

# rkt and Kubernetes

What's new (and coming) with  
Container Runtimes and Orchestration

---



**OSDC.de**  
OPEN SOURCE DATA  
CENTER CONFERENCE

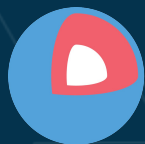
APRIL 26 - 28, 2016 | BERLIN



# Jonathan Boulle

[github.com/jonboulle](https://github.com/jonboulle) - [@baronboulle](https://twitter.com/baronboulle)

---



Core OS

# Why rkt and Kubernetes?

~~Why rkt and Kubernetes?~~  
**Why container runtimes  
and orchestration?**

# CoreOS, Inc (2013 - today)

Mission: "Secure the Internet"

## **Started at the OS level: CoreOS Linux**

- Modern, minimal operating system
- Self-updating (read-only) image
- Updates must be *automatic* and *seamless*

# ***Automatic and seamless***

- If the OS is always updating, what about applications running on it?

# *Automatic and seamless*

- If the OS is always updating, what about applications running on it?
- Classic use case for *containers* and *orchestration*
  - *containers* decouple the application and OS update lifecycles (update at different cadences)
  - *orchestration* decouples application and OS uptime (services can remain unaffected during OS downtime)

# Why container runtimes?

- So we can update the OS without affecting application dependencies



kernel  
systemd  
rkt  
ssh  
docker

python  
java  
nginx  
mysql  
openssl

stro distro distro distro distro

app

kernel  
systemd  
rkt  
ssh  
docker

stro distro distro distro distro

python  
java  
nginx  
mysql  
openssl

app

kernel  
systemd  
rkt  
ssh  
docker

stro distro distro distro distro

python  
openssl-A

app1

java  
openssl-B

app2

java  
openssl-B

app3

# CoreOS

stro distro distro distro distro

container

container

container

# CoreOS

stro distro distro distro distro


rkt container

Docker container

nspawn container

# Why orchestration?

- So we can update the OS without affecting application uptime



The diagram shows three servers, labeled server1, server2, and server3, arranged horizontally. Each server is represented by a rounded rectangle with a white border. Inside each rectangle, a list of applications is shown in green text, and the server name is shown in white text at the bottom. Server1 contains app1, app2, and app3. Server2 contains app4 and app5. Server3 contains app6 and app7.

app1  
app2  
app3

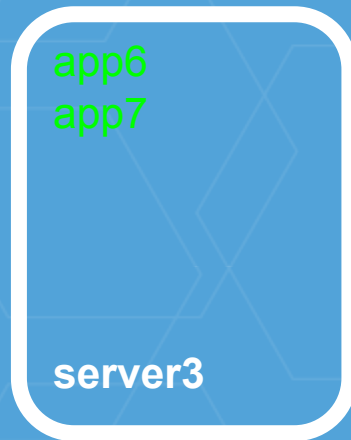
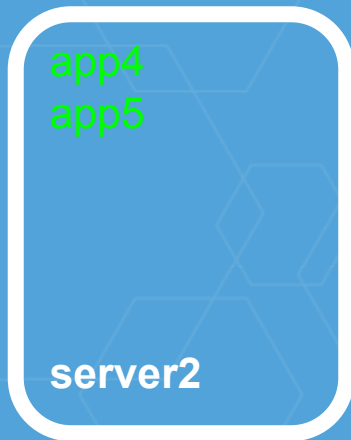
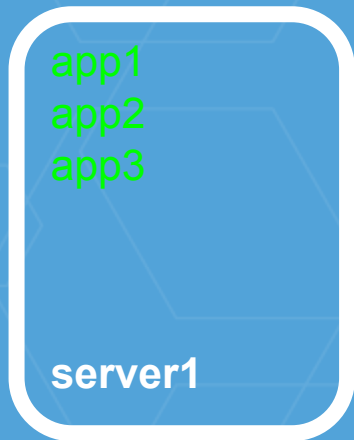
server1

app4  
app5

server2

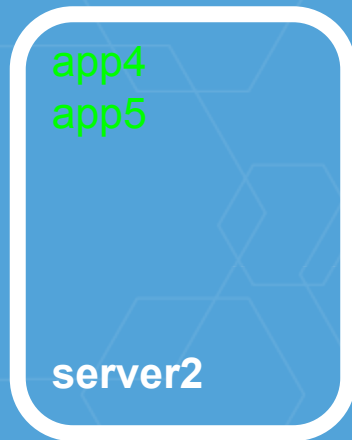
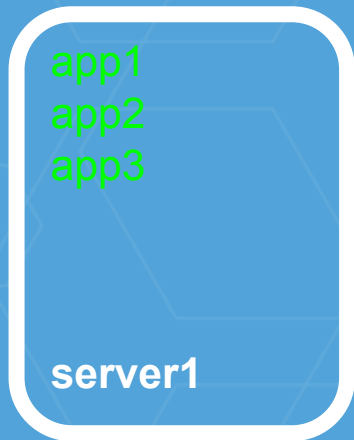
app6  
app7

server3



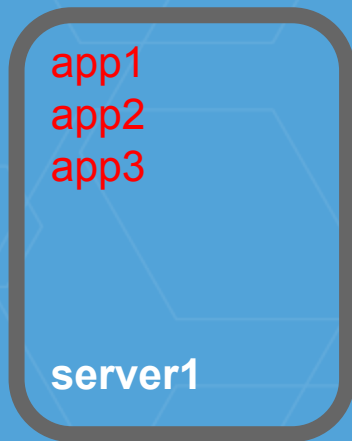
updating...



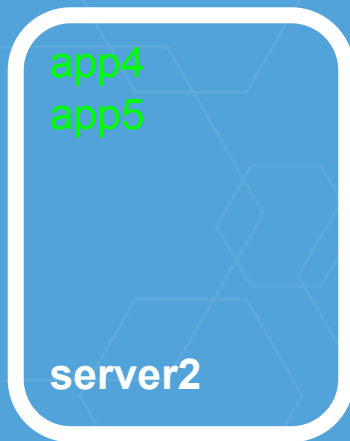


**needs reboot**

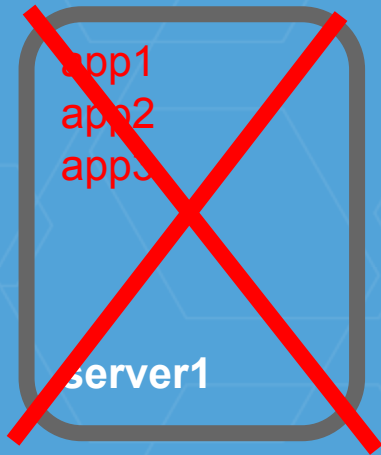
# Without orchestration



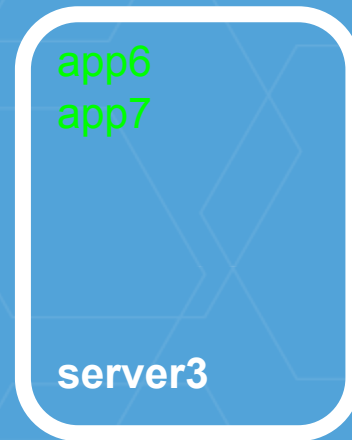
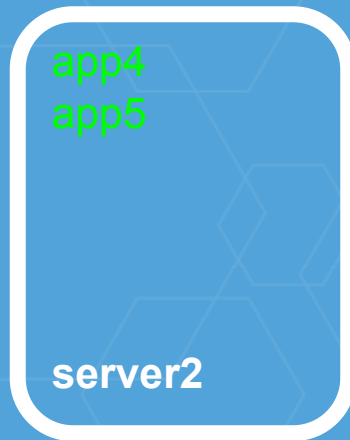
rebooting...



