LXC, Docker, and the future of software delivery

Linuxcon – New Orleans, 2013

Jérôme Petazzoni, dotCloud Inc.
Best practices in development and deployment, with Docker and Containers
Cluster Management with Kubernetes

Please open the gears tab below for the speaker notes

Satnam Singh satnam@google.com

Work of the Google Kubernetes team and many open source contributors
Kubernetes
An Introduction
Rishabh Indoria
The promise of cloud computing
Typically a cloud cluster node is a VM running a specific version of Linux.

User applications comprise components each of which may have different and conflicting requirements from libraries, runtimes and kernel features.

Applications are coupled to the version of the host operating system: bad.

Evolution of the application components is coupled to (and in tension with) the evolution of the host operating system: bad.

Also need to deal with node failures, spinning up and turning down replicas to deal with varying load, updating components with disruption ...

You thought you were a programmer but you are now a sys-admin.
Why Linux Containers?

What are we trying to solve?
The Matrix From Hell
Many payloads

- backend services (API)
- databases
- distributed stores
- webapps
Many payloads

- Go
- Java
- Node.js
- PHP
- Python
- Ruby
- …
Many targets

- your local development environment
- your coworkers' development environment
- your Q&A team's test environment
- some random demo/test server
- the staging server(s)
- the production server(s)
- bare metal
- virtual machines
- shared hosting
  + your dog's Raspberry Pi
Many targets

- BSD
- Linux
- OS X
- Windows
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Static website</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Web frontend</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>background workers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>User DB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Analytics DB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Queue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Matrix From Hell
Real-world analogy: containers
Many products

- clothes
- electronics
- raw materials
- wine
- …
Many transportation methods

- ships
- trains
- trucks
- ...
Another matrix from hell

|---|---|---|---|---|---|---|---|

- Barrels
- Wooden crate
- Car
- Barrel
- Grand piano
- Fish
- Train
- Container
- Crane
- Ship
- Factory
- Truck
Solution to the transport problem: the *intermodal shipping* container
## Problem: shipping code

<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Logos:
- [docker logo](#)
- [Scale logo](#)
- [Database logos](#)
- [Cloud logos](#)
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
Linux containers...

- run everywhere
  - regardless of kernel version
  - regardless of host distro
  - (but container and host architecture must match)
- run anything
  - if it can run on the host, it can run in the container
  - i.e., if it can run on a Linux kernel, it can run
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)
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)
How does it work?

Isolation with namespaces

- pid
- mnt
- net
- uts
- ipc
- user
How does it work?
Isolation with cgroups

- memory
- cpu
- blkio
- devices
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 optimized to zero overhead
Docker
Docker

Source: Google Trends
What is Docker?

An implementation of the container idea
A package format
Resource isolation
An ecosystem
What's Docker?

- Open Source engine to commoditize LXC
- using copy-on-write for quick provisioning
- allowing to create and share images
- propose a standard format for containers
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?
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
VM vs. Docker

VM1
- Application: elasticsearch (music store)
- Guest OS: Ubuntu 14.04
- Hypervisor: Xen
- Host OS: Debian Wheezy
- Server

VM2
- Application: elasticsearch (site search)
- Guest OS: Ubuntu 14.04
- Hypervisor: Xen
- Host OS: Debian Wheezy
- Server

VM3
- Application: redis
- Guest OS: Ubuntu 14.04
- Hypervisor: Xen
- Host OS: Debian Wheezy
- Server

VM4
- Application: CouchDB
- Guest OS: CentOS 6
- Hypervisor: Docker
- Host OS: Debian Wheezy
- Server

VM5
- Application: MySQL
- Guest OS: CentOS 6
- Hypervisor: Docker
- Host OS: Debian Wheezy
- Server

Docker containers
- elasticsearch (music store)
- elasticsearch (site search)
- redis
- CouchDB
- MySQL
Docker

Docker containers

elasticsearch (music store)  elasticsearch (site search)  redis  CouchDB  MySQL

Oracle Java8  JDK  EPEL

supervisor, net-tools

Debian Wheezy

Server

“build once, run anywhere”
Docker Architecture

- Docker Engine
  - CLI
  - Docker Daemon
  - Docker Registry
- Docker Hub
  - Cloud service
    - Share Applications
    - Automate workflows
    - Assemble apps from components

- Docker images
- Docker containers
Docker images......

