CoreOS

Clustered computing with CoreOS, fleet and etcd

DEVIEW 2014

Jonathan Boulle
CoreOS, Inc.

Who am I?

Jonathan Boulle

@baronboulle

jonboulle

South Africa -> Australia -> London -> San Francisco

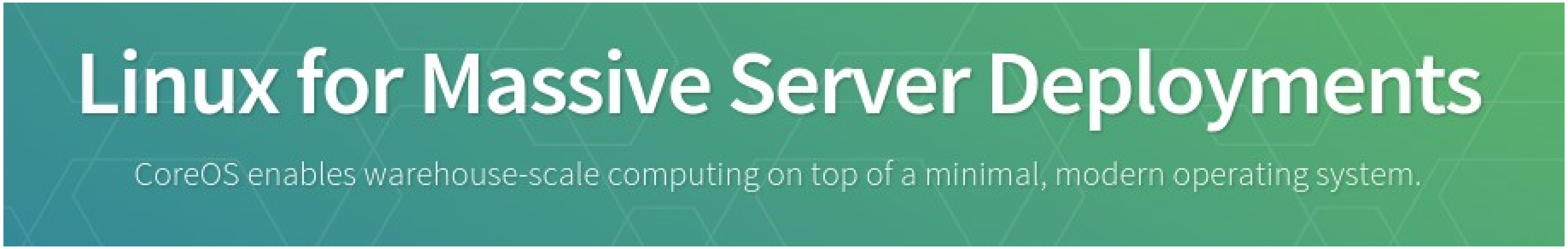
Red Hat -> Betfair -> Twitter -> **CoreOS**

Engineer: *Linux, Python, Go, FOSS*

Agenda

- CoreOS Linux
  - Securing the internet
  - Application containers
  - Automatic updates
- fleet
  - cluster-level init system
  - `etcd + systemd`
- fleet and...
  - systemd: the good, the bad
  - etcd: the good, the bad
  - Golang: the good, the bad
- Q&A

CoreOS Linux



A minimal, automatically-updated Linux distribution, designed for distributed systems.

Why?

- CoreOS mission: “Secure the internet”
- Status quo: set up a server and never touch it
- Internet is full of servers running years-old software with dozens of vulnerabilities
- CoreOS: make updating the default, seamless option
    - Regular
    - Reliable
    - Automatic

How do we achieve this?

- Containerization of applications
- Self-updating operating system
- Distributed systems tooling to make applications resilient to updates

Application Containers

- Bundle all dependencies with the application to make a portable, self-contained bundle to run on any Linux system
- Docker is leading this space
  - installed by default on CoreOS