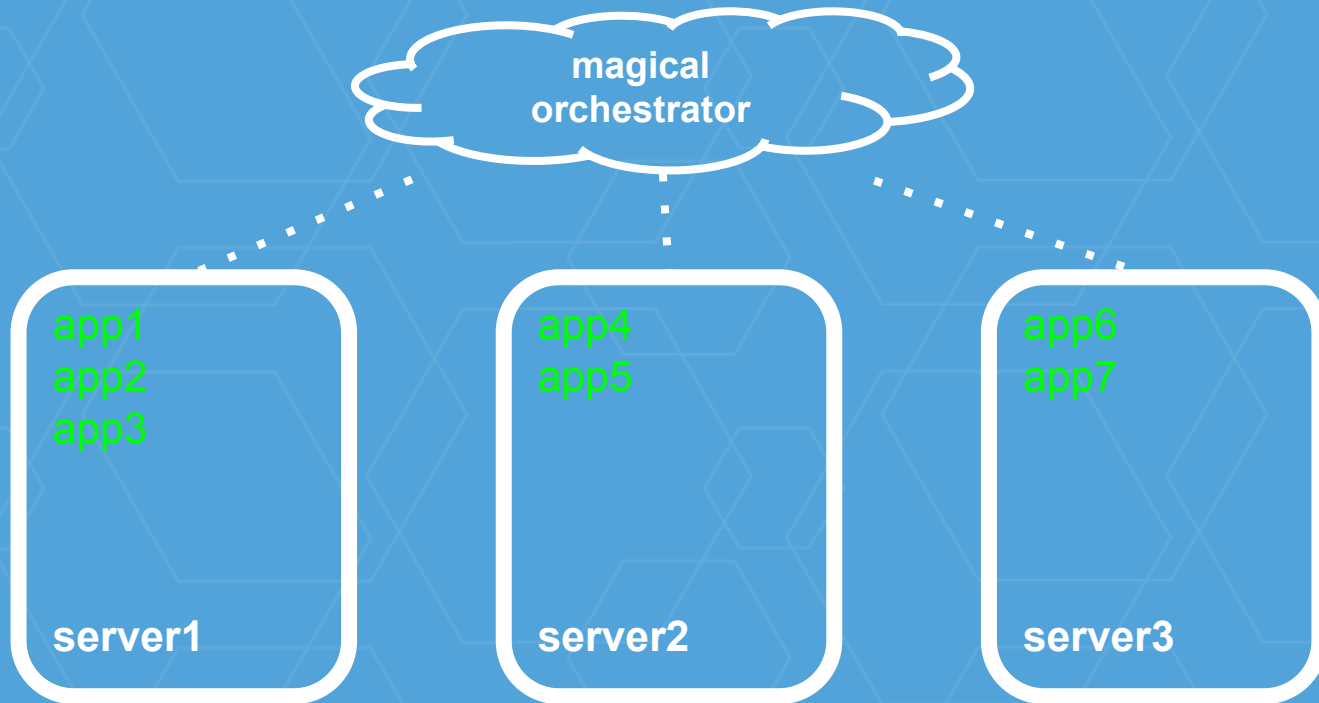
# Without orchestration



rebooting...

