Docker Deep Dive

Daniel Klopp

The Talk

- I’m not telling you what fishing rod to use

The Talk

- I’m not telling you what fishing rod to use
- I’m helping you understand the fishing rod

The Talk

- I’m not telling you what fishing rod to use
- I’m helping you understand the fishing rod
- You can decide if you should fish with it

The Talk

- This is an intermediate deep dive.

The Talk

- This is an intermediate deep dive.
  - Don’t panic

The Talk

- This is an intermediate deep dive.
  - Don’t panic
- Each category starts easy, and assumes progressively more knowledge

The Talk

- This is an intermediate deep dive.
  - Don’t panic
- Each category starts easy, and assumes progressively more knowledge
  - Don’t panic!

The Talk

- This is an intermediate deep dive.
  - Don’t panic
- Each category starts easy, and assumes progressively more knowledge
  - Don’t panic!
- This is not a demo

The Talk

- This is an intermediate deep dive.
  - Don’t panic
- Each category starts easy, and assumes progressively more knowledge
  - Don’t panic!
- This is not a demo
- If you have questions, ask.

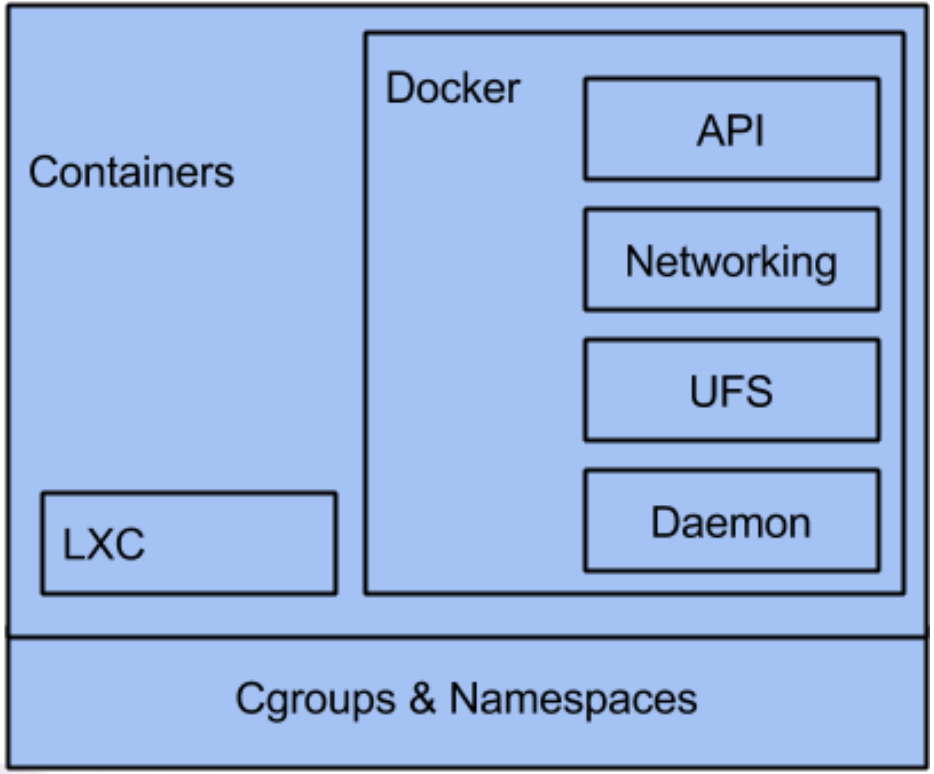
Assumptions

- I assume basic Linux knowledge
- I assume basic TCP/IP knowledge
- I assume basic Filesystem layout knowledge
- No knowledge of containers necessary

Who Am I?

- EE by training, DevOps by practice
- Worked in HPC at National Radio Astronomy Observatory
  - Including kernel development
- Taos, Unix / Scripting / DevOps handiman
- Unix Practice Leader at Taos Consulting

The Talk



TAOS

Outline

- What are containers
- Containers, hypervisors, jails, chroot
- Docker & Containers Intro
- Linux Container Implementation
  - LXC

Outline

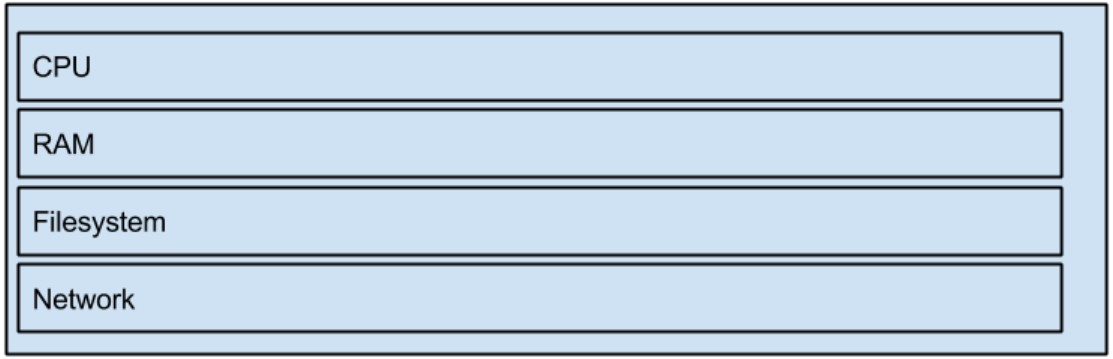
- Docker
- Docker Filesystem
- Docker Networking
- Docker API
- Docker Management Tools

Section
What Are Containers?

