06:51 myapp
-rw-r--r-- 1 user user     887 Sep  3 11:39 myapp.go
```

- "root" owns "myapp"
- This is because we're running as "root" in the container

- Solution: Modify the Makefile to include the user to run as:

```
docker run -u 1000:1000 -v $(PWD):/src -w /src golang \
    go build -tags netgo -installsuffix netgo -o myapp myapp.go
```

  - **$ make -f Makefile2** if you want to try it, but "**rm -f myapp**" first

Discussion Point

- Did I lie about the prerequisites being just VM+Docker?
- What about "git" ?

```
$ which git                     # should be empty
$ command -V git                # to show the 'alias'
alias git='docker run -ti -u 1000:1000 \
    -v /home/user:/tmphome -v $PWD:/wd \
    -w /wd -e HOME=/tmphome git'
```

- Git is being run from a container!
- Overrides $HOME so .gitconfig maps to ~/.gitconfig on host
- Just need to provide the dev with alias - so just a small white-lie
- Containers/isolation/sharing of images isn't just for "real work/services"

DevOps Enablement: Building Value

Topics

- Becoming a creator, and exporter of content, via Docker Images
- Adding value to existing Images
- Sharing this content via Docker Registries
- Becoming part of the value-add chain

Scenario

- Sharing the result of a build with the rest of the CI/CD pipeline
- We have the output of a product build ("myapp" executable)
- We need to build a Docker Image and share it
- NOTE: make sure you're in the "myapp" directory and have build "myapp" executable:

```
$ rm -f myapp
$ make -f Makefile2
```

Creating a Docker Image - Manually

- Create a Docker Image by "snapshotting" a container
- First we need to create a new container for our application

```
$ docker create ubuntu
5ed983843bbaef1062096e456e6fd931e6f24e9399d7c801adc7f
```

- Now let's copy our executable into it:

```
$ docker cp myapp 5ed98:/myapp
```

- Finally, snapshot the container as a Docker Image - called "myapp"

```
$ docker commit -c "entrypoint /myapp" 5ed98 myapp
sha256:7c640789dae5607c868a56883189d6c72478eff1080a67
```

Test the Image

```
$ docker run -ti myapp
Will show:
<pre><b>v1.0 Host: 165dcbc3e6f8  Date: 2016-09-05 02:47:50.2...</b>
127.0.0.1
172.17.0.2

Listening on: 0.0.0.0:80
```

- In another window:

```
$ curl 172.17.0.2
<pre><b>v1.0 Host: b8d73b85cc04  Date: 2016-09-05 02:53:49.803922...</b>
127.0.0.1
172.17.0.2
```

- Stop the app by pressing: **ctrl-c** in first window

Discussion

- Can we expose this container at the host level so others can access it?

- Yes, by mapping port 80 in the container to a unique port on the host

```
$ docker run -d -ti -p 9999:80 myapp
4b08d035deb6135eff60babd1368ab47c0c1f1d09a8ddf3f9417e7e4c4

$ curl localhost:9999
<pre><b>v1.0 Host: 4b08d035deb6  Date: 2016-09-05
03:14:31.713...</b>
127.0.0.1
172.17.0.2

$ docker rm 4b08
Failed to remove container ...

$ docker rm -f 4b08
4b08
```

Creating a Docker Image - With Docker Build

- Docker provides a "build" feature

- Uses a "Dockerfile"
  - Like a "Makefile", a list of instructions for how to construct the container

```
$ cat Dockerfile
FROM ubuntu
ADD myapp /
EXPOSE 80
ENTRYPOINT /myapp
```

Creating a Docker Image - With Docker Build

```
$ docker build -t myapp .
Sending build context to Docker daemon 5.767 MB
Step 1/4 : FROM ubuntu
 ---> ff6011336327
Step 2/4 : ADD myapp /
 ---> b867e19a859b
Removing intermediate container ea699ecc51a0
Step 3/4 : EXPOSE 80
 ---> Running in 85c240f03ae9
 ---> 5d8e53bbf9e4
Removing intermediate container 85c240f03ae9
Step 4/4 : ENTRYPOINT /myapp
 ---> Running in f318d82c2c38
 ---> 684c6c2572ff
Removing intermediate container f318d82c2c38
Successfully built 684c6c2572ff
```

Test the image - With Auto-Port Allocation