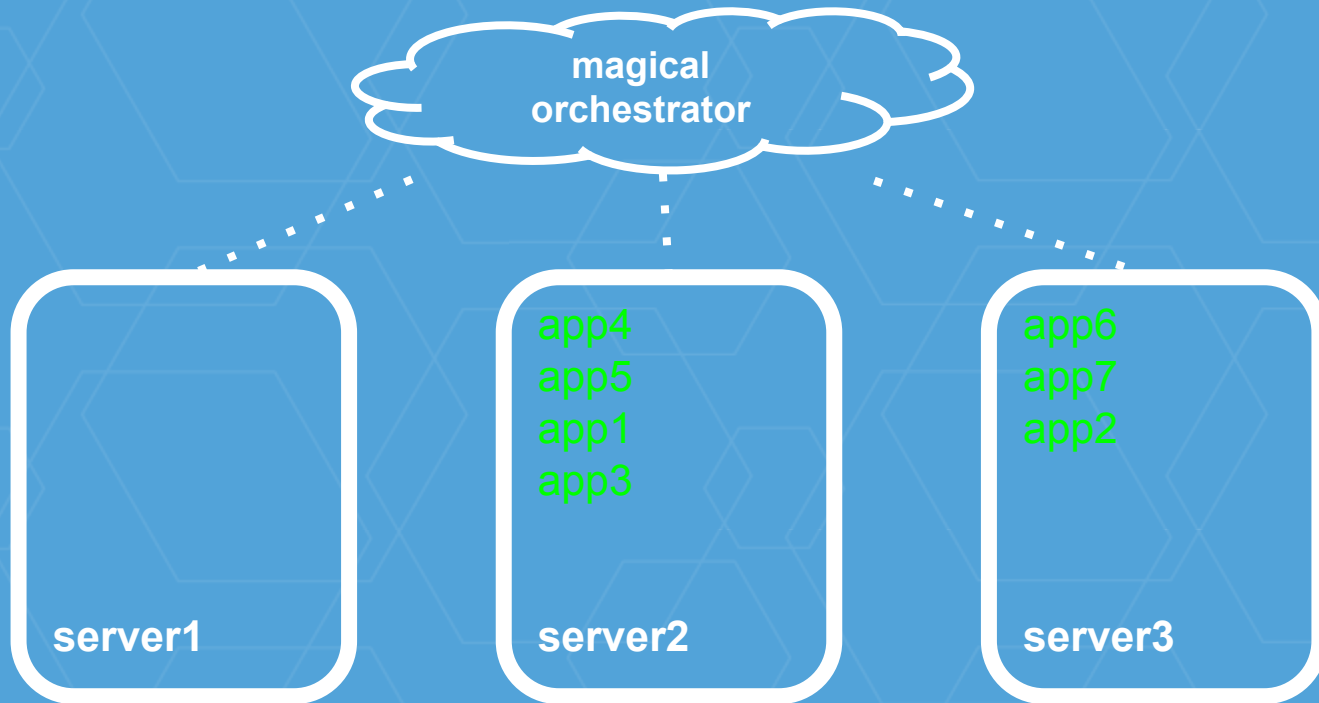


# With orchestration



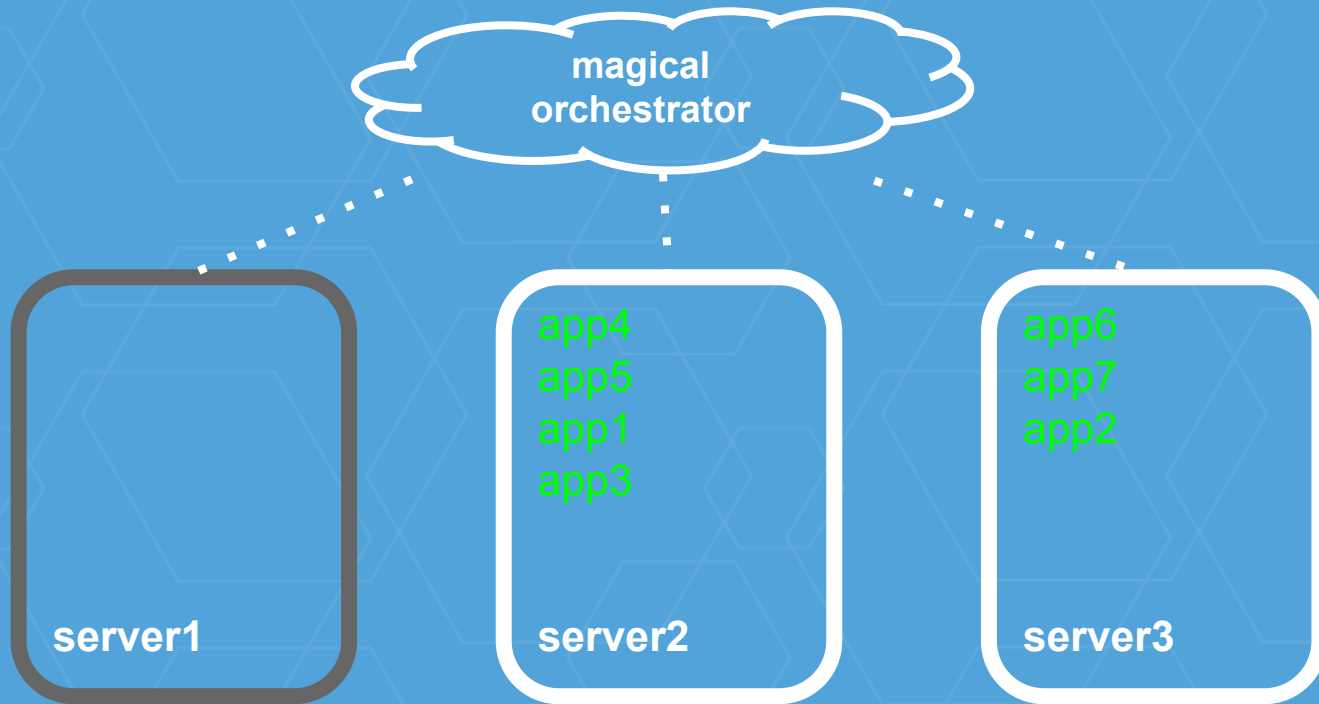
**needs reboot**

# With orchestration



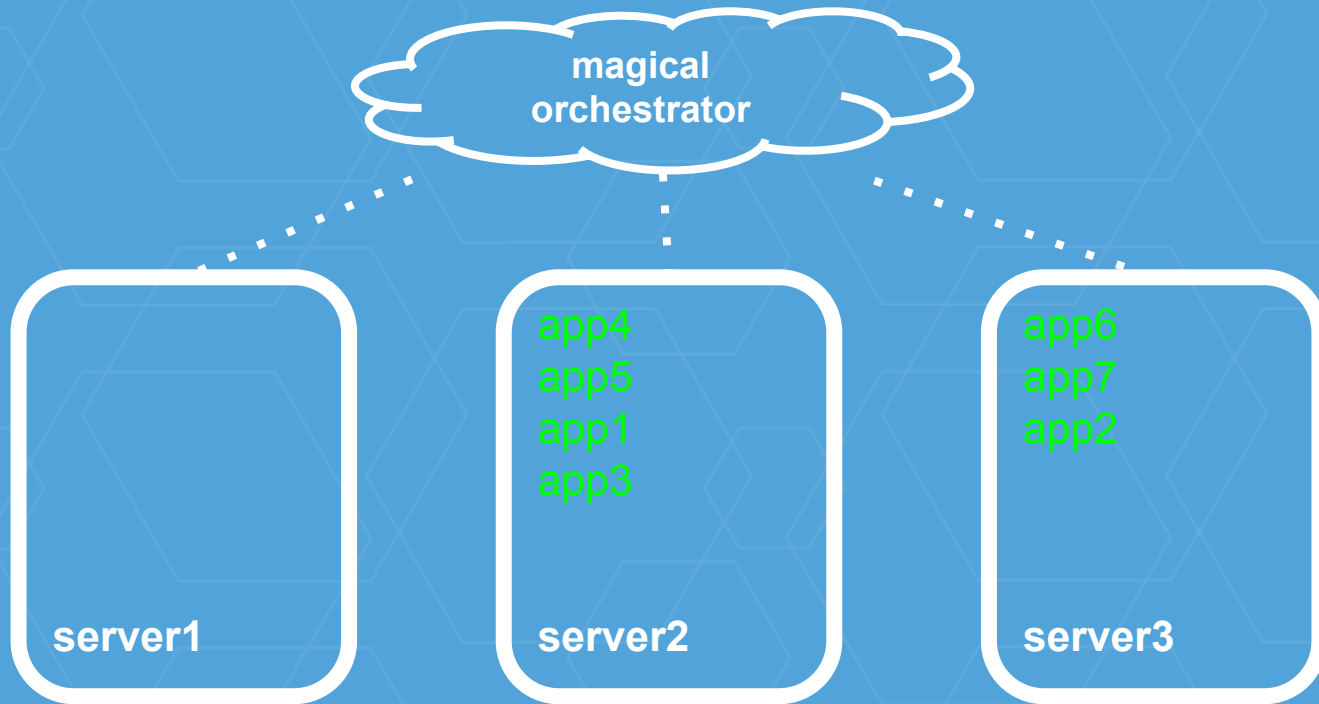
**needs reboot**

# With orchestration



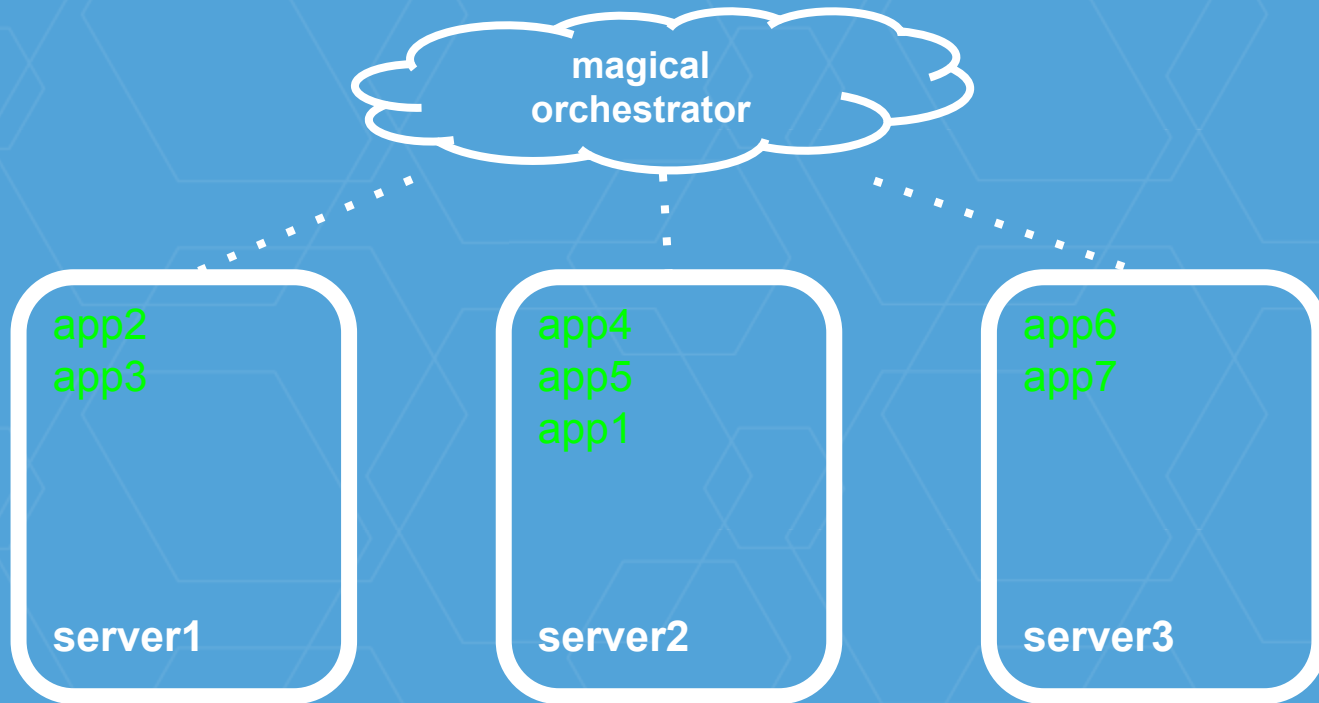
**rebooting...**

# With orchestration



updated!