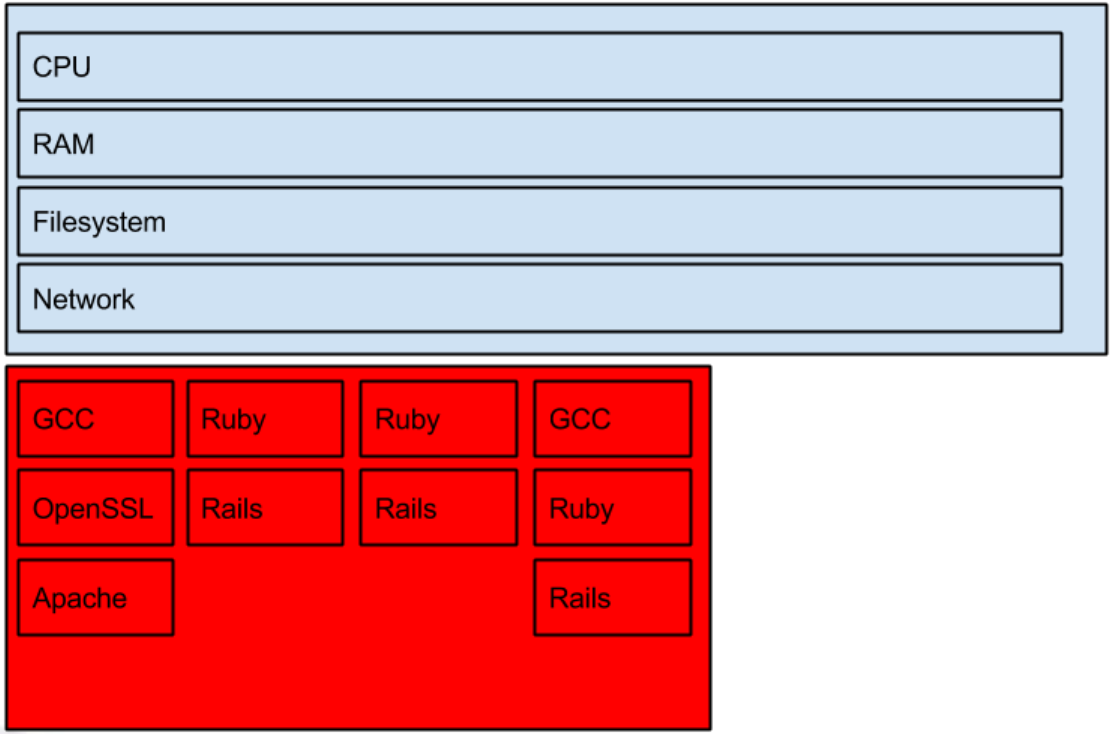
- A brief story
- Containers

A Story

- The good ol' days

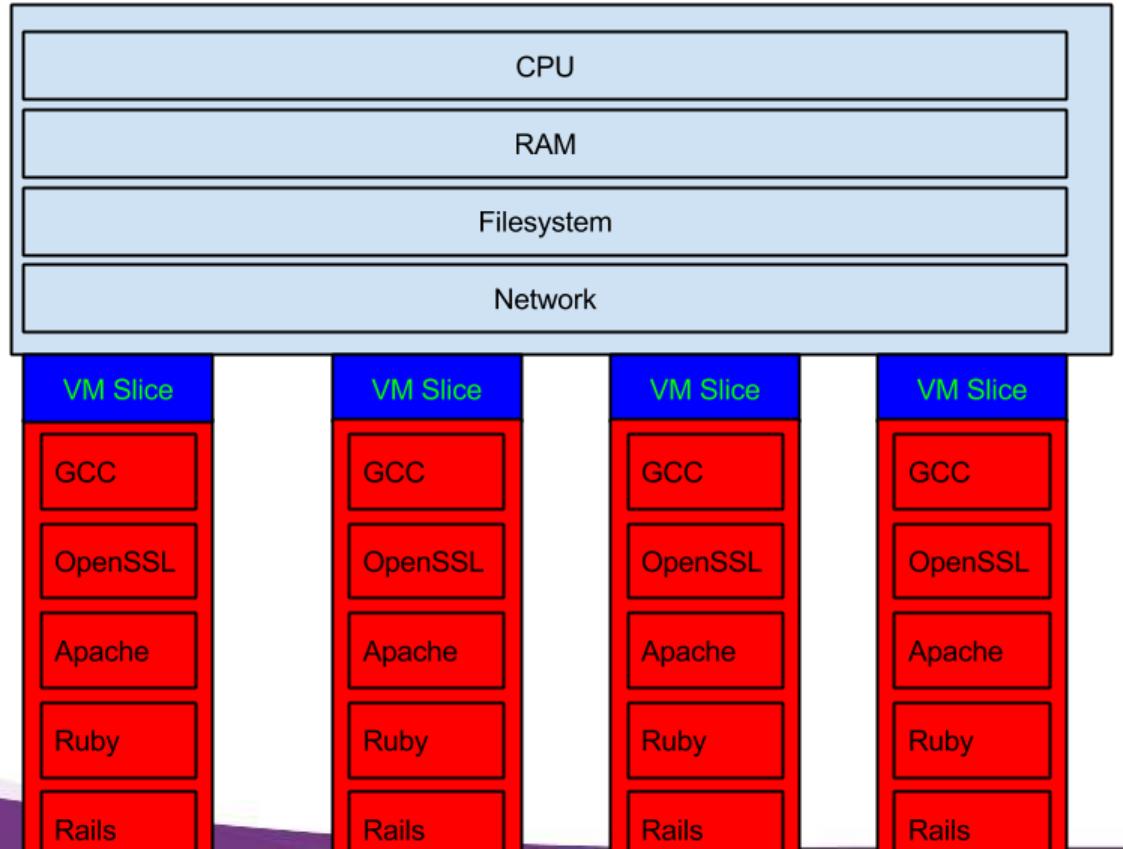


A Story


CPU
RAM
Filesystem
Network
GCC
Ruby
Ruby
GCC
OpenSSL
Rails
Rails
Ruby
Apache
Rails

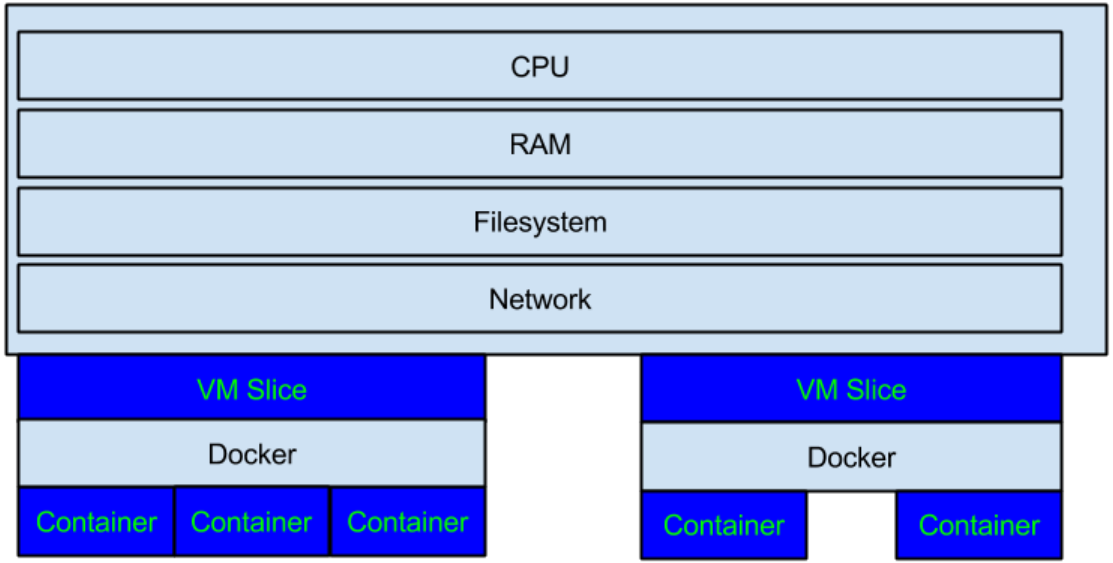
TAOS

A Story


CPU
RAM
Filesystem
Network
VM Slice
VM Slice
VM Slice
VM Slice
GCC
OpenSSL
Apache
Ruby
Rails
GCC
OpenSSL
Apache
Ruby
Rails
GCC
OpenSSL
Apache
Ruby
Rails
GCC
OpenSSL
Apache
Ruby
Rails

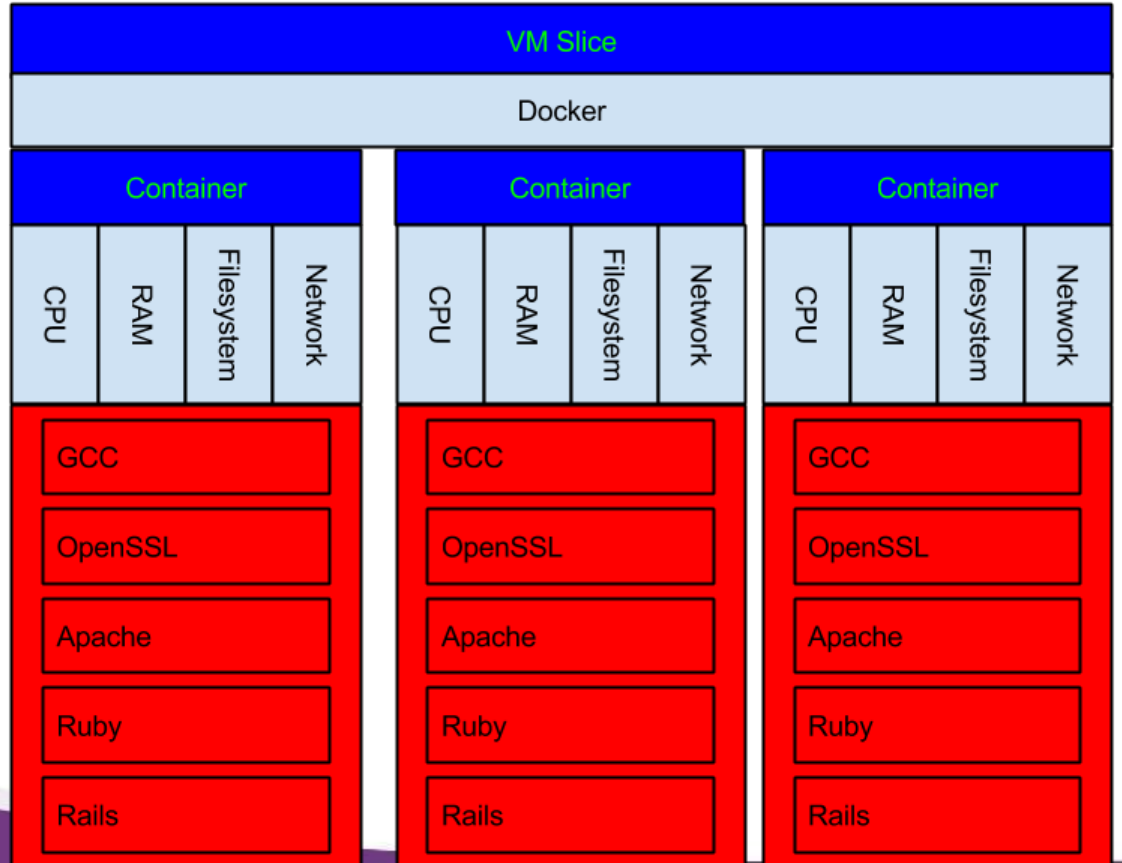
TAOS

A Story


CPU
RAM
Filesystem
Network
VM Slice
Docker
Container
Container
Container
VM Slice
Docker
Container
Container

TAOS

A Story


VM Slice
Docker
Container
Container
Container
CPU
RAM
Filesystem
Network
CPU
RAM
Filesystem
Network
CPU
RAM
Filesystem
Network
GCC
OpenSSL
Apache
Ruby
Rails
GCC
OpenSSL
Apache
Ruby
Rails
GCC
OpenSSL
Apache
Ruby
Rails

TAOS

How Is This Achieved?