- NOT A VHD
- NOT A FILESYSTEM
- uses a Union File System
- a read-only Layer
- do not have state
- Basically a tar file
- Has a hierarchy
  - Arbitrary depth
- Fits into the Docker Registry
Docker Containers...

Units of software delivery (ship it!)

- run everywhere
  - regardless of kernel version
  - regardless of host distro
  - (but container and host architecture must match*)

- run anything
  - if it can run on the host, it can run in the container
  - i.e., if it can run on a Linux kernel, it can run

*Unless you emulate CPU with qemu and binfmt
How does Docker work?

- You can build Docker images that hold your applications.
- You can create Docker containers from those Docker images to run your applications.
- You can share those Docker images via Docker Hub or your own registry.
Docker Container Lifecycle …..

- The Life of a Container
  - Conception
    - **BUILD** an Image from a Dockerfile
  - Birth
    - **RUN** (create+start) a container
  - Reproduction
    - **COMMIT** (persist) a container to a new image
    - **RUN** a new container from an image
  - Sleep
    - **KILL** a running container
  - Wake
    - **START** a stopped container
  - Death
    - **RM** (delete) a stopped container
- Extinction
  - **RMI** a container image (delete image)
Resource isolation

Implemented by a number of Linux APIs:

• **cgroups**: Restrict resources a process can consume
  • CPU, memory, disk IO, ...

• **namespaces**: Change a process’s view of the system
  • Network interfaces, PIDs, users, mounts, ...

• **capabilities**: Limits what a user can do
  • mount, kill, chown, ...

• **chroots**: Determines what parts of the filesystem a user can see
We need more than just packing and isolation

**Scheduling**: Where should my containers run?

**Lifecycle and health**: Keep my containers running despite failures

**Discovery**: Where are my containers now?

**Monitoring**: What’s happening with my containers?

**Auth{n,z}**: Control who can do things to my containers

**Aggregates**: Compose sets of containers into jobs

**Scaling**: Making jobs bigger or smaller

...
Everything at Google runs in containers:

- Gmail, Web Search, Maps, ...
- MapReduce, MillWheel, Pregel, ...
- Colossus, BigTable, Spanner, ...
- Even **Google’s Cloud Computing product GCE itself**: VMs run in containers
Open Source Containers: Kubernetes

Greek for “Helmsman”; also the root of the word “Governor” and “cybernetic”

- Container orchestrator
- Builds on Docker containers
  - also supporting other container technologies
- Multiple cloud and bare-metal environments
- Supports existing OSS apps
  - cannot require apps becoming cloud-native
- Inspired and informed by Google’s experiences and internal systems

- **100% Open source**, written in **Go**

Let users manage **applications**, not machines
Project Stats

- Over 46,600 stars on Github
- 1800+ Contributors to K8s Core
- Most discussed Repository by a large margin
- 50,000+ users in Slack Team
Primary concepts

**Container**: A sealed application package (Docker)

**Pod**: A small group of tightly coupled Containers

**Labels**: Identifying metadata attached to objects

**Selector**: A query against labels, producing a set result

**Controller**: A reconciliation loop that drives current state towards desired state

**Service**: A set of pods that work together
Architecture
Overview
Control Plane Components

- kube-apiserver
- etcd
- kube-controller-manager
- kube-scheduler
- cloud-controller-manager
kube-apiserver

- Provides a forward facing REST interface into the kubernetes control plane and datastore.

- All clients and other applications interact with kubernetes **strictly** through the API Server.

- Acts as the gatekeeper to the cluster by handling authentication and authorization, request validation, mutation, and admission control in addition to being the front-end to the backing datastore.
etcd

- etcd acts as the cluster datastore.
- Purpose in relation to Kubernetes is to provide a strong, consistent and highly available key-value store for persisting cluster state.
- Stores objects and config information.
etcd acts as the cluster datastore.

Purpose in relation to Kubernetes is to provide a strong, consistent and highly available key-value store for persisting cluster state.

Stores objects and config information.
etcd

Uses “Raft Consensus” among a quorum of systems to create a fault-tolerant consistent “view” of the cluster.

https://raft.github.io/
kube-controller-manager

- Monitors the cluster state via the apiserver and **steers the cluster towards the desired state**.

