Best practices in development and deployment, with Docker and Containers

February 2014—Docker 0.8.1
@jpetazzo

• Wrote dotCloud PAAS deployment tools
  – EC2, LXC, Puppet, Python, Shell, ØMQ...
• Docker contributor
  – Docker-in-Docker, VPN-in-Docker, router-in-Docker...
    CONTAINERIZE ALL THE THINGS!
• Runs Docker in production
  – You shouldn't do it, but here's how anyway!
Outline

- Why should I care?
- The container metaphor
- Very quick demo
- Working with Docker
- Building images
- Docker future
Outline

- Why should I care?
- The container metaphor
- Very quick demo
- Working with Docker
- Building images
- Docker future
Deploy everything

- webapps
- backends
- SQL, NoSQL
- big data
- message queues
- … and more
Deploy almost everywhere
Deploy *almost* everywhere

YUP
Deploy almost everywhere
Deploy almost everywhere

YUP  SOON  SOON
Deploy almost everywhere

YUP

SOON

SOON
Deploy almost everywhere

YUP	SOON	SOON	CLI
Deploy *almost* everywhere

YUP    SOON    SOON    CLI
Deploy almost everywhere

YUP  SOON  SOON  CLI

Yeah, right...
Deploy almost everywhere

YUP       SOON       SOON       CLI

SCALE

docker
Deploy almost everywhere

- Linux servers
- VMs or bare metal
- Any distro
- Kernel 3.8 (or RHEL 2.6.32)
Deploy reliably & consistently
WORKED FINE IN DEV

OPS PROBLEM NOW
Deploy reliably & consistently

- If it works locally, it will work on the server
- *With exactly the same behavior*
- Regardless of versions
- Regardless of distros
- Regardless of dependencies
Deploy efficiently

- Containers are lightweight
  - Typical laptop runs 10-100 containers easily
  - Typical server can run 100-1000 containers

- Containers can run at native speeds
  - Lies, damn lies, and other benchmarks:
    http://qiita.com/syoyo/items/bea48de8d7c6d8c73435
The performance!
It's over 9000!

**InfiniBand bandwidth performance**

- **ib_read_bw**
  - Native
  - LXC (Docker container)

- **ib_write_bw**
  - Native
  - LXC (Docker container)

**InfiniBand latency performance**

- **ib_read_lat**
  - Native
  - LXC (Docker container)

- **ib_write_lat**
  - Native
  - LXC (Docker container)
Outline

- Why should I care?
- **The container metaphor**
- Very quick demo
- Working with Docker
- Building images
- Docker future
... Container ?
High level approach: it's a lightweight VM

- own process space
- own network interface
- can run stuff as root
- can have its own /sbin/init (different from the host)

« Machine Container »
Low level approach: it's chroot on steroids

- can also *not* have its own /sbin/init
- container = isolated process(es)
- share kernel with host
- no device emulation (neither HVM nor PV)

« Application Container »
How does it work?
Isolation with namespaces

- pid
- mnt
- net
- uts
- ipc
- user
pid namespace

jpetazzo@tarrasque:~$ ps aux | wc -l
212

jpetazzo@tarrasque:~$ sudo docker run -t -i ubuntu bash
root@ea319b8ac416:/# ps aux
USER    PID %CPU %MEM   VSZ   RSS TTY STAT START  TIME COMMAND
root      1  0.0  0.0  18044  1956 ?     S    02:54  0:00 bash
root     16  0.0  0.0  15276  1136 ?     R+   02:55  0:00 ps aux

(That's 2 processes)
mnt namespace

jpetazzo@tarrasque:~$ wc -l /proc/mounts
32 /proc/mounts

root@ea319b8ac416:/# wc -l /proc/mounts
10 /proc/mounts
net namespace

root@ea319b8ac416:/# ip addr

1: lo: <LOOPBACK,UP,LOWER_UP>  mtu 65536 qdisc noqueue state UNKNOWN
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00  inet 127.0.0.1/8  scope host lo
      valid_lft forever preferred_lft forever
   inet6 ::1/128  scope host
      valid_lft forever preferred_lft forever