- Namely
  - Linux Namespaces
  - Kernel cgroups
  - Union Filesystems

How Is This Achieved?

- Namely
  - Linux Namespaces
  - Kernel cgroups
  - Union Filesystems
- We'll review this foundation soon

Kernel Cgroups

- Started by Google
  - Paul Menage and Rohit Seth
- Merged in kernel 2.6.24 (2008)
- Provides
  - Resource Limitation
  - Prioritization
  - Accounting
  - Control

Linux Namespaces

- Isolation
  - Deployment
  - Security

Linux Namespaces

- Isolation
  - Deployment
  - Security
- We will hardly cover this.

Section
Containers, Hypervisors, Jails

- Operating Systems
- Hypervisors
- Jails
- Chroots
- Containers

Operating Systems

- Manage hardware resources
- Provide shared interfaces for programs
- Police programs
- Operating Systems
  - Android
  - Linux
  - Windows
  - BSD
  - Mac

TAOS

Hypervisors

- Hypervisors virtualize and manage hardware resources
- Popular Type I Hypervisors:
  - KVM
  - Xen
  - VMWare ESX(i)
- Popular Type II Hypervisors:
  - Virtualbox

Chroot

- In practice used to narrow filesystem view
- Cannot isolate process tree
- Cannot enforce resource limits
  - Filesystem only exception
- 20 customers, 20 chroots?
  - 1 misbehaving customer, all customers suffer
- Suffers from escalation insecurities

Jails

- BSD and Solaris
- Operating System level container

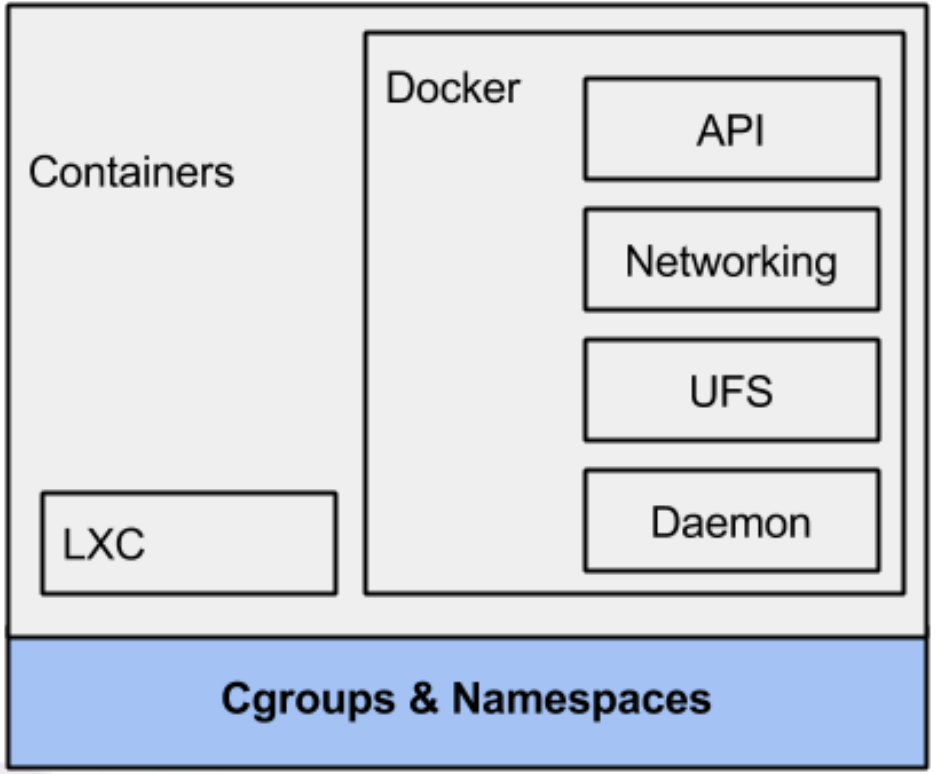
Containers

- Operating System level container
- Almost, but not quite, a Linux “jail”
  - (Don’t kill the messenger!)
- Provides meaningful resource controls
  - CPU
  - RAM
- Provides isolation

Section
Linux Container Implementation

- Cgroups
- Namespaces

Where We Are



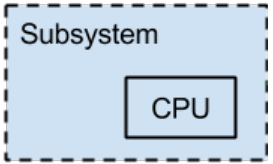
TAOS

Kernel Cgroups

- Hierarchical

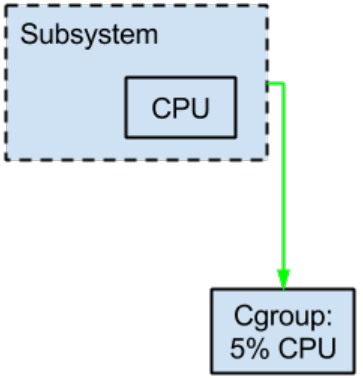
Kernel Cgroups

- Hierarchical
  - Top-level are “subsystems” or “controllers”



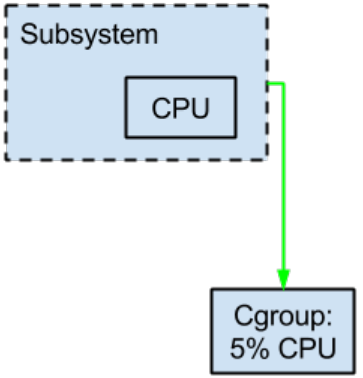
Kernel Cgroups

- Hierarchical
  - Top-level are “subsystems” or “controllers”
  - Lower-level are “cgroups”



Kernel Cgroups

- Hierarchical
  - Top-level are “subsystems” or “controllers”
  - Lower-level are “cgroups”
    - “cgroups” can be nested



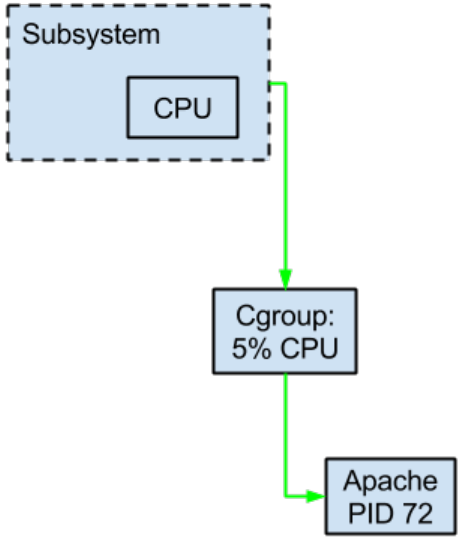
Subsystems

- Also known as “Controllers”
- They are the root of a hierarchy
- They control resources, such as
  - CPU
  - Memory
  - Network
- A “cgroup” is attached to a subsystem, forming a hierarchy