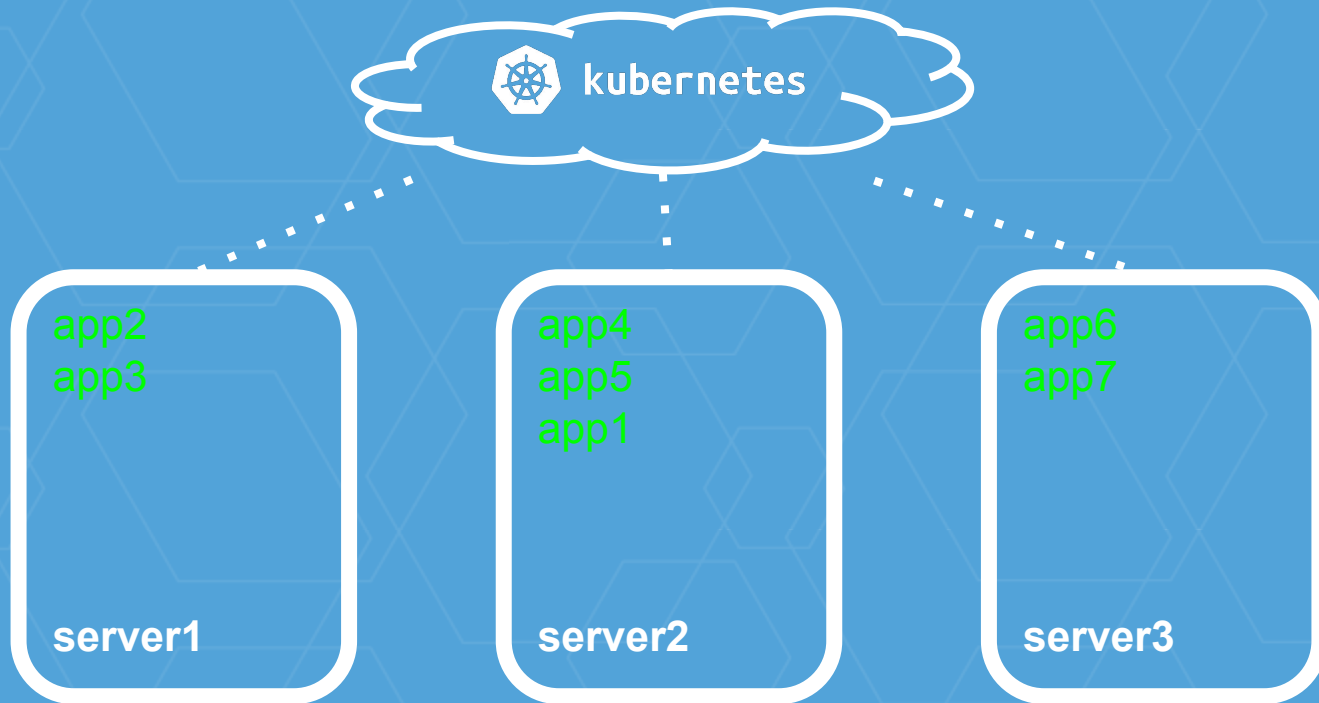
# With orchestration



**updated!**



# With orchestration



# Why container runtimes and orchestration?

So we can provide *seamless updates* and push forward the security of application servers

The background is a solid blue color with a repeating pattern of white-outlined hexagons of various sizes, creating a honeycomb-like texture.

# Why rkt?

**A long time ago in an  
ecosystem far, far away....**

(2014, to be precise)

# 2014

- Popular incumbent container tool (in CoreOS)
- *Common* practices, but few *best* practices
  - unsigned images (`curl | sudo sh -`)
  - inefficient/insecure images (`FROM ubuntu:14.04`)
  - PID1 or not to PID1 ([zombie reaping problem](#))
- New platforms emerging, difficult to integrate
  - `systemd + dockerd` = sad times had by all

# 2014 (December)

- Enter rkt (and appc)
  - Create an *alternative* container runtime (competition drives innovation)
  - Emphasise the importance of *security* and *composability*
  - Spur conversation around *standards* in the application container ecosystem



a modern, secure container runtime  
a simple, composable tool  
an implementation of open standards



a modern, secure container runtime  
a simple, composable tool  
**an implementation of open standards**





a standard application container  
open specification  
associated tooling

# appc spec in a nutshell

- Image Format (ACI)
  - what does an application consist of?
  - how can an image be located on the internet?
  - how can an image be securely signed & distributed?
- Pods
  - how can applications be grouped and run?
- Runtime format (ACE)
  - what does the execution environment look like?

# appc pods

- grouping of applications executing in a shared context (network, namespaces, volumes)
- shared fate
- the *only* execution primitive: single applications are modelled as singleton pods

# appc pods $\approx$ Kubernetes pods

- grouping of applications executing in a shared context (network, namespaces, volumes)
- shared fate
- the *only* execution primitive: single applications are modelled as singleton pods



a modern, secure container runtime

a simple, composable tool

**an implementation of open standards (appc)**

# 2016

- Docker and rkt both:
  - post 1.0
  - "production ready" (and actively used in production)
- Kubernetes too!
- Container standards?
  - ongoing...

# Container standards

- appc (December 2014)
- OCI (June 2015)
- CNCF (December 2015)



# 2015: appc vs OCI

## appc

- Image format
  - Cryptographic identity
  - Signing
  - Discovery/federation
- Runtime format
- Pods

## OCI

- Runtime format



# 2016: today

- New OCI project: *image format*
  - [github.com/opencontainers/runtime-spec](https://github.com/opencontainers/runtime-spec)
  - [github.com/opencontainers/image-spec](https://github.com/opencontainers/image-spec)
- Merging the best of Docker + appc
  - Container peace?!



+



# OCI Image Format maintainers

- Vincent Batts, **Red Hat**
- Brandon Philips, **CoreOS**
- Brendan Burns, **Google**
- Jason Bouzane, **Google**
- John Starks, **Microsoft**
- Jonathan Boulle, **CoreOS**
- Stephen Day, **Docker**

# Image Formats and standards

|                                  | Docker v1 | appc          | Docker v2.2   | OCI (in progress) |
|----------------------------------|-----------|---------------|---------------|-------------------|
| <b>Introduced</b>                | 2013      | December 2014 | April 2015    | April 2016        |
| <b>Content-addressable</b>       | No        | Yes           | Yes           | Yes               |
| <b>Signable</b>                  | No        | Yes, optional | Yes, optional | Yes, optional     |
| <b>Federated namespace</b>       | Yes       | Yes           | Yes           | Yes               |
| <b>Delegatable DNS namespace</b> | No        | Yes           | No            | Yes               |

# Container standards

Why should you care?

- For users, things should "just work":
  - `docker run example.com/org/app:v1.0.0`
  - `rkt run example.com/org/app,version=v1.0.0`
- For administrators and operators:
  - Stronger security primitives built-in
  - Intercompatibility: mix and match tools, or write your own



a modern, secure container runtime

**a simple, composable tool**

an implementation of open standards (appc)

# **rkt architecture**

A quick introduction

# rkt - simple CLI tool

no central daemon

no (mandatory) API

self-contained execution

apps run directly under spawning process

bash/systemd/kubelet



rkt run ...



application(s)



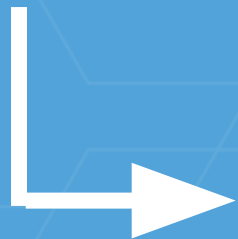
# rkt internals

modular architecture  
execution divided into *stages*  
stage0 → stage1 → stage2

# rkt internals

modular architecture  
take advantage of different technologies  
provide a consistent experience to users

bash/systemd/kubelet



rkt run ...