KERNEL

SYSTEMD

SSH

DOCKER

PYTHON

JAVA

NGINX

MYSQL

OPENSSL

distro distro distro distro distro distro distro distro distro distro distro

APP

KERNEL

SYSTEMD

SSH

DOCKER

distro distro distro distro distro distro distro distro distro distro distro

PYTHON

JAVA

NGINX

MYSQL

OPENSSL

APP

CoreOS

KERNEL
SYSTEMD
SSH
DOCKER

distro distro distro distro distro distro distro distro distro distro distro

Application Containers

PYTHON
JAVA
NGINX
MYSQL
OPENSSL

APP

Core OS

KERNEL
SYSTEMD
SSH
DOCKER

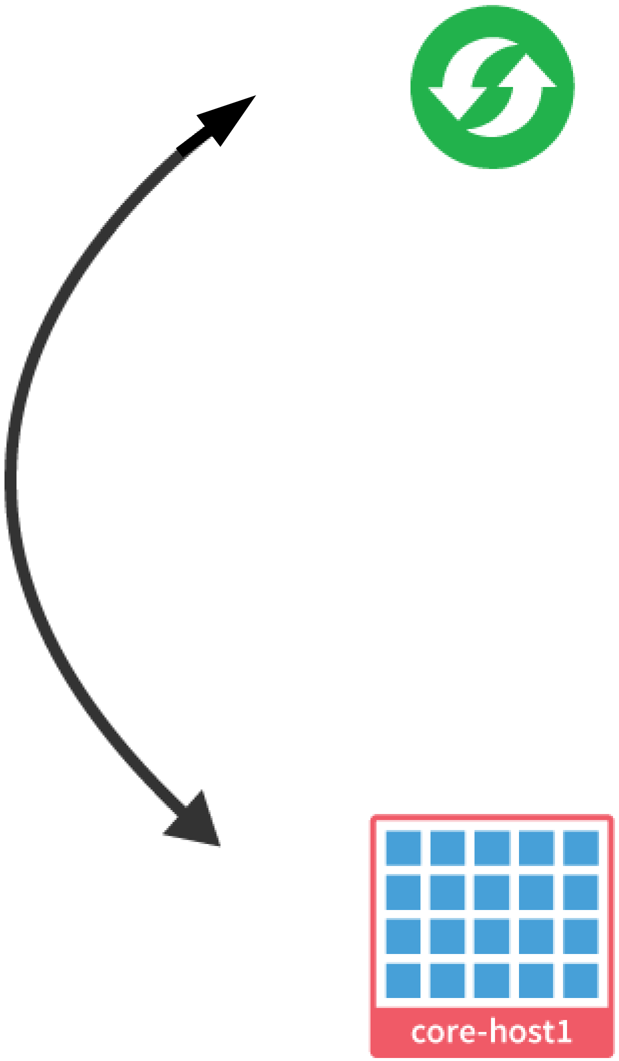
- Minimal base OS (~100MB)
- Vanilla upstream components wherever possible
- Decoupled from applications
- Automatic, atomic updates

Automatic updates

How do updates work?

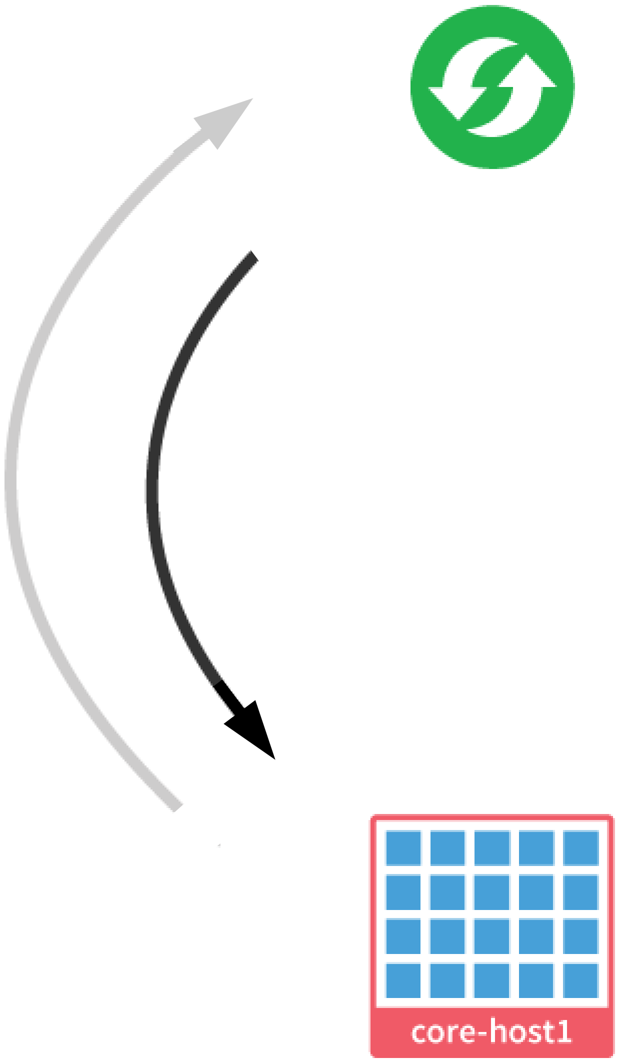
- Omaha protocol (check-in/retrieval)
  - Simple XML-over-HTTP protocol developed by Google to facilitate polling and pulling updates from a server