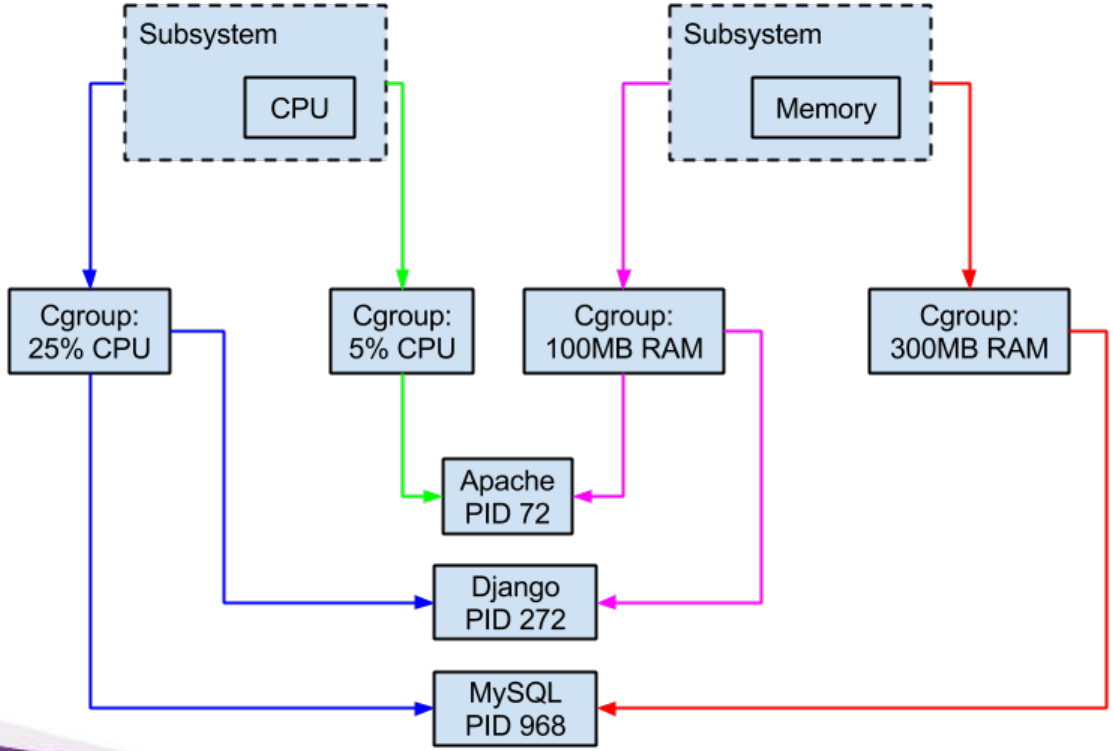
Cgroups

- Encapsulate a set of tasks
- Encapsulate a set of parameters
- The parameters confine the tasks
  - Limiting memory, cpu shares, etc
- Tasks are processes in Linux parlance

Simple Cgroup Hierarchy



Complicated Cgroup Hierarchy



TAOS

Interacting with Cgroups

- View subsystems in /sys/fs/cgroup
  - They are mounted cgroup filesystems

Interacting with Cgroups

- View subsystems in /sys/fs/cgroup
  - They are mounted cgroup filesystems
- Memory root cgroup: /sys/fs/cgroup/memory
- CPU root cgroup in: /sys/fs/cgroup/cpu

Interacting with Cgroups

- View subsystems in /sys/fs/cgroup
  - They are mounted cgroup filesystems
- Memory root cgroup: /sys/fs/cgroup/memory
- CPU root cgroup in: /sys/fs/cgroup/cpu
- They are filesystems, view subsystems with ‘mount’ command

Some Cgroups on Fedora 21

- Sample mounted cgroups (output formatted)

```
cgroup on /sys/fs/cgroup/cpuset type cgroup (rw,n
osuid,nodev,noexec,relatime,cpuset)

cgroup on /sys/fs/cgroup/cpu,cpuacct type cgroup
(rw,nosuid,nodev,noexec,relatime,cpu,cpuacct)

cgroup on /sys/fs/cgroup/memory type cgroup (rw,n
osuid,nodev,noexec,relatime,memory)
```

TAOS

Interacting with Cgroups

- Let's write a horrible python program

```
def eat_all_ram():
  x=[]
  for i in itertools.count(start=0, step=1):
    x.append(operation(i))
  return len(x)
eat_all_ram()
```

Interacting with Cgroups

- Let's write a horrible python program

```
def eat_all_ram():
  x=[]
  for i in itertools.count(start=0, step=1):
    x.append(operation(i))
  return len(x)
eat_all_ram()
```

- Oh no! A misbehaving program is eating all our memory!

TAOS

Interacting with Cgroups

- Create a cgroup under the memory subsystem and limit the PID to 10MB

Interacting with Cgroups

- Create a cgroup under the memory subsystem and limit the PID to 10MB

```
[root@fedora ~]# mkdir /sys/fs/cgroup/memory/dangroup
[root@fedora ~]# cd /sys/fs/cgroup/memory/dangroup
[root@fedora dangroup]# echo 10485760 > memory.limit_in_bytes
[root@fedora dangroup]# echo 2345 > tasks
```

Interacting with Cgroups

- Create a cgroup under the memory subsystem and limit the PID to 10MB

```
[root@fedora ~]# mkdir /sys/fs/cgroup/memory/dangroup
[root@fedora ~]# cd /sys/fs/cgroup/memory/dangroup
[root@fedora dangroup]# echo 10485760 > memory.limit_in_bytes
[root@fedora dangroup]# echo 2345 > tasks
```

- All fixed!

Interacting with Cgroups

- Create a cgroup under the memory subsystem and limit the PID to 10MB

```
[root@fedora ~]# mkdir /sys/fs/cgroup/memory/dangroup
[root@fedora ~]# cd /sys/fs/cgroup/memory/dangroup
[root@fedora dangroup]# echo 10485760 > memory.limit_in_bytes
[root@fedora dangroup]# echo 2345 > tasks
```

- All fixed!
- Except...process will swap near 10 MB.

Linux Namespaces

- PID
- NET
- MNT
- IPC
- UTS

Linux Namespaces

- PID
- NET
- MNT
- IPC
- UTS
- I'll only summarize PID and NET

PID Namespace

- Isolated process tree
  - eg, each tree may run its own init as pid 1
- PID Namespace is hierarchical, but one way
  - A parent namespace process is aware of child namespace processes
  - The converse is not true.
- With one parent and one child namespace, a child PID has two PID's.

NET Namespace

- Isolated network
- Separate route tables
- Seperate IP space
- Separate ARP tables
- Very powerful
- Also used by OpenStack

NET Namespace

- Isolated network
- Separate route tables
- Seperate IP space
- Separate ARP tables
- Very powerful
- Also used by OpenStack
- Check out ‘ip netns’

Section
LXC

- LXC

LXC

- Released 2008
- First native Linux container technology
  - Prior work required kernel patches
- Powerful but limited
- Docker is to Ubuntu what LXC is to Gentoo

LXC

- LXC is lightweight, but at a cost
- Not portable between distributions
  - Docker is
- Manual configuration of networking
  - iptables, brctl, ip netns, etc
- Manual configuration of union filesystem

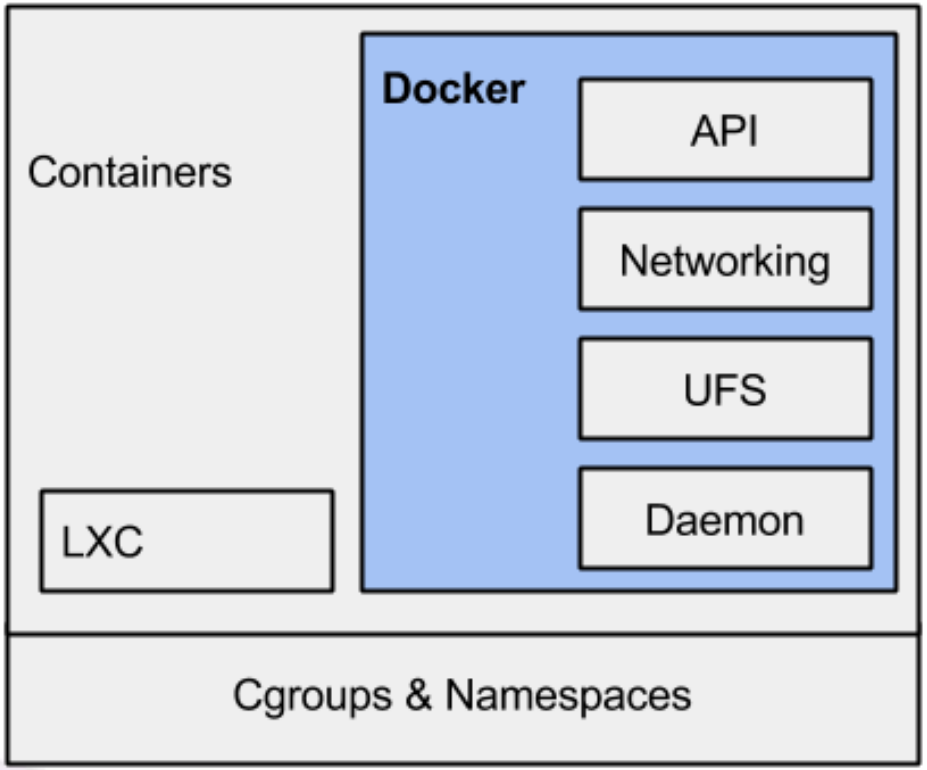
LXC versus Docker

- Both use the same cgroups, namespaces, and low level tools
- Fundamental distinction
  - Docker packages portable application centric containers
  - LXC is only a container technology

Section
Docker

- Docker

Where We Are



TAOS

What is Docker?