```
$ docker run -tidP myapp
469221295fae1b57615286ec7268272e3d3583c12ea66e14b2

$ docker ps
CONTAINER ID   IMAGE   COMMAND                  CREATED          STATUS
                PORTS                       NAMES
469221295fae   myapp   "/bin/sh -c /myapp"   4 seconds ago    Up 4
seconds         0.0.0.0:32768->80/tcp       clever_ardinghelli

$ curl localhost:32768
<pre><b>v1.0 Host: 469221295fae  Date: 2016-09-05 03:34:49....</b>
127.0.0.1
172.17.0.3

$ docker rm -f 469
469
```

Sharing the Image

- The "myapp" image is only in our local image cache
- To distribute it we need to upload it to a shared registry
- First, let's start a registry - locally, but pretend its public

```
$ regStart
```

Naming Images

- Before uploading an image, its name must include the registry
- General syntax of image names:
  - `[[registry/][namespace/]]name[:tag]`
  - `E.g. docker:5000/myapp:1.0` `# "docker" is our hostname`
- Registry: host:port - presence of ":" disambiguates from "namespace"
- Namespace: user, owner
- Tag: typically a version string - defaults to "latest"

Preparing our Image

```
$ docker build -t docker:5000/myapp:1.0 .
Sending build context to Docker daemon 5.767 MB
Step 1/4 : FROM ubuntu
 ---> ff6011336327
Step 2/4 : ADD myapp /
 ---> Using cache
 ---> b867e19a859b
Step 3/4 : EXPOSE 80
 ---> Using cache
 ---> 5d8e53bbf9e4
Step 4/4 : ENTRYPOINT /myapp
 ---> Using cache
 ---> 684c6c2572ff
Successfully built 684c6c2572ff
```

- Alternative:

```
$ docker tag myapp docker:5000/myapp:1.0
```

Pushing the Image

```
$ docker push docker:5000/myapp:1.0
The push refers to a repository [docker:5000/myapp]
5d1c38831713: Pushed
447f88c8358f: Pushed
df9a135a6949: Pushed
dbaa8ea1faf9: Pushed
8a14f84e5837: Pushed
latest: digest: sha256:71f76c1b360e340614a52bcfef2cb78d8f0aa3604 size: 1363
```

- Image is in the registry and can be used by other parts of the pipeline

```
$ docker run -ti docker:5000/myapp:1.0
$ docker pull docker:5000/myapp:1.0
```

Summary

- Can build an image via "docker build" or manually/scripted

- Share image via registries with the rest of the CI/CD pipeline
  - Images are not just for "products"
  - Move your testcases through the CI/CD pipeline as well
  - Anything you want to share can be in an Image

- Discussion / Questions?

Discussion Point

- Our Dockerfile started with: **`FROM ubuntu`** do we really need Ubuntu ?
- No, there is nothing in our app that uses the operating system

```
$ docker images | grep myapp
docker:5000/myapp  1.0   684c6c2572ff  7 hours ago  193.7 MB
```

- Instead our Dockerfile could use: **`FROM scratch`**
  - Let's do that so our image is smaller

```
$ docker build -f Dockerfile2 -t docker:5000/myapp:1.0 .
$ docker images | grep myapp
docker:5000/myapp  1.0  9b604f2e42da 7 seconds ago 5.84 MB
```