22: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP>  mtu 1500 qdisc pfifo_fast state UP qlen 1000
   link/ether 2a:d1:4b:7e:bf:b5 brd ff:ff:ff:ff:ff:ff  inet 10.1.1.3/24  brd 10.1.1.255 scope global eth0
      valid_lft forever preferred_lft forever
   inet6 fe80::28d1:4bff:fe7e:bfb5/64  scope link
      valid_lft forever preferred_lft forever
uts namespace

jpetazzo@tarrasque:~$ hostname
tarrasque

root@ea319b8ac416:/# hostname
ea319b8ac416
### ipc namespace

```sh
jpetazzo@tarrasque:~$ ipcs
------- Shared Memory Segments -------
key    shmid   owner   perms  bytes  nattch  status
0x00000000 3178496  jpetazzo  600  393216  2  dest
0x00000000 557057   jpetazzo  777  2778672 0
0x00000000 3211266  jpetazzo  600  393216  2  dest
```

```sh
root@ea319b8ac416:/# ipcs
------- Shared Memory Segments -------
key    shmid   owner   perms  bytes  nattch  status
------- Semaphore Arrays -------
key    semid   owner   perms  nsems
------- Message Queues -------
key    msqid   owner   perms  used-bytes  messages
```
user namespace

• No demo, but see LXC 1.0 (just released)

• UID 0→1999 in container C1 is mapped to UID 10000→11999 in host; UID 0→1999 in container C2 is mapped to UID 12000→13999 in host; etc.

• what will happen with copy-on-write?
  – double translation at VFS?
  – single root UID on read-only FS?
How does it work?
Isolation with cgroups

- memory
- cpu
- blkio
- devices
memory cgroup

• keeps track pages used by each group:
  – file (read/write/mmap from block devices; swap)
  – anonymous (stack, heap, anonymous mmap)
  – active (recently accessed)
  – inactive (candidate for eviction)
• each page is « charged » to a group
• pages can be shared (e.g. if you use any COW FS)
• Individual (per-cgroup) limits and out-of-memory killer
cpu and cpuset cgroups

- keep track of user/system CPU time
- set relative weight per group
- pin groups to specific CPU(s)
  - Can be used to « reserve » CPUs for some apps
  - This is also relevant for big NUMA systems
blkio cgroups

- keep track IOs for each block device
  - read vs write; sync vs async
- set relative weights
- set throttle (limits) for each block device
  - read vs write; bytes/sec vs operations/sec

**Note**: earlier versions (<3.8) didn't account async correctly. 3.8 is better, but use 3.10 for best results.
devices cgroups

- controls read/write/mknod permissions
- typically:
  - allow: /dev/{tty,zero,random,null}...
  - deny: everything else
  - maybe: /dev/net/tun, /dev/fuse, /dev/kvm, /dev/dri...
- fine-grained control for GPU, virtualization, etc.
How does it work?
Copy-on-write storage

- Create a new machine instantly (Instead of copying its whole filesystem)
- Storage keeps track of what has changed
- Since 0.7, Docker has a storage plugin system
# Storage: many options!

<table>
<thead>
<tr>
<th></th>
<th>Union Filesystems</th>
<th>Snapshotting Filesystems</th>
<th>Copy-on-write block devices</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Provisioning</strong></td>
<td>Superfast</td>
<td>Fast</td>
<td>Fast</td>
</tr>
<tr>
<td><strong>Changing small files</strong></td>
<td>Superfast</td>
<td>Fast</td>
<td>Fast Costly</td>
</tr>
<tr>
<td><strong>Changing large files</strong></td>
<td>Slow (first time)</td>
<td>Fast</td>
<td>Fast Cheap</td>
</tr>
<tr>
<td><strong>Diffing</strong></td>
<td>Superfast</td>
<td>Superfast</td>
<td>Slow</td>
</tr>
<tr>
<td><strong>Memory usage</strong></td>
<td>Efficient</td>
<td>Efficient</td>
<td>Inefficient (at high densities)</td>
</tr>
<tr>
<td><strong>Drawbacks</strong></td>
<td>Random quirks</td>
<td>ZFS not mainline</td>
<td>Higher disk usage</td>
</tr>
<tr>
<td></td>
<td>AUFS not mainline</td>
<td>BTRFS not as nice</td>
<td>Great performance</td>
</tr>
<tr>
<td></td>
<td>!AUFS more quirks</td>
<td></td>
<td>(except diffing)</td>
</tr>
<tr>
<td><strong>Bottom line</strong></td>
<td>Ideal for PAAS and high density things</td>
<td>This is the Future (probably)</td>
<td>Dodge Ram 3500</td>
</tr>
</tbody>
</table>
Compute efficiency: *almost* no overhead