- A container implementation for Linux
- Released in 2013
- Current release 1.5
- Open Source, Apache License
  - github.com/docker/docker
- Owned by the Docker Corporation

Docker Implementation

- A monolithic daemon enabling operating system level virtualization for Linux.
- Relatively modern kernel needed
  - 2.6.32+
  - Centos/RHEL 6.5+
  - Ubuntu 12.04+
- Cgroups were merged in 2.6.24 (2008)

Platforms other than Linux?

- BSD
  - Not really
  - Jails are preferred
- Windows
  - With a Linux virtual machine
- Mac
  - With a Linux virtual machine

Platforms other than Linux?

- BSD
  - Not really
  - Jails are preferred
- Windows
  - With a Linux virtual machine
- Mac
  - With a Linux virtual machine
- Docker is a Linux container implementation

Terminology

- Image
- Container
- Layer
- Dockerfile

Image

- A set of UFS layers and metadata
- The “Ubuntu” image
- Images can be downloaded from Dockerhub
- Images are never “run”
  - Containers are
- Images are never “modified”
  - Modifications create new images

Container

- An instance of an image
- Containers are a running image with a topmost read-write UFS layer
- A container should serve one function
  - Pending business logic it can be complicated
  - Don’t replicate a monolithic stack for “everything”
  - One function scales better
  - One function is easier to isolate and debug

Layer

- Docker containers are built upon UFS
- UFS is the Union File System
  - Not “strictly” a union, as upper layer can overwrite lower layer.
  - Once committed, nothing in a layer is ever deleted
- Docker images limited to 127 Layers

Dockerfile

- An image build recipe
  - Procedural
  - Basic operations for executing programs and copying files.
  - Not configuration management
- Each Dockerfile directive creates a layer
  - Maximum number of layers: 127
- Examples will follow later

Networking

- Virtual bridges
- Container Linking
  - Environment variables
  - /etc/hosts
  - Retained within Linux network namespace
- Host to container port forwarding
- iptables is used for masquerading

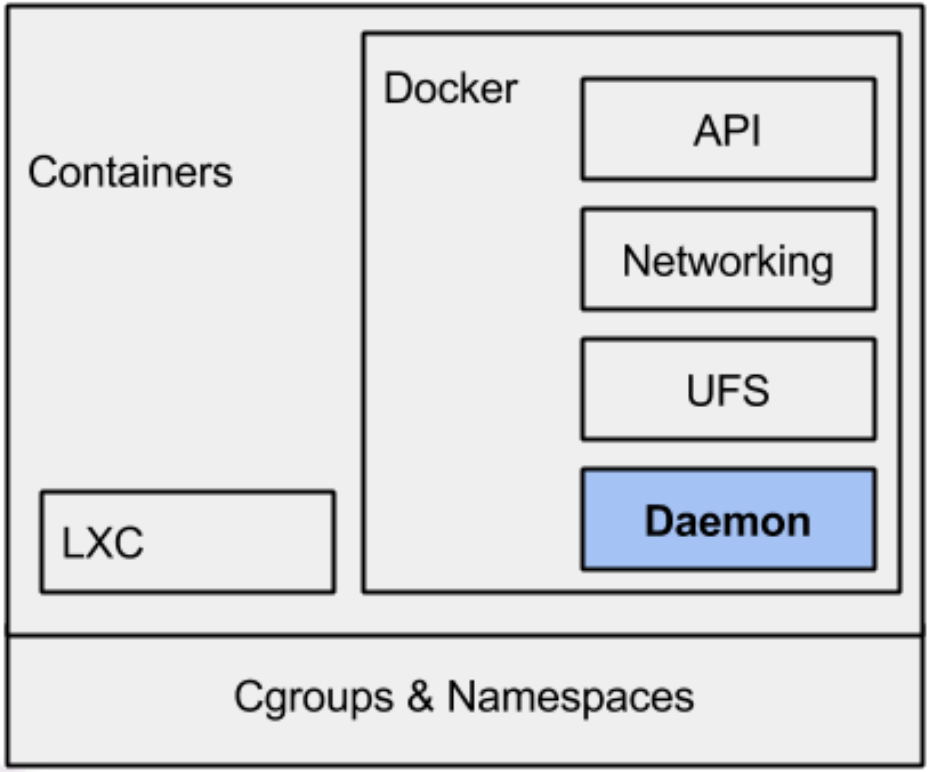
Filesystem

- Union File System
  - Base image + layers defines final image
  - Limit of 127 “layers”
  - Each docker “RUN” adds to a layer
- “Flattening” image is not natively supported
  - Achievable by exporting a container as an image
  - docker-squash utility available to help

Docker Resource Controls

- No Disk I/O Throttling
- No Disk quota
- Filesystem isolation
- CPU quotas
- RAM limits
- Network isolation but no limits

Where We Are


Containers
Docker
API
Networking
UFS
Daemon
LXC
Cgroups & Namespaces

Docker Daemon

- Client-server Architecture
  - Monolithic design
- Daemon manages containers
- Has a REST API via HTTP and Socket
  - /var/run/docker.sock
  - HTTP will be covered later by example
- Note: API uses an Open Schema.
  - Incorrect entries are ignored.

Basic Operation

- Use the ‘docker’ command.
- This is not a guide about Docker CLI
  - Examples are for illustration only
- Download an image

```
docker pull fedora
```

- Run an image as a container

Basic Operation

- List all images

```
[root@fedora ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
dsklopp/multilayer  latest              fa1864ec747a        About an hour ago   229.2 MB
ubuntu              14.04               d0955f21bf24        9 days ago          188.3 MB
ubuntu              14.04.2             d0955f21bf24        9 days ago          188.3 MB
ubuntu              latest              d0955f21bf24        9 days ago          188.3 MB
ubuntu              trusty              d0955f21bf24        9 days ago          188.3 MB
ubuntu              trusty-20150320     d0955f21bf24        9 days ago          188.3 MB
<none>              <none>              00a0c78eeb6d        4 months ago        0 B
[root@fedora ~]#
```

TAOS

Basic Operation

- Run an image interactively

```
docker run -t -i ubuntu:14.04 /bin/bash
```

Basic Operation

- Run an image interactively

```
docker run -t -i ubuntu:14.04 /bin/bash
```

- Start a container

```
docker start c6fda327bc33
```

Basic Operation

- Run an image interactively

```
docker run -t -i ubuntu:14.04 /bin/bash
```

- Start a container

```
docker start c6fda327bc33
```

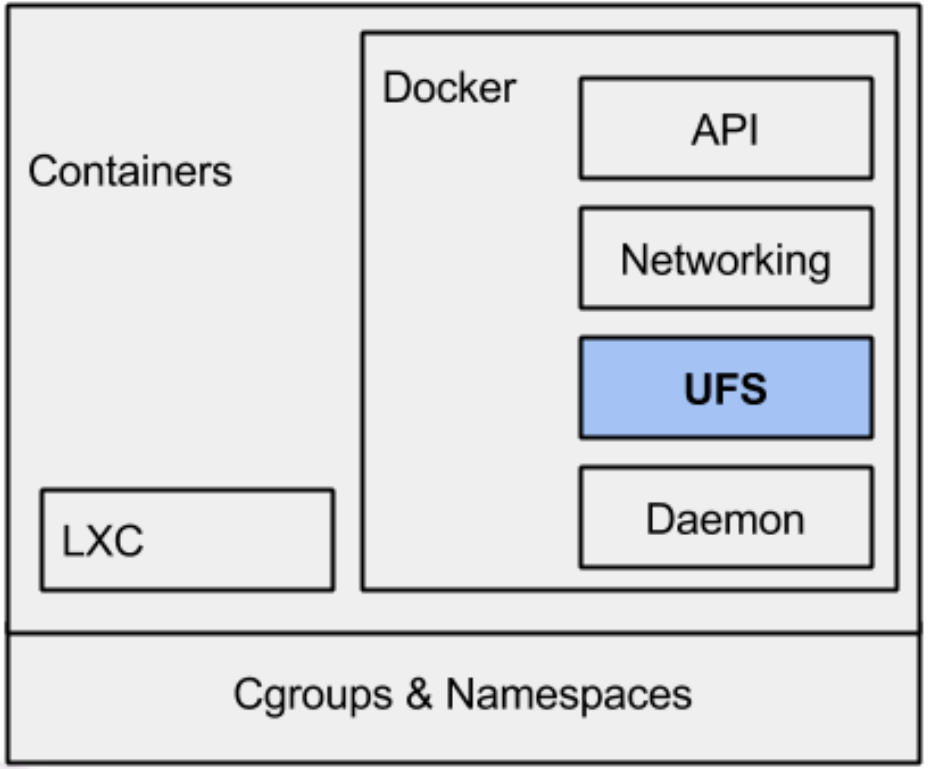
- Stop a container

```
docker stop c6fda327bc33
```