```
$ docker push docker:5000/myapp:1.0   # update our registry
```

Discussion Point

- What are **some** of the other instructions can we have in a Dockerfile?
  - RUN
  - HEALTHCHECK
  - COPY/ADD
  - CMD & ENTRYPOINT
  - LABEL
  - ENV/ARG
  - VOLUME
  - USER
  - WORKDIR

DevOps Enablement: Scaling and Reliability

Topics

- Take a single container/image and spread it
  - To provide redundancy, reliability and performance
- Can be complex to manage production-level deployments
  - Clustering
  - High-availability
  - Scaling
  - Versioning - with no downtime

Scenario

- Given an application/image we need to deploy it
  - With high-availability
- Then we need to upgrade it
  - With zero down-time

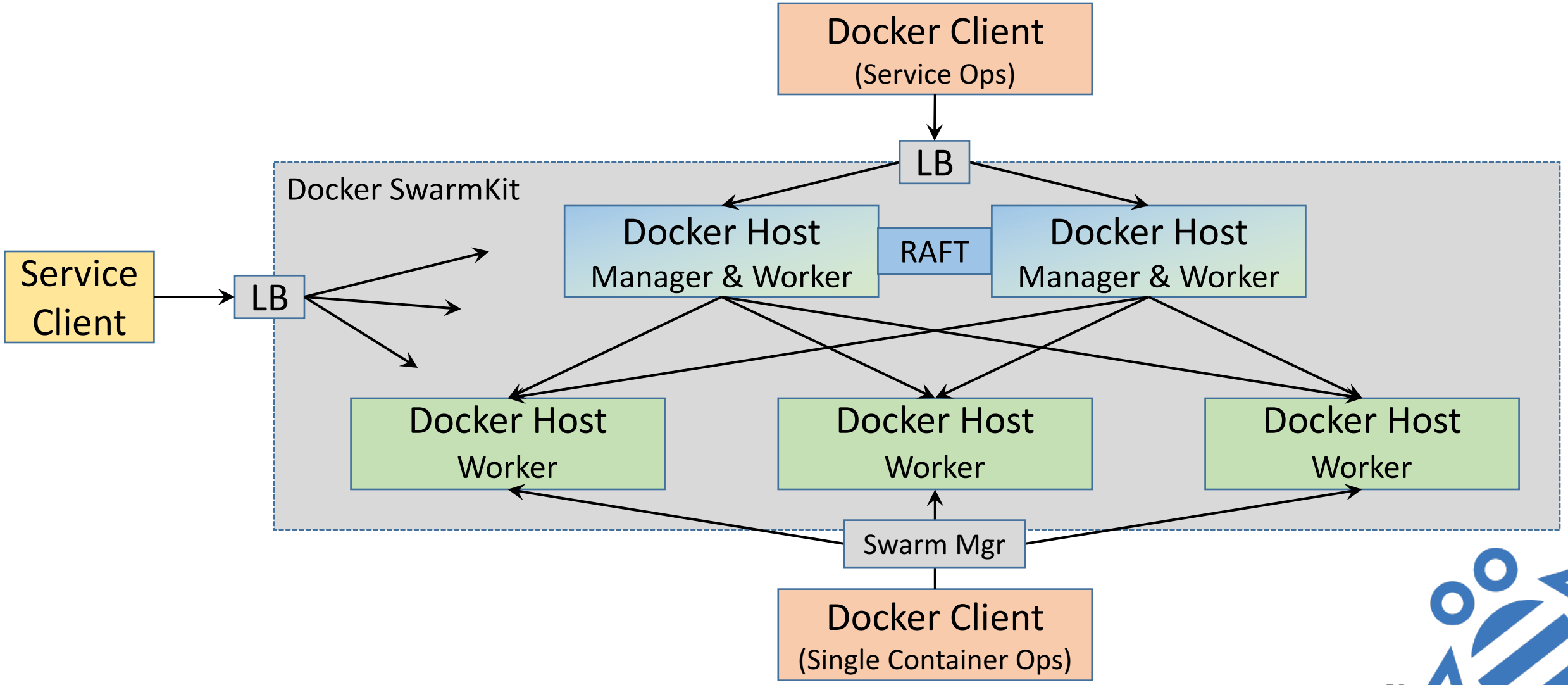
Docker Orchestration

- In v1.12 Docker introduced the notion of a "Service"
- A scaled cluster of Docker containers based on the same Image
- Across a cluster of Docker Hosts
- Built into the Docker engine

Terminology

- Worker
  - A Docker Host capable of accepting requests to manage single containers
  - New name for the existing Docker host/engine
- Manager
  - A Docker Host capable of accepting requests to manage "services"
  - Also acts as a Worker
- As a Docker client you need to know which to talk to
  - Single container management vs Service management

Workers, Managers and the "Mesh"

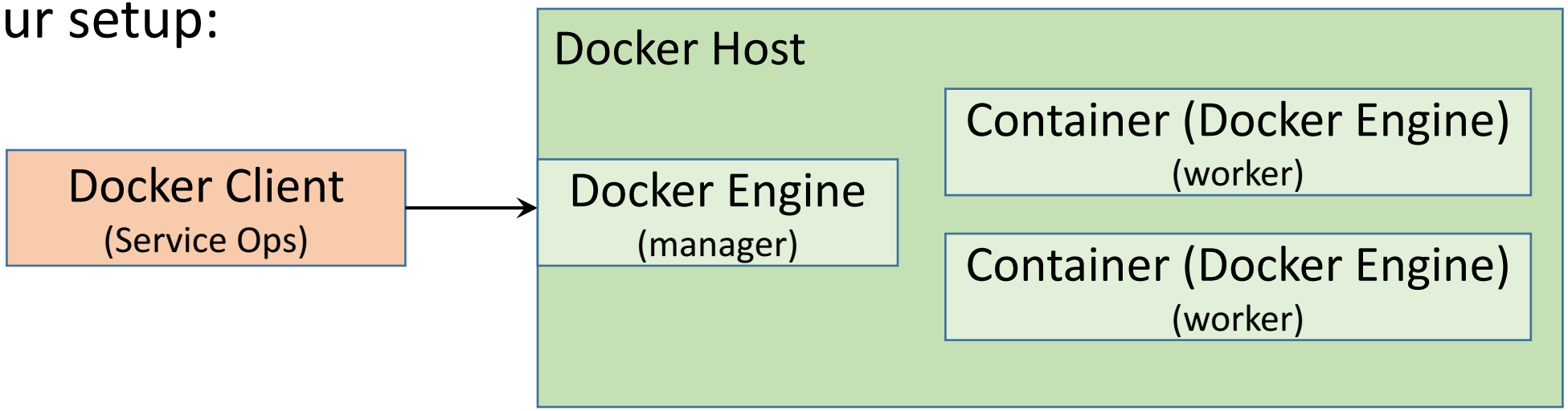