- processes are isolated, but run straight on the host
- CPU performance = native performance
- memory performance = a few % shaved off for (optional) accounting
- network performance = small overhead; can be reduced to zero
Alright, I get this.
Containers = nimble VMs.
NO.
The container metaphor
Problem: shipping goods

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Image" /></td>
<td><img src="image2.png" alt="Image" /></td>
<td><img src="image3.png" alt="Image" /></td>
<td><img src="image4.png" alt="Image" /></td>
<td><img src="image5.png" alt="Image" /></td>
<td><img src="image6.png" alt="Image" /></td>
<td><img src="image7.png" alt="Image" /></td>
<td><img src="image8.png" alt="Image" /></td>
</tr>
<tr>
<td><img src="image9.png" alt="Image" /></td>
<td><img src="image10.png" alt="Image" /></td>
<td><img src="image11.png" alt="Image" /></td>
<td><img src="image12.png" alt="Image" /></td>
<td><img src="image13.png" alt="Image" /></td>
<td><img src="image14.png" alt="Image" /></td>
<td><img src="image15.png" alt="Image" /></td>
<td><img src="image16.png" alt="Image" /></td>
</tr>
<tr>
<td><img src="image17.png" alt="Image" /></td>
<td><img src="image18.png" alt="Image" /></td>
<td><img src="image19.png" alt="Image" /></td>
<td><img src="image20.png" alt="Image" /></td>
<td><img src="image21.png" alt="Image" /></td>
<td><img src="image22.png" alt="Image" /></td>
<td><img src="image23.png" alt="Image" /></td>
<td><img src="image24.png" alt="Image" /></td>
</tr>
</tbody>
</table>

**SCALE 12x**
Solution: the *intermodal shipping container*
Solved!
## Problem: shipping code

<table>
<thead>
<tr>
<th>Database</th>
<th>Container</th>
<th>Network</th>
<th>API</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>Docker</td>
<td>Kubernetes</td>
<td>Docker</td>
<td>Docker</td>
</tr>
<tr>
<td>Cassandra</td>
<td>Docker</td>
<td>Kubernetes</td>
<td>Docker</td>
<td>Docker</td>
</tr>
</tbody>
</table>

### Tools
- MySQL
- Docker
- Kubernetes
Solution: the *Linux* container
Solved!
Separation of concerns: Dave the Developer

- inside my container:
  - my code
  - my libraries
  - my package manager
  - my app
  - my data
Separation of concerns: Oscar the Ops guy

- outside the container:
  - logging
  - remote access
  - network configuration
  - monitoring
Separation of concerns: what it *doesn't* mean

« I don't have to care » ≠ « I don't care »
DEVOPS

VS
Outline

• Why should I care?
• The container metaphor
• **Very quick demo**
• Working with Docker
• Building images
• Docker future
Yes, but...

- « I don't need Docker; I can do all that stuff with LXC tools, rsync, and some scripts! »

- correct on all accounts; but it's also true for apt, dpkg, rpm, yum, etc.

- the whole point is to commoditize, i.e. make it ridiculously easy to use
What this really means…

- instead of writing « very small shell scripts » to manage containers, write them to do the rest:
  - continuous deployment/integration/testing
  - orchestration
- = use Docker as a building block
- re-use other people images (yay ecosystem!)
Docker-what?
The Big Picture

- Open Source engine to commoditize LXC
- using copy-on-write for quick provisioning
- allowing to **create and share** images
- **standard format** for containers
  (stack of layers; 1 layer = tarball+metadata)
- standard, **reproducible** way to **easily** build **trusted** images (Dockerfile, Stackbrew...)
Docker-what?