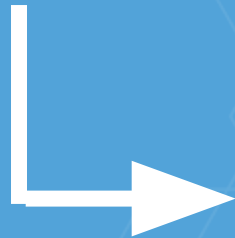


application(s)

bash/systemd/kubelet



rkt run ...



pod

bash/systemd/kubelet... (invoking process)



rkt (stage0)



pod (stage1)



app1 (stage2)

app2 (stage2)

# stage0 (rkt binary)

- primary interface to rkt
- discover, fetch, manage application images
- set up pod filesystems
- manage pod lifecycle
  - rkt run
  - rkt image list
  - rkt gc
  - ...

# stage1 (swappable execution engines)

- default implementation
  - based on systemd-nspawn+systemd
  - Linux namespaces + cgroups for isolation
- kvm implementation
  - based on lkvm+systemd
  - hardware virtualisation for isolation
- others?
  - e.g. xhyve (OS X), unc (unprivileged containers)

## stage2 (inside the pod)

- actual app execution
- independent filesystems (chroot)
- shared namespaces, volumes, IPC, ...



# rkt TPM measurement (new!)

- TPM, Trusted Platform Module
  - physical chip on the motherboard
  - cryptographic keys + processor
- Used to "*measure*" system state
- Historically just use to verify bootloader/OS (on proprietary systems)

# rkt TPM measurement (new!)

- CoreOS added support to GNU Grub
- rkt can now record information about running pods in the TPM
- attestable record of what images and pods are running on a system

# rkt TPM measurement (new!)

Containers

**rkt**

Verify images with trusted keys

Verify configuration state

OS

**CoreOS Linux**

Verify integrity of the OS release

Hardware

**Firmware & TPM**

Customer key embedded in firmware

Tamper-proof  
Audit log

# rkt API service (new!)

- *optional*, gRPC-based API daemon
- exposes *read-only* information on pods/images
- runs as *unprivileged* user
- easier integration with other projects

# Why rkt?

Secure  
Standards  
Composable



+



kubernetes

# rkt + Kubernetes

rkt ♥ k8s in a few ways:

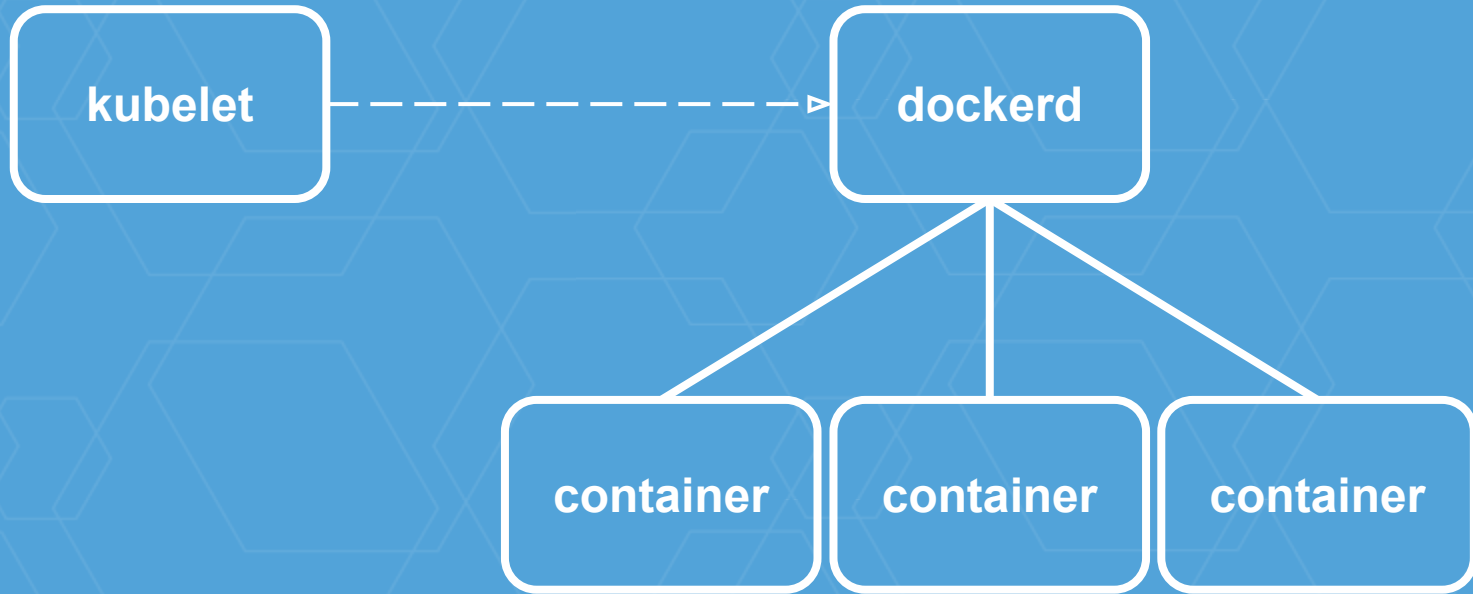
- using rkt as container runtime (aka "*rktnetes*")
- using rkt to run Kubernetes ("*rkt fly*")
- integrating with rkt networking (CNI)

# Kubelet + Container Runtimes

- Kubelet code provides a Runtime interface
  - SyncPod()
  - GetPod()
  - KillPod()
  - ...
- in theory, anyone can implement this
- in practise, lots of Docker assumptions