Extra Details

- Each "service" gets its own Virtual IP
- DNS resolution will map "service name" to "VIP"
- Introduces a declarative model with eventual consistency
- RAFT : in-memory distributed state store
- Shared port space across **all** nodes

Part 1 - Deploy our Docker Cluster

- Setup a Docker SwarmKit Cluster to host our application
- Our setup:




```
$ swarmDemo1
```

Part 2 - Deploy v1.0 of our app

- Deploys the application to the swarm
- Scale it to 3 instances

```
$ swarmDemo2
```

Part 3 - Build a v2.0 of the app

```
$ docker build -f Dockerfile3 -t docker:5000/myapp:2.0 .
```

- Dockerfile3:

```
FROM scratch
ADD myapp /
EXPOSE 80
ENV APP_VER=2.0
ENTRYPOINT /myapp
```

```
$ docker push docker:5000/myapp:2.0
```

Part 4 - Upgrade the app

In one window run:

```
$ docker exec –ti node3 watch -d -n 1 curl -s 127.0.0.1
```

This will continually hit the app, but do so by sending the request to node3.
Notice the version string starts out as "1.0".

In another window update the service:

```
$ docker service update --image=docker:5000/myapp:2.0 myapp
```

Notice the version string will eventually change to "2.0"

```
$ swarmDemo9        # to clean up
```

Summary

- Deploying, scaling and upgrading an app can be complicated
- Docker manages this for us with an "eventual consistency" model
  - We provide the "desired state"
- Discussion / Questions?