History

- rewrite of dotCloud internal container engine
  - original version: Python, tied to dotCloud PaaS
  - released version: Go, legacy-free
- remember SCALE11X talk about LXC?
  - Docker was announced one month later!
Docker-what?
Under the hood

- the Docker daemon runs in the background
  - manages containers, images, and builds
  - HTTP API (over UNIX or TCP socket)
  - embedded CLI talking to the API
Docker-what?
Take me to your dealer

• Open Source
  – GitHub public repository + issue tracking
    https://github.com/dotcloud/docker

• Nothing up the sleeve
  – public mailing lists (docker-user, docker-dev)
  – IRC channels (Freenode: #docker #docker-dev)
  – public decision process
Docker-what?
The ecosystem

- Docker Inc. (formerly dotCloud Inc.)
  - ~30 employees, VC-backed
  - SAAS and support offering around Docker

- Docker, the community
  - more than 300 contributors, 1500 forks on GitHub
  - dozens of projects around/on top of Docker
  - x100k trained developers
Outline

- Why should I care?
- The container metaphor
- Very quick demo
- Working with Docker
- Building images
- Docker future
One-time setup

• On your servers (Linux)
  - Packages (Ubuntu, Debian, Fedora, Gentoo, Arch...)
  - Single binary install (Golang FTW!)
  - Easy provisioning on Rackspace, Digital Ocean, EC2, GCE...

• On your dev env (Linux, OS X, Windows)
  - Vagrantfile
  - boot2docker (25 MB VM image)
  - Natively (if you run Linux)
The Docker workflow 1/2

- Work in dev environment (local machine or container)
- Other services (databases etc.) in containers (and behave just like the real thing!)
- Whenever you want to test « for real »:
  - Build in seconds
  - Run *instantly*
The Docker workflow 2/2

Satisfied with your local build?
- Push it to a registry (public or private)
- Run it (automatically!) in CI/CD
- Run it in production
- Happiness!

Something goes wrong? Rollback painlessly!
Outline

- Why should I care?
- The container metaphor
- Very quick demo
- Working with Docker
- **Building images**
- Docker future
Authoring images with run/commit
1) docker run ubuntu bash
2) apt-get install this and that
3) docker commit <containerid> <imagename>
4) docker run <imagename> bash
5) git clone git://.../mycode
6) pip install -r requirements.txt
7) docker commit <containerid> <imagename>
8) repeat steps 4-7 as necessary
9) docker tag <imagename> <user/image>
10) docker push <user/image>
Authoring images with run/commit

- **Pros**
  - Convenient, nothing to learn
  - Can roll back/forward if needed

- **Cons**
  - Manual process
  - Iterative changes stack up
  - Full rebuilds are boring, error-prone
Authoring images with a Dockerfile
FROM ubuntu

RUN apt-get -y update
RUN apt-get install -y g++
RUN apt-get install -y erlang-dev erlang-manpages erlang-base-hipe ...
RUN apt-get install -y libmozjs185-dev libicu-dev libtool ...
RUN apt-get install -y make wget

RUN wget http://.../apache-couchdb-1.3.1.tar.gz | tar -C /tmp -zxf-
RUN cd /tmp/apache-couchdb-* && ./configure && make install

RUN printf "[httpd]\nport = 8101\nbinding_address = 0.0.0.0" >
/usr/local/etc/couchdb/local.d/docker.ini

EXPOSE 8101
CMD ["/usr/local/bin/couchdb"]

docker build -t jpetazzo/couchdb .
Authoring images with a Dockerfile

- Minimal learning curve
- Rebuilds are easy
- Caching system makes rebuilds faster
- Single file to define the whole environment!
Do you even Chef?
Puppet?
Ansible?
Salt?
Docker and Puppet
Docker and Puppet

- Get a Delorean
- Warm up flux capacitors
- Time-travel to yesterday
- Check Brandon Burton's lightning talk
- Check my talk

— Or —

- Get the slides, ask questions 😊
Outline

- Why should I care?
- The container metaphor
- Very quick demo
- Working with Docker
- Building images
- Docker future
Coming Soon

- Network acceleration
- Container-specific metrics
- Consolidated logging
- Plugins (compute backends...)
- Orchestration hooks

Those things are already possible, but will soon be part of the core.
Docker 1.0

- Multi-arch, multi-OS
- Stable control API
- Stable plugin API
- Resiliency
- Signature
- Clustering
Recap

Docker:

- Is easy to install
- Will run anything, anywhere
- Gives you repeatable builds
- Enables better CI/CD workflows
- Is backed by a strong community
- Will change how we build and ship software
Thank you! Questions?

http://docker.io/
http://docker.com/
@docker
@jpetazzo