Section
Docker Filesystem

- Docker Filesystem
- Dockerfiles
- Squashing Docker Containers

Where We Are



TAOS

Union File System

- Docker utilizes Union File Systems
  - AUFS
  - BTRFS
  - VFS
  - DeviceMapper

Union File System

- Docker utilizes Union File Systems
  - AUFS
  - BTRFS
  - VFS
  - DeviceMapper
- Implementation depends on the platform
  - Fedora 21 uses DeviceMapper
  - Ubuntu 14.04 uses AUFS

Dockerfiles

- Docker UFS is best explained by Dockerfiles
- Dockerfile runs commands on an image
- Each directive creates a new layer
- The end result is an image
  - Send it to Dockerhub!
  - Share to the world!
  - Deploy it in your infrastructure

Dockerfile Structure

- Structured text file called ‘Dockerfile’

Dockerfile Structure

- Structured text file called ‘Dockerfile’
- Sample

```
FROM ubuntu:14.04
MAINTAINER Daniel Klopp

RUN touch $HOME/shell_run_variant
RUN [ "/bin/bash", "-c", "touch $HOME/exec_run_variant" ]
RUN [ "/bin/mkdir", "-p", "$HOME/exec_run_variant_mkdir" ]
RUN [ "/usr/bin/touch", "$HOME/exec_run_variant_noshell" ]
```

Dockerfile RUN

- Creates file /root/shell_run_variant

```
RUN touch $HOME/shell_run_variant
```

Dockerfile RUN

- Creates file /root/shell_run_variant

```
RUN touch $HOME/shell_run_variant
```

- Creates /root/exec_run_variant

```
RUN [ "/bin/bash", "-c", "touch $HOME/exec_run_variant" ]
```

Dockerfile RUN

- Creates file /root/shell_run_variant

```
RUN touch $HOME/shell_run_variant
```

- Creates /root/exec_run_variant

```
RUN [ "/bin/bash", "-c", "touch $HOME/exec_run_variant" ]
```

- ‘$HOME/exec_run_variant_mkdir’, literally.

```
RUN [ "/bin/mkdir", "-p", "$HOME/exec_run_variant_mkdir" ]
```

Dockerfile build

- Build and cropped output

```
docker build --no-cache -t dsklopp/run_variants .
```

Dockerfile build

- Build and cropped output

```
docker build --no-cache -t dsklopp/run_variants .
```

```
Step 0 : FROM ubuntu:14.04
 ---> d0955f21bf24
Step 1 : MAINTAINER Daniel Klopp
 ---> Running in 1f4b4a83bd1e
 ---> aa57bc7e6633
Removing intermediate container 1f4b4a83bd1e
Step 2 : RUN touch $HOME/shell_run_variant
 ---> Running in 24e2d896c2e5
 ---> 981adeaf7d60
```

The Layers of a Docker Build

- Five directives, five new layers

```
ubuntu:trusty-20150320
├─aa57bc7e6633 Virtual Size: 188.3 MB
│ └─981adeaf7d60 Virtual Size: 188.3 MB
│   └─48a88966b2b8 Virtual Size: 188.3 MB
│     └─dabec6013060 Virtual Size: 188.3 MB
│       └─8570adf87230 Virtual Size: 188.3 MB Tags: dsklopp/run_variants
```

- One layer for MAINTAINER
- Four layers for four RUN directives

Honorable Directive Mentions

- ENV, set environment variables for container
- These work in Dockerfiles, sample:

```
ENV myName="Dan Klopp" myDistro="Ubuntu"
RUN echo $myName > $HOME/name.txt
RUN echo $myDistro >> $HOME/name.txt
```

- The variables are accessible from Dockerfile and the container itself.

Honorable Directive Mentions

- Inside the container:

```
root@8ea2a330dcf1:/# env |grep -i my
myDistro=Ubuntu
myName=Dan Klopp
root@8ea2a330dcf1:/# cat $HOME/name.txt
Dan Klopp
Ubuntu
root@8ea2a330dcf1:/#
```

Honorable Directive Mentions

- Environment variables can be set on launch
- Pass to docker run command:

```
--env newKey="newValue"
```

Honorable Directive Mentions

- Environment variables can be set on launch
- Pass to docker run command:

```
--env newKey="newValue"
```

- A handy debug option!

More Directive Details

- See the official documentation

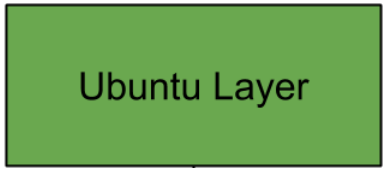
  < https://docs.docker.com/reference/builder/ >

Layers, Size and Building

- Each Layer adds to the size

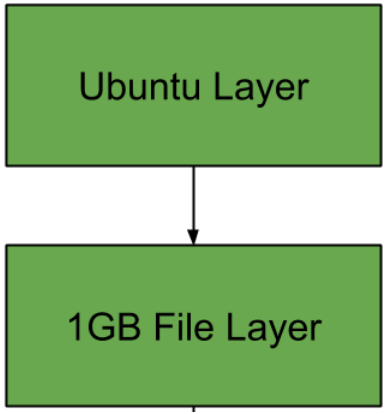
Layers, Size and Building

- Each Layer adds to the size
  - Layer 1 has Ubuntu, ~188 MB



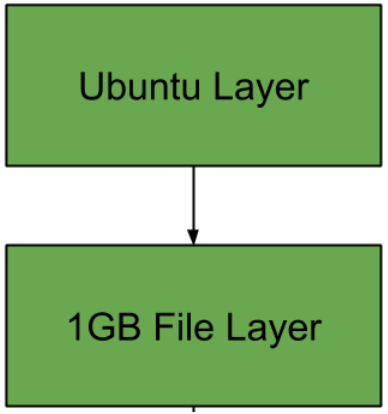
Layers, Size and Building

- Each Layer adds to the size
  - Layer 1 has Ubuntu, ~188 MB
  - Layer 2 adds a 1 GB file



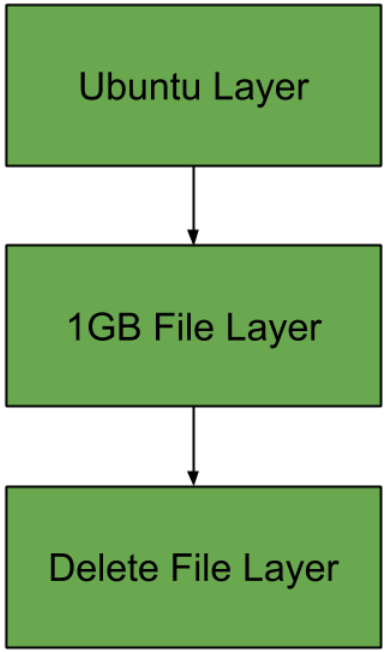
Layers, Size and Building

- Each Layer adds to the size
  - Layer 1 has Ubuntu, ~188 MB
  - Layer 2 adds a 1 GB file
  - Total size ~1.2 GB



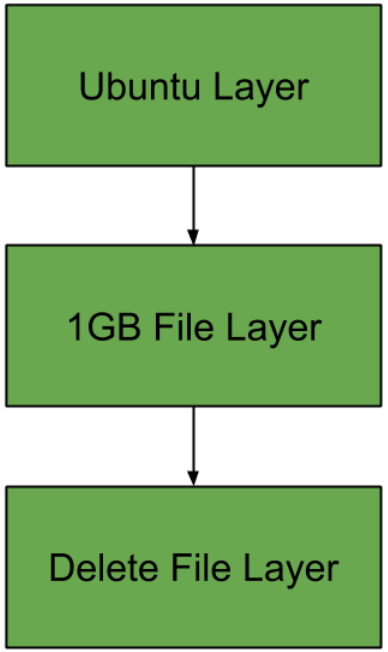
Layers, Size and Building

- Each Layer adds to the size
  - Layer 1 has Ubuntu, ~188 MB
  - Layer 2 adds a 1 GB file
  - Total size ~1.2 GB
- Delete the 1 GB file in Layer 3