# Kubelet + Docker (default)

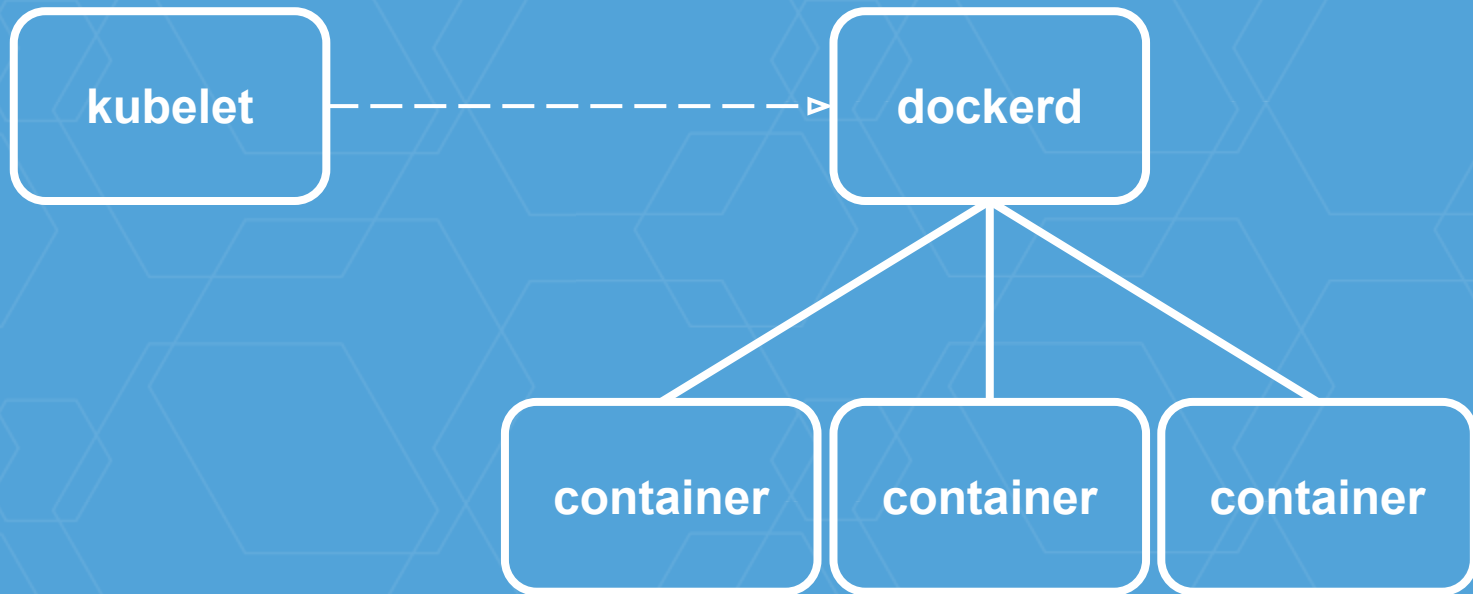


# Kubelet + Docker (default)

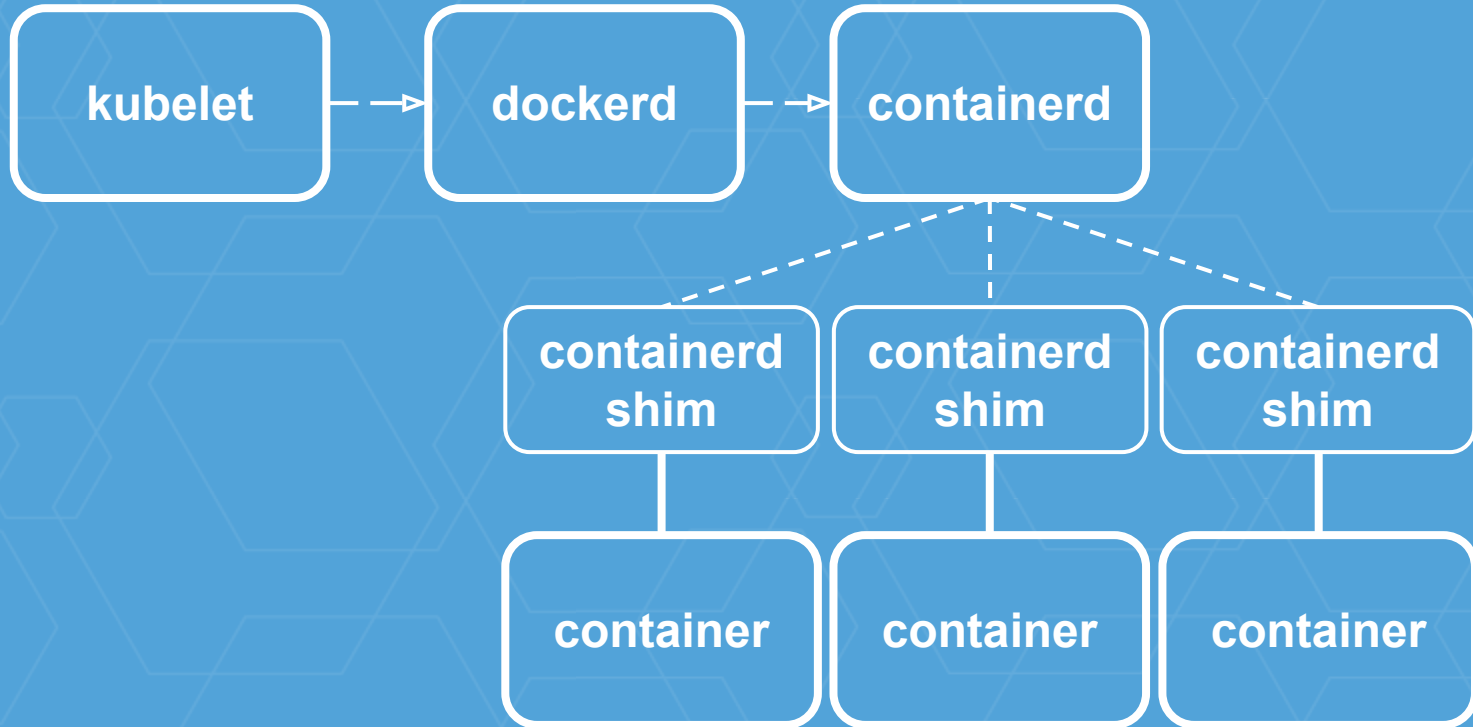
## Problems:

- Docker doesn't understand *Pods*
  - kubelet must maintain pod<->container mapping
  - "infra container" to hold namespaces for pod
- dockerd = SPOF for node
  - if Docker goes down, so do all containers
- Docker doesn't interact well with systemd
  - References