Omaha protocol



Client sends application id and current version to the update server

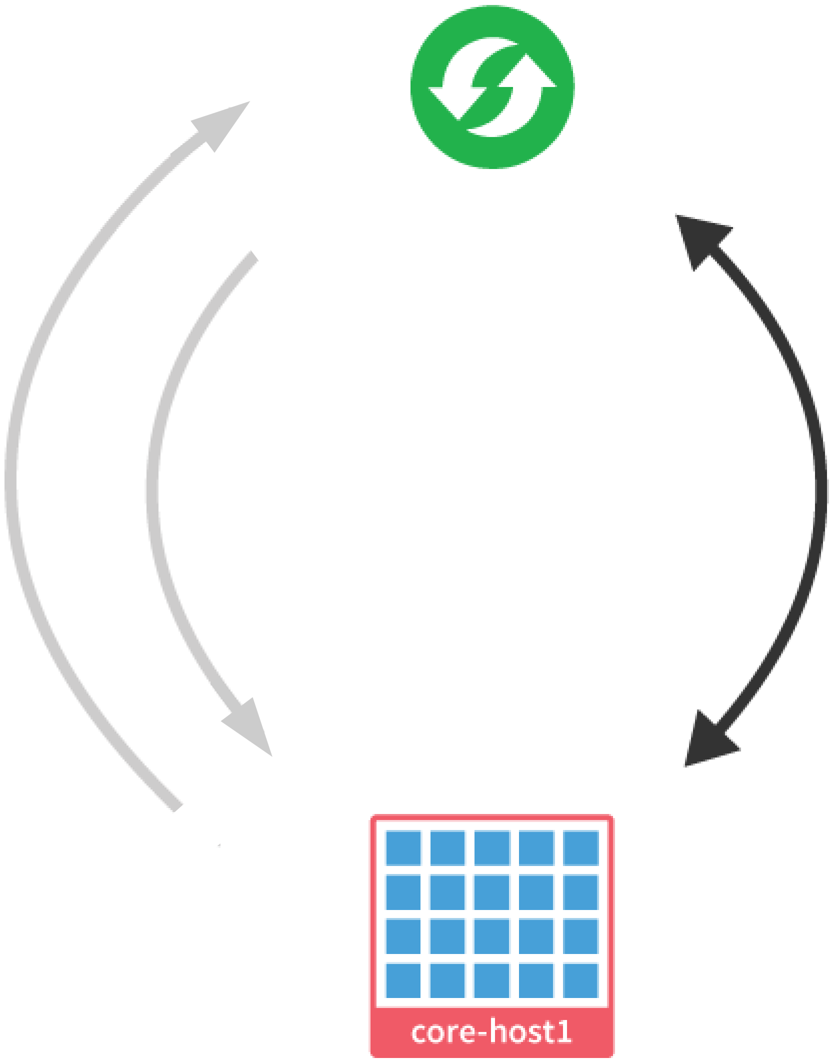
Omaha protocol



Client sends application id and current version to the update server

Update server responds with the URL of an update to be applied

Omaha protocol

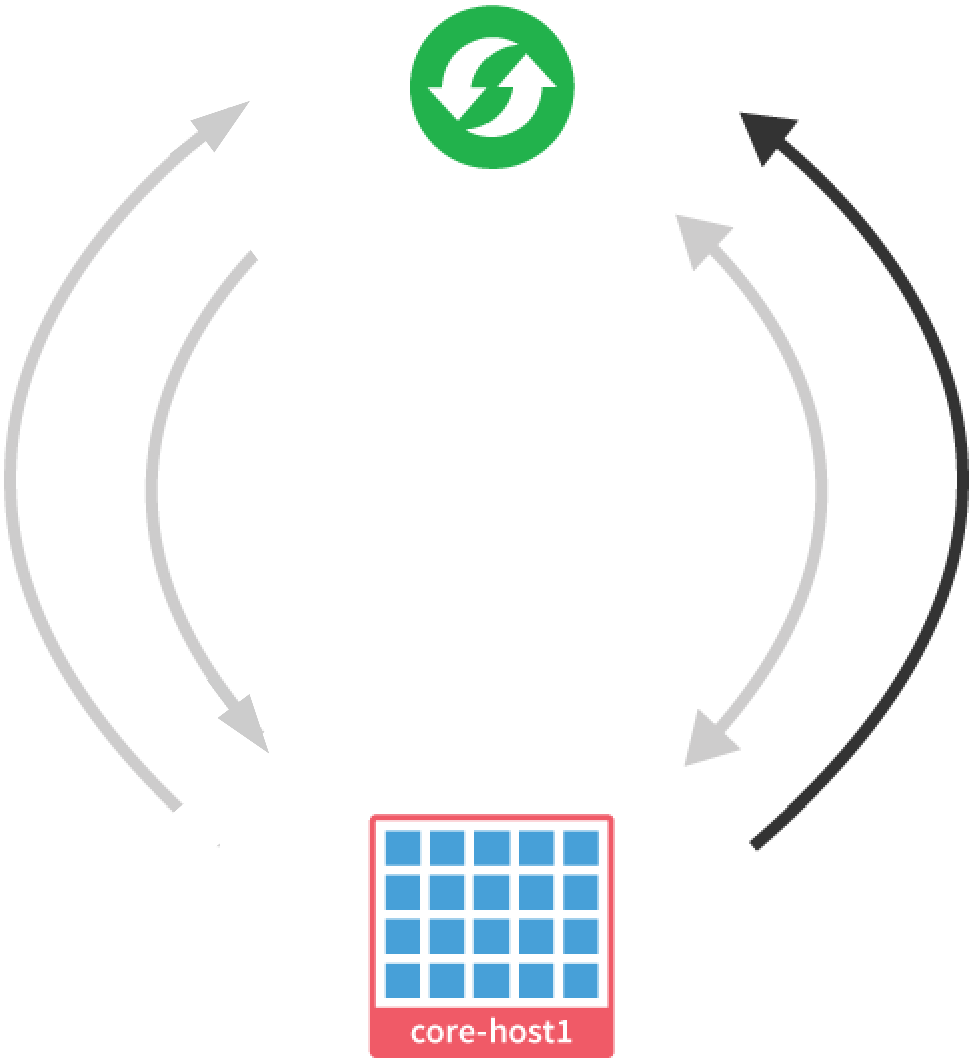


Client sends application id and current version to the update server

Update server responds with the URL of an update to be applied