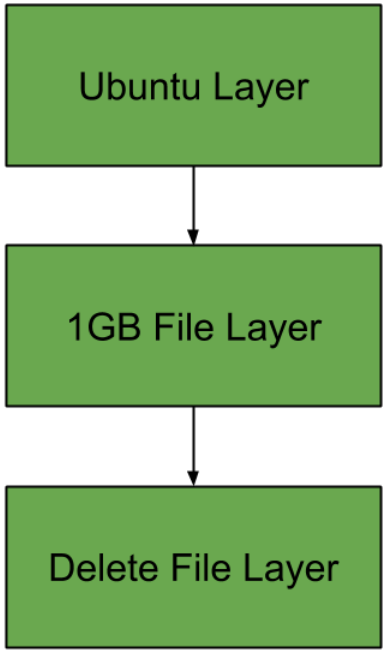
Layers, Size and Building

- Each Layer adds to the size
  - Layer 1 has Ubuntu, ~188 MB
  - Layer 2 adds a 1 GB file
  - Total size ~1.2 GB
- Delete the 1 GB file in Layer 3
  - Layer 3 deletes the 1 GB file



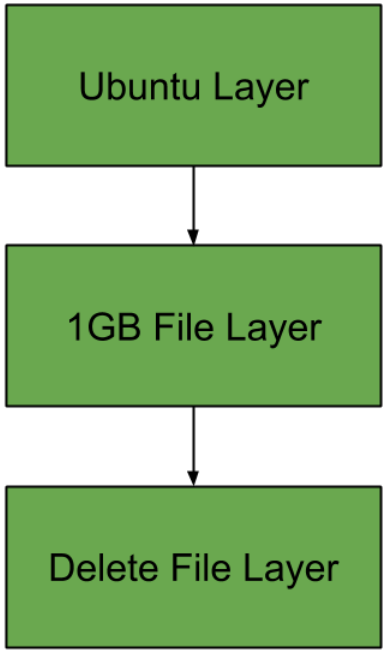
Layers, Size and Building

- Each Layer adds to the size
  - Layer 1 has Ubuntu, ~188 MB
  - Layer 2 adds a 1 GB file
  - Total size ~1.2 GB
- Delete the 1 GB file in Layer 3
  - Layer 3 deletes the 1 GB file
  - Total size ~1.2 GB



Layers, Size and Building

- Each Layer adds to the size
  - Layer 1 has Ubuntu, ~188 MB
  - Layer 2 adds a 1 GB file
  - Total size ~1.2 GB
- Delete the 1 GB file in Layer 3
  - Layer 3 deletes the 1 GB file
  - Total size ~1.2 GB
- This can be a problem



Layers, Size and Building

- Each ‘directive’ adds a layer
- Run all commands in one directive, contrast:

```
RUN dd if=/dev/zero of=/bigfile.zeroes \
 bs=1k count=$((1024*1024))
RUN rm -f /bigfile.zeroes
```

- With

```
RUN dd if=/dev/zero of=/bigfile.zeroes \
 bs=1k count=$((1024*1024)) \
 && rm -f /bigfile.zeroes
```

TAOS

Layers, Size and Building

- This makes a significant difference in size

```
dsklopp/optimized     ...   188.3  MB
dsklopp/unoptimized   ...   1.262  GB
```

Dockerfiles

- Are not configuration management
- Package the results of a build process
- Each layer should be functional
  - OS layer
  - Patch to latest
  - Webserver
  - Webserver content
- A build process makes this less relevant

How To Compress Layers

- A container is a collection of layers
- In Ubuntu, this is implemented with aufs
- /var/lib/docker stores the FS and metadata.
- Edit the half dozen json files and aufs entries
- ...
- Maybe that isn't such a good idea
  - If only there was a program to do it for us
  - Jason Wilder's "docker-squash"

Docker-squash

- https://github.com/jwilder/docker-squash
- Trivial to use. Sample Dockerfile:

```
RUN dd if=/dev/zero of=/bigfile.zeroes \
 bs=1k count=$((1024*128))
RUN rm -f /bigfile.zeroes
RUN touch /testfile.txt
```

- Size

```
dsklopp/nonflat   ...   322.5  MB
```

TAOS

Docker-squash

- Present layer hierarchy

```
ubuntu:trusty, ubuntu:trusty-20150320
    ├─8406a4346504 Virtual Size: 188.3 MB
    │ ├─76bf86be913c Virtual Size: 322.5 MB
    │ │ └─1f79b453eaf8 Virtual Size: 322.5 MB
    │ │   └─d9b59ab2e0ad Virtual Size: 322.5 MB Tags: dsklopp/nonflat
```

- Compress it with docker-squash:

```
docker save d9b59ab2e0ad |\
./docker-squash --from 8406a4346504 -t dsklopp/flat |\
docker load
```

TAOS

Docker-squash

- The new hierarchy:

```
ubuntu:trusty, ubuntu:trusty-20150320
    ├─8406a4346504 Virtual Size: 188.3 MB
    │ ├─5aa152140bf7 Virtual Size: 188.3 MB Tags: dsklopp/flat:latest
    │ ├─76bf86be913c Virtual Size: 322.5 MB
    │ │ └─1f79b453eaf8 Virtual Size: 322.5 MB
    │ │   └─d9b59ab2e0ad Virtual Size: 322.5 MB Tags: dsklopp/nonflat
```

- Sizing:

```
dsklopp/flat      ...   188.3  MB
dsklopp/nonflat   ...   322.5  MB
```

TAOS

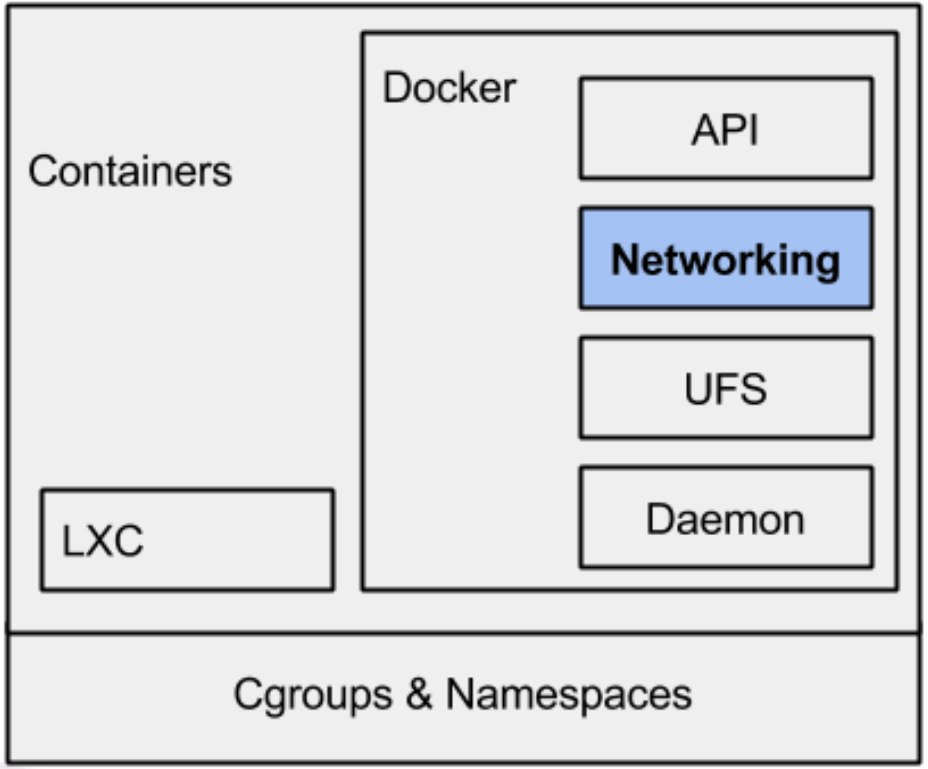
Docker-squash

- One downside is you lose layer history
- A problem if this is significant
- If the build process is automated, it rarely is.

Section
Docker Networking

- Docker Networking

Where We Are



TAOS

This Will Be Brief

- Networking deserves its own presentation to do it justice
- Docker networking is also non-trivial
- I will only go over the basics today

Default Networking

- Docker utilizes a virtual ethernet bridge
  - By default called ‘docker0’
- Containers connect via a veth interface
- Containers on docker0 can freely communicate with each other
  - Controlled by start flag ‘icc=true’, the default

Networking

- Four modes
  - None (self-explanatory)
  - Bridged (default)
  - Host
  - Shared

Bridged Networking

- Virtual Linux Ethernet bridge
- Docker default
- Dynamically or manually configured
- With icc=true (default), all containers can contact other containers.