# Kubelet + Docker (before 1.11+)



# Kubelet + Docker (1.11+ with containerd)

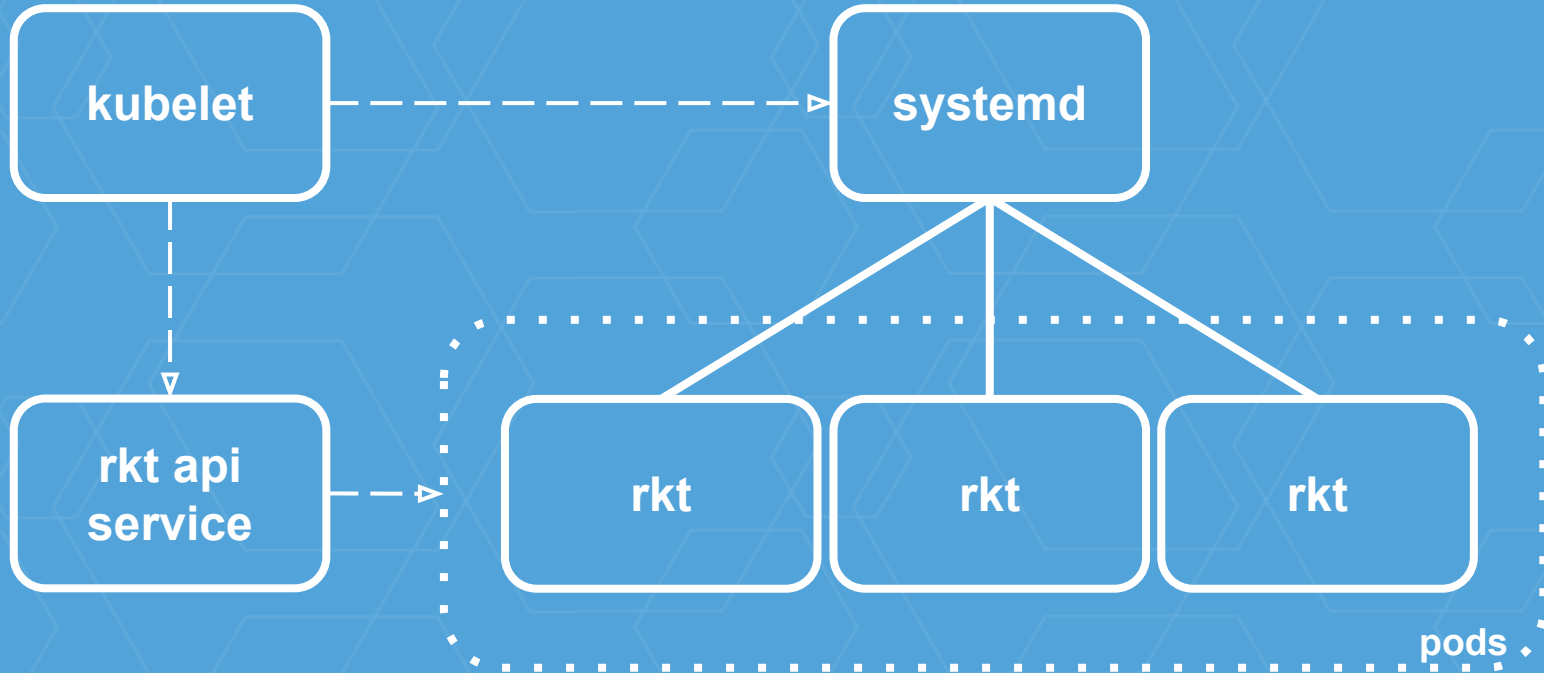


# Kubelet + rkt (rktnetes)

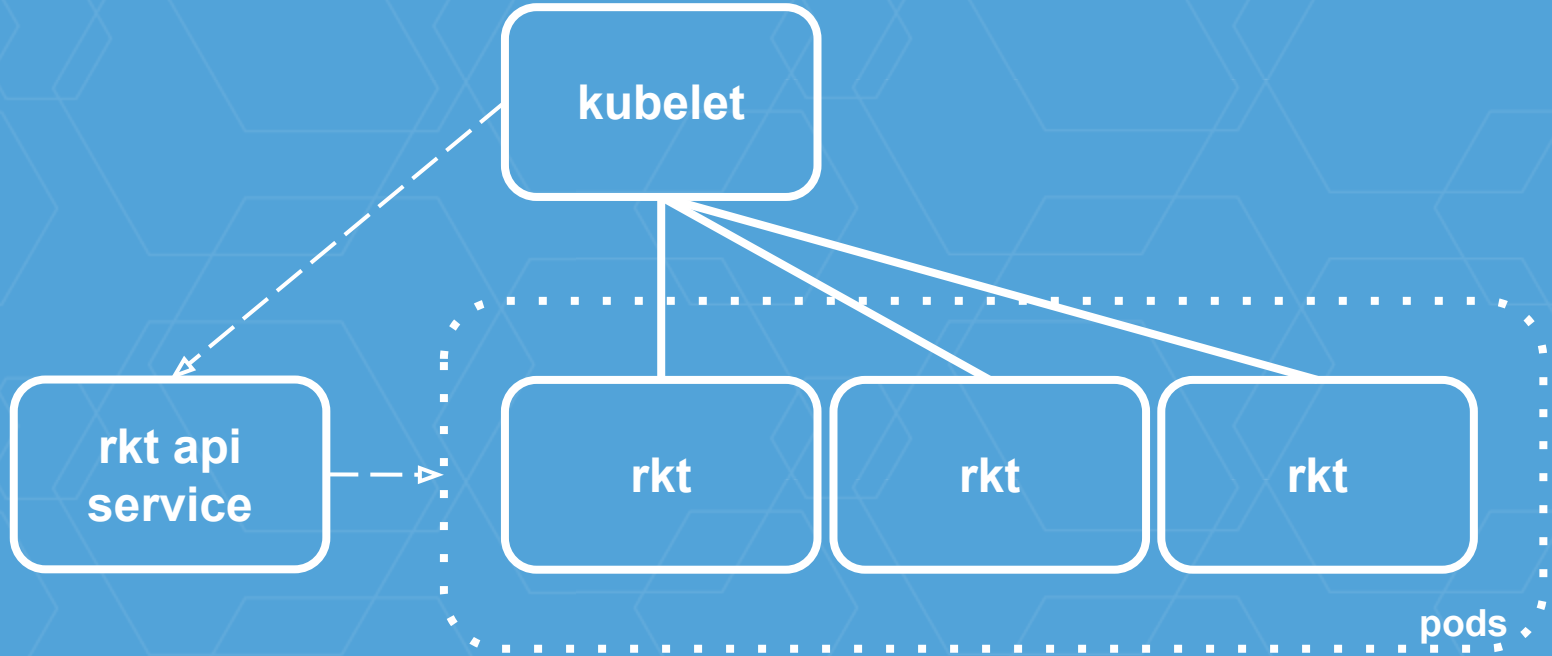
Using rkt as the kubelet's container runtime

- *A pod-native* runtime
- First-class integration with systemd hosts
- *self-contained pods* process model = no SPOF
- Multi-image compatibility (e.g. docker2aci)
- Transparently swappable

# Kubelet + rkt (rktnetes - with systemd)



# Kubelet + rkt (rktnetes - without systemd)



# rktnetes today

Nearly complete!

~90% of end-to-end tests passing

<http://rktnetes.io/>

P0 for Kubernetes 1.3



# Kubelet + Container Runtimes

- Kubelet's Runtime interface rework
  - Granular control over applications in containers
  - Dynamic resource management
  - Directly managing containers?

# Using rkt to run Kubernetes

- Kubernetes components are largely self-hosting, but not entirely
  - Need a way to bootstrap kubelet on the host
  - kubelets can then host control plane components
- On CoreOS, this means in a container..
  - ... but kubelet has some unique requirements (like mounting volumes on the host)

# Using rkt to run Kubernetes

- rkt "*fly*" feature (new in 0.15.0+)
- unlike rkt run, does *\*not\** execute pods
- execute a *single application* in an *unconstrained environment*
- all the other advantages of rkt (image discovery, signing/verification, management)

bash/systemd/... (invoking process)

└─▶ rkt (stage0) - without *fly*

└─▶ pod (stage1)

└─▶ app1 (stage2)

└─▶ app2 (stage2)

bash/systemd/... (invoking process)

└─▶ rkt (stage0) - without *fly*

└─▶ pod (stage1)

└─▶ app1 (stage2)

└─▶ app2 (stage2)

*Isolated mount (and PID, ...) namespace*

bash/systemd/... (invoking process)

└─▶ rkt (stage0) - with *fly*

└─▶ application

bash/systemd/... (invoking process)

└─▶ rkt (stage0) - with *fly*

└─▶ application

*Host mount (and PID, ...) namespace*

bash/systemd/... (invoking process)

└─▶ rkt (stage0) - with *fly*

└─▶ kubelet

*Host mount (and PID, ...) namespace*



# rkt networking

Plugin-based

IP(s)-per-pod

Container Networking Interface (CNI)

# Container Runtime (e.g. rkt)

---

## Container Networking Interface (CNI)

---

**ptp**

**macvlan**

**ipvlan**

**OVS**

# CNI in a nutshell

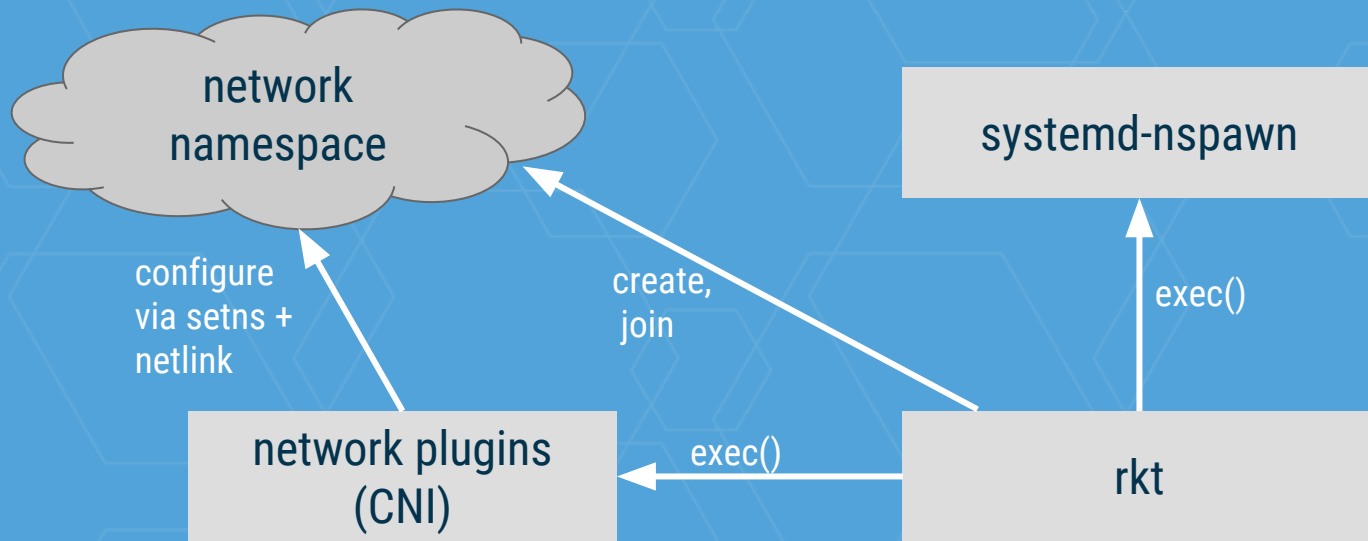
- Container can join multiple networks
- Network described by JSON config
- Plugin supports two commands
  - ADD container to the network
  - REMOVE container from the network
- Plugins are responsible for all logic
  - allocating IPs, talking to backend components, ...