Client downloads data, verifies hash & cryptographic signature, and applies the update

Omaha protocol



Client sends application id and current version to the update server

Update server responds with the URL of an update to be applied

Client downloads data, verifies hash & cryptographic signature, and applies the update

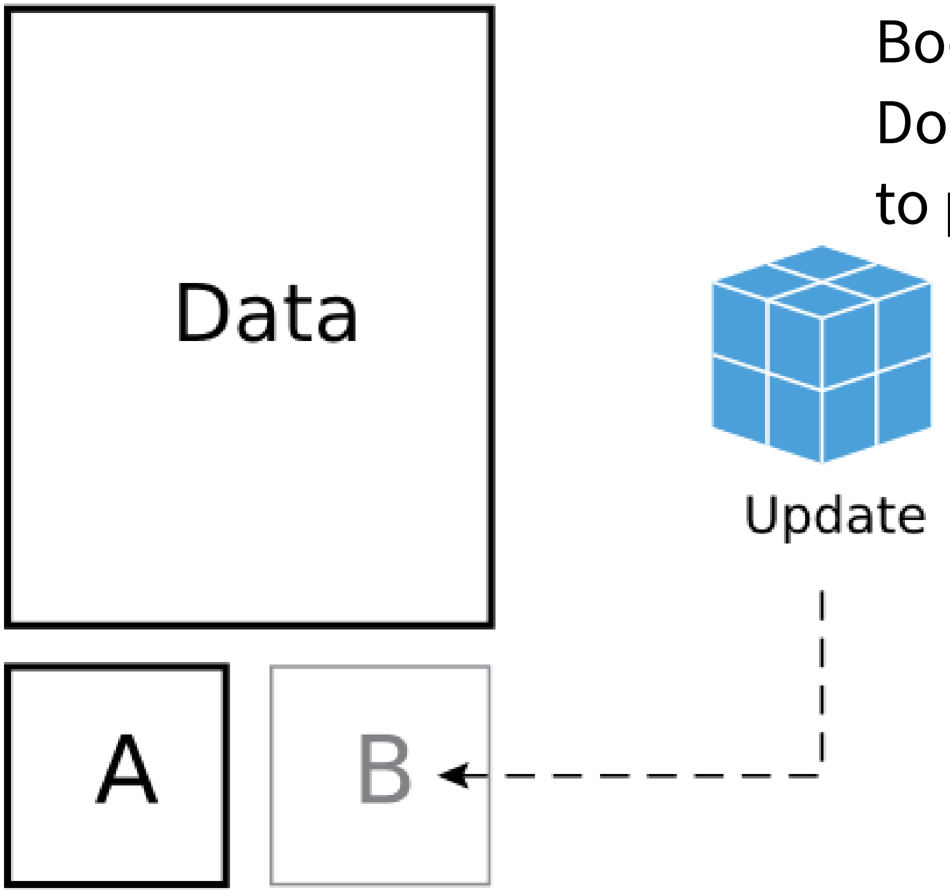
Updater exits with response code then reports the update to the update server