Host Networking

- Docker containers use host network stack
- Though conceptually easy, not good practice
- Allows access to D-Bus
- Unexpected behavior may result
- Generally a bad idea

Shared Networking

- Existing container network stack is shared
- Each container PID and FS are distinct
- Only network namespace is shared.

Default Docker Networking

- Containers can reach out
- Outside cannot reach in without explicit port mappings
- Containers can communicate with other containers

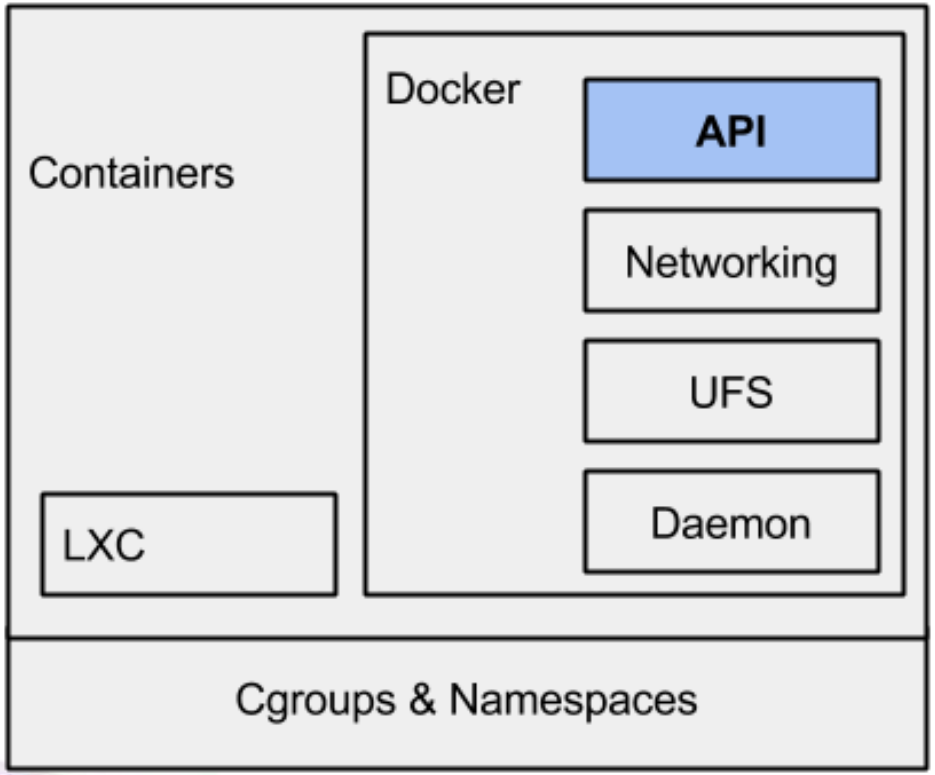
Default Docker Networking

- Ultimately, it is made up of Iptables, network namespaces, and Linux bridges (virtual switches).
  - Your limits are your imagination
- If there is interest, I can create an in depth networking presentation

Section
Docker API

- Docker API

Where We Are



TAOS

Docker API

- Has a (non) REST API via Unix Socket
  - /var/run/docker.sock
- Note: API uses an Open Schema.
  - Incorrect entries are ignored.
- There is an HTTP endpoint
  - disabled by default

Docker API via HTTP

- Disabled by default
- Pass “-H $host:$port” to daemon
  - Ubuntu: /etc/default/docker
  - Fedora: /etc/sysconfig/docker

Docker API via HTTP

- Disabled by default
- Pass “-H $host:$port” to daemon
  - Ubuntu: /etc/default/docker
  - Fedora: /etc/sysconfig/docker
- It is REST-like, but not REST
  - It does not allow querying for URI’s
  - Certain operations are hijacked for STDOUT/ERR

Docker API via HTTP

- Disabled by default
- Pass “-H $host:$port” to daemon
  - Ubuntu: /etc/default/docker
  - Fedora: /etc/sysconfig/docker
- It is REST-like, but not REST
  - It does not allow querying for URI’s
  - Certain operations are hijacked for STDOUT/ERR
- https://docs.docker.com/reference/api/docker_remote_api

TAOS

Docker API Query

```
curl -sX GET 127.0.0.1:2375/info | python -m json.tool
```

Docker API Query

```
curl -sX GET 127.0.0.1:2375/info | python -m json.tool
```


```
{
    "Containers": 5,
    "Debug": 0,
    "DockerRootDir": "/var/lib/docker",
    "Driver": "devicemapper",
    "DriverStatus": [
        [
            "Pool Name",
            "docker-253:1-263795-pool"
        ],
```


TAOS

Docker API

- Queries are nice, but wouldn't it be awesome if we could create containers via the API?

Docker API

- Queries are nice, but wouldn't it be awesome if we could create containers via the API?
  - We can

Docker API

- Queries are nice, but wouldn't it be awesome if we could create containers via the API?
  - We can
- POST /containers/create

Docker API

- POST /containers/create
- That’s easy!

Docker API

- POST /containers/create
- That’s easy!
- Pet peeve: online documentation for generating proper POST requests is scant

Docker API Launch Container

- The Docker POST requires a configuration.
- Get one from a running container
- Start an interactive container

```
docker run -t -i ubuntu:14.04 /bin/bash
```

TAOS

Docker API Launch Container

- Get the container's ID

```
curl -sX GET 127.0.0.1:2375/containers/json
```

TAOS

Docker API Launch Container

- Get the container's ID

```
curl -sX GET 127.0.0.1:2375/containers/json
```


```
[
    {
        "Command": "/bin/bash",
        "Created": 1427597350,
        "Id": "027912d4100bebc3a43bdd1c466df2142ac0cda2bddc876
        "Image": "ubuntu:14.04",
        "Names": [
            "/angry_pike"
        ],
        "Ports": [],
        "Status": "Up 32 seconds"
    }
]
```


TAOS

Docker API Launch Container

- Get the container's Configuration

```
curl -X GET 127.0.0.1:2375/containers/$ID/json
```

- Take the config output (cropped below)

```
"Config": {
    "AttachStderr": true,
    "AttachStdin": true,
    "AttachStdout": true,
    "Cmd": [
        "/bin/bash"
    ],
```

- Docker inspect will also display this information.

TAOS

Docker API Launch Container

- Modify the config to your liking, and send it as the POST body. For example

```
curl -sX POST -H "Content-Type: application/json" \
http://127.0.0.1:2375/containers/create -d '{ $CONTENT_HERE }'
```

- Where $CONTENT_HERE is the POST body you copied from before.
- Cropped output:

```
{"Id":"ae821e145e6f1e9beb4848722d2fde82d33535756c2a9625430bb1d88
```

TAOS

Docker API Launch Container

- The container isn't started yet.
- Start it with the ID returned from the POST

```
curl -sX POST -H 'Content-Type: application/json'
127.0.0.1:2375/containers/$IDNEW/start
```

- Check on your container

```
docker ps
```

```
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
ae821e145e6f        ubuntu:14.04        "/bin/bash"         4 minutes ago       Up 2 sec
```

TAOS

Docker API

- Start / stop containers
- Import / export images
- Look at running processes
- Attach to a running container
- Extremely powerful
  - Be wary of access, it is inherently insecure

Section
Docker Management

- Docker Management

Docker Management

- In short, there aren’t.

Docker Management

- In short, there aren't.
- Ok, that was harsh.
  - This is a very new field.
  - There are a lot of newfound projects for this gap

Docker Management

- A major gap
- New field, largely untested, mostly custom
- Noteworthy Solutions
  - Swarm (official)
  - Machine
  - Compose
  - Mesos

Docker Management

- Config Management is entering the race too!
  - Chef
  - Puppet
  - Ansible

Summary

- Docker is built upon Linux Containers
  - Cgroups
  - Namespaces
  - Union File Systems
- Docker provides single-host management functionality of containers
- Containers on one system will work on another.

What We Covered

- Why containers are important
- Kernel features enabling containerization
  - cgroups
  - namespaces
- LXC and its complexities

What We Covered

- Docker Implementation
  - General operation
  - Filesystem
  - Networking
  - API
- Docker Management, or lack thereof

Summary

- We talked about the fishing rod
  - Now go fish!

Special Thanks

- Taos
- SVLUG
- GE
- Especially
  - Rachel Pecchenino
  - Kevin Dankwardt

Questions?