# CNI: example configuration

```
{  
  "name": "mynet",  
  "type": "ptp",  
  "ipam": {  
    "type": "host-local",  
    "subnet": "10.1.1.0/24"  
  }  
}
```

```
$ rkt run --net=mynet coreos.com/etcd
```

# How rkt uses CNI



`/var/lib/rkt/pods/run/$POD_UUID/netns`

# Kubernetes networking

Plugin-based (but never left alpha)

IP(s)-per-pod  
(sound familiar?)

# Kubernetes and CNI

Soon to be "*the* Kubernetes plugin model"



# CNI today

v0.2.0-rc1

Handles all networking in rkt

Integrations with Project Calico, Weaveworks

Hoping to donate to the CNCF



# Looking ahead

What's coming up for rkt and Kubernetes

# rktnetes 1.0

2016Q2

Fully supported, full feature parity  
Automated end-to-end testing on CoreOS

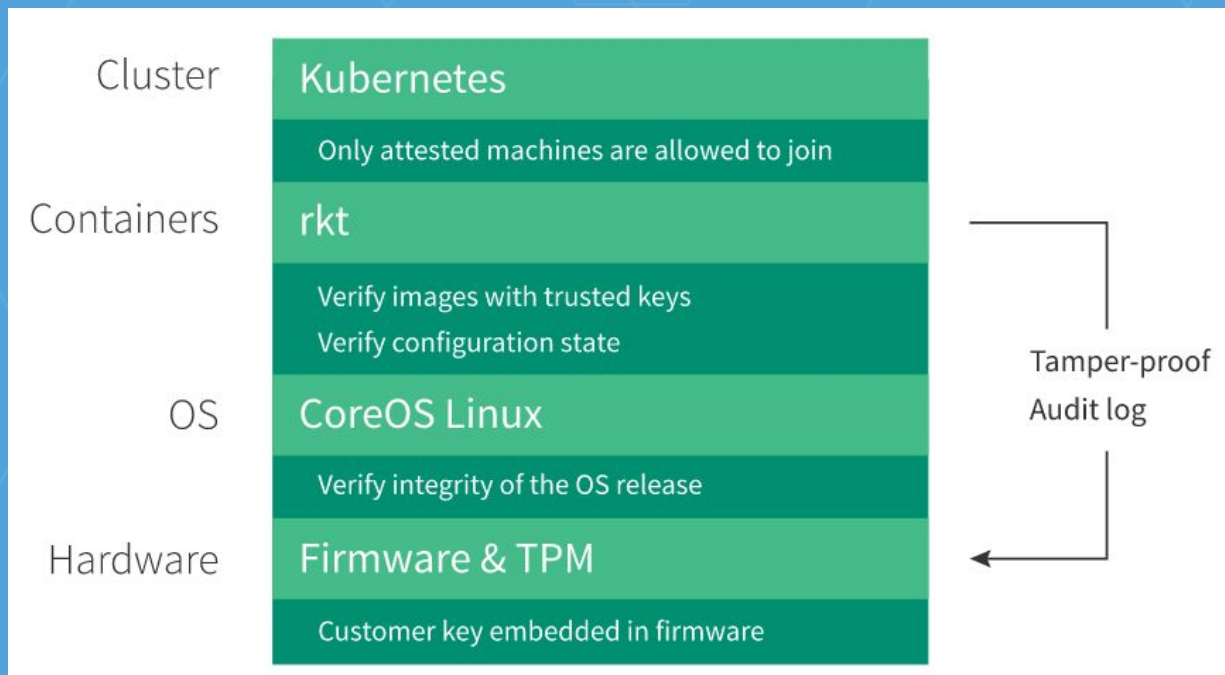
# **rktnetes 1.0+**

LKVM backend by default

Native support for OCI in Kubernetes API

TPM up to the Kubernetes level

# Tectonic Trusted Computing



<https://coreos.com/blog/coreos-trusted-computing.html>

# Try rktnetes today!

<https://gist.github.com/yifan-gu/091d258107f37ecdff48>

# rkt 1.0+

- Loads of bugfixes
- app exit status propagation
- Discover and fetch stage1 images
  - e.g. from [coreos.com](https://coreos.com)

# rkt 1.0+

- rktnetes fixes
  - Hostname
  - Docker volume semantics
  - Improvements in the API Service
- SELinux support on Fedora
  - Fixes in nspawn, selinux-policy and rkt
- Concurrent image fetching

# What's next?

- rkt fly as a top-level command
- IPv6 support on CNI
- Full SELinux enforcing on Fedora
- Packaged in Debian (almost there!)



# CNI 1.0

Stable configuration

Stateless plugins (runtime responsibility)

IPv6

*<your suggestions here>*

# Kubernetes 1.3

Move all networking code into CNI plugins

# Kubelet upgrades

- Remember from CoreOS mission:  
"updates must be *automatic and seamless*"
- If kubelet is in OS, must be upgraded in lock-step
- But mixed-version clusters don't always work  
(e.g. upgrading from 1.07 - 1.1.1: <https://github.com/kubernetes/kubernetes/issues/16961> )

# Kubelet upgrades

- Solution: API driven upgrades
- Small agent living on host, invoking kubelet (using rkt fly)
- Reading annotations from the kubelet API server
- Follow along:

<https://github.com/coreos/bugs/issues/1051>

# Graceful kubelet shutdown

- When an update is ready, locksmith signals kubelet to gracefully shut down
- Kubernetes can then gracefully migrate apps before shutdown
- <https://github.com/coreos/bugs/issues/1112>
- <https://github.com/kubernetes/kubernetes/issues/7351>

# tl;dr:

- Use rkt
- Use Kubernetes
- Use rkt + Kubernetes (rktnetes)
- Get involved and help define the future of application containers and Kubernetes



**May 9 & 10, 2016 - Berlin, Germany**

**[coreos.com/fest](http://coreos.com/fest) - [@coreosfest](https://twitter.com/coreosfest)**

# Questions?



kubernetes



CoreOS

## Join us!

contribute: [github.com/coreos/rkt](https://github.com/coreos/rkt)

careers: [coreos.com/careers](https://coreos.com/careers) *(now in Berlin!)*



# Extra slides

Did I speak too fast?

# rkt security ("secure by default")

- image security
  - cryptographic addressing
  - signature verification
- privilege separation
  - e.g. fetch images, expose API (new!) as non-root
- SELinux integration
- lkvm stage1 for true hardware isolation
- TPM attestation (new!)