Automatic updates

How do updates work?

- Omaha protocol (check-in/retrieval)
  - Simple XML-over-HTTP protocol developed by Google to facilitate polling and pulling updates from a server
- Active/passive read-only root partitions
  - One for running the live system, one for updates

Active/passive root partitions



Booted off partition A.
Download update and commit to partition B.

Active/passive root partitions

Reboot. If tests on partition B succeed, continue normal operation.

Active/passive root partitions

But what if partition B fails update tests...

Active/passive root partitions

Revert to known successful partition, reboot, try again.

Active/passive root partitions

Active/passive ~~root~~ partitions

- Read-only images containing *most* of the OS
  - Mounted read-only onto `/usr`
  - `/` is mounted read-write on top (persistent data)
  - Parts of `/etc` generated dynamically at boot
  - A lot of work moving default configs from `/etc` to `/usr`

```
# /etc/nsswitch.conf:

passwd:       files usrfiles
shadow:       files usrfiles
group:        files usrfiles
```

`/usr/share/baselayout/passwd`

Atomic updates

- Entire OS is a single read-only image
  - Easy to verify cryptographically
    - `sha1sum` on AWS or bare metal gives the same result
  - No chance of inconsistencies due to partial updates
    - e.g. pull a plug on a CentOS system during a `yum update`
    - At large scale, such events are inevitable

But…

- **Problem**:
  - updates still require a reboot (to use new kernel and mount new filesystem)
  - Reboots cause application downtime…
- **Solution**: `fleet`
  - Highly available, fault tolerant, distributed process scheduler to run applications
  - fleet keeps applications running during server downtime