- Node Controller: Responsible for noticing and responding when nodes go down.
- Replication Controller: Responsible for maintaining the correct number of pods for every replication controller object in the system.
- Endpoints Controller: Populates the Endpoints object (that is, joins Services & Pods).
- Service Account & Token Controllers: Create default accounts and API access tokens for new namespaces.
kube-scheduler

- Component on the master that watches newly created pods that have no node assigned, and selects a node for them to run on.
- Factors taken into account for scheduling decisions include individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference and deadlines.
cloud-controller-manager

- Node Controller: For checking the cloud provider to determine if a node has been deleted in the cloud after it stops responding
- Route Controller: For setting up routes in the underlying cloud infrastructure
- Service Controller: For creating, updating and deleting cloud provider load balancers
- Volume Controller: For creating, attaching, and mounting volumes, and interacting with the cloud provider to orchestrate volumes
Node Components
Node Components

- kubelet
- kube-proxy
- Container Runtime Engine
kubelet

- An agent that runs on each node in the cluster. It makes sure that containers are running in a pod.
- The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy.
kube-proxy

- Manages the network rules on each node.
- Performs connection forwarding or load balancing for Kubernetes cluster services.
A container runtime is a CRI (Container Runtime Interface) compatible application that executes and manages containers.

- Containerd (docker)
- Cri-o
- Rkt
- Kata (formerly clear and hyper)
- Virtlet (VM CRI compatible runtime)
Pods: Grouping containers

- Container Foo
- Container Bar

Namespaces
- Net
- IPC
- ..
Pods: Networking

Namespaces
- Net
- IPC
- ..
Pods: Volumes

Namespaces
- Net
- IPC
- ..
Pods: Labels

- Container Foo
- Container Bar

Namespaces
- Net
-IPC
Persistent Volumes

A higher-level abstraction - insulation from any one cloud environment

Admin provisions them, users claim them

Independent lifetime and fate

Can be handed-off between pods and lives until user is done with it

Dynamically “scheduled” and managed, like nodes and pods
Labels

Arbitrary metadata

Attached to any API object

Generally represent **identity**

Queryable by **selectors**
- think SQL ‘select ... where ...’

The **only** grouping mechanism

Use to determine which objects to apply an operation to
- pods under a ReplicationController
- pods in a Service
- capabilities of a node (scheduling constraints)
Selectors

App: Nifty
Phase: Dev
Role: FE

App: Nifty
Phase: Test
Role: FE

App: Nifty
Phase: Dev
Role: BE

App: Nifty
Phase: Test
Role: BE
App: Nifty
Phase: Dev
Role: FE

App: Nifty
Phase: Test
Role: FE

App: Nifty
Phase: Dev
Role: BE

App: Nifty
Phase: Test
Role: BE
Selectors

App: Nifty
Phase: Dev
Role: FE

App: Nifty
Phase: Test
Role: FE

App: Nifty
Phase: Dev
Role: BE

App: Nifty
Phase: Test
Role: BE
Pod lifecycle

Once scheduled to a node, pods do not move
- restart policy means restart in-place

Pods can be observed pending, running, succeeded, or failed
- failed is really the end - no more restarts
- no complex state machine logic

Pods are not rescheduled by the scheduler or apiserver
- even if a node dies
- controllers are responsible for this
- keeps the scheduler simple

Apps should consider these rules
- Services hide this
- Makes pod-to-pod communication more formal
Replication Controllers
Replication Controllers

A type of controller (control loop)

Ensure N copies of a pod always running
  • if too few, start new ones
  • if too many, kill some
  • group == selector

Cleanly layered on top of the core
  • all access is by public APIs

Replicated pods are fungible
  • No implied ordinality or identity

Other kinds of controllers coming
  • e.g. job controller for batch

Replication Controller
- Name = “nifty-rc”
- Selector = {“App”: “Nifty”}
- PodTemplate = { ... }
- NumReplicas = 4

API Server
Services

(name) 1.2.3.4 "name"

port(s)

backend production

backend production
**Services**

- **Client**
  - **Service**
    - **Name**: "nifty-svc"
    - **Selector**: `{"App": "Nifty"}`
    - **Port**: 9376
    - **ContainerPort**: 8080

- **iptables DNAT**
  - TCP / UDP

- **kube-proxy**
  - **10.0.0.1 : 9376**
    - Portal IP is assigned

- **apiserver**
  - **10.240.2.2 : 8080**
  - **10.240.1.1 : 8080**
  - **10.240.3.3 : 8080**

- **watch**
A fresh Kubernetes cluster