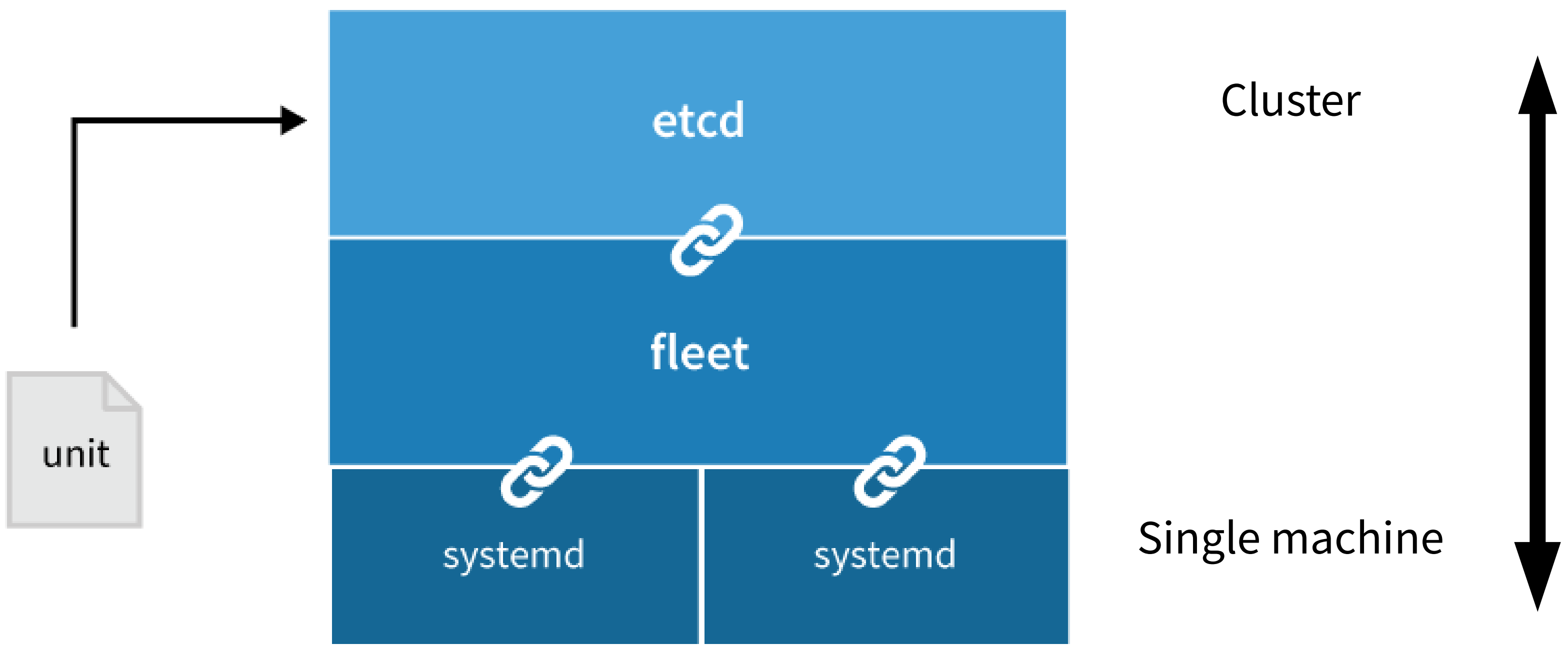
fleet – the "cluster-level init system"

- fleet is the abstraction between *machine* and *application:*
  - init system manages process on a machine; fleet manages applications on a cluster of machines
- Similar to Mesos, but very different architecture (e.g. based on etcd/Raft, not Zookeeper/Paxos)
- Uses systemd for machine-level process management, etcd for cluster-level co-ordination

fleet components

- `fleetd` binary (running on all CoreOS nodes)
  - encapsulates two roles:
    - *engine* (cluster-level unit scheduling – talks to etcd)
    - *agent* (local unit management – talks to etcd and systemd)
- `fleetctl` command-line administration tool
  - create, destroy, start, stop units
  - Retrieve current status of units/machines in the cluster
- HTTP API

fleet – high level view



systemd

- Linux init system (PID 1)
  - Relatively new, replaces SysVinit, upstart or others
  - Being adopted by all major Linux distributions
- Fundamental concept is the **unit**
  - Units include services (e.g. applications), mount points, sockets, timers, etc.
  - Each unit configured with a simple unit file

Quick comparison

compare

```
#!/bin/bash
#
# logstash              Startup script for logstash
# chkconfig: 2345 20 80
# description: Logstash is a log shipping, indexing, and colloc
ation tool.
# processname: java
### BEGIN INIT INFO
# Provides: logstash
# Required-Start: $local_fs $remote_fs
# Required-Stop: $local_fs $remote_fs
# Default-Start: 2 3 4 5
# Default-Stop: S 0 1 6
# Short-Description: Logstash
# Description: logstash is a tool for managing events and logs.
 You can use it
#              to collect logs, parse them, and store them for
later use (like,i
#              for searching).
### END INIT INFO

# Amount of memory for Java
JAVAMEM=256M
JAVA="/etc/alternatives/java"
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/b
in

# We use name to allow for running multiple instances of logsta
sh, thus
# supporting copying this file and supporting separate pids for
 shipper, web,
logstash [+]                                  24,1          Top
```

```
[Unit]
Description=Logstash

[Install]
WantedBy=multi-user.target

[Service]
User=logstash
Group=logstash
ExecStart=/opt/logstash/bin/logstash
Environment=JAVAMEM=256M
~
logstash.service [+]                                  11,24          All
```

fleet + systemd

- systemd exposes a D-Bus interface
  - D-Bus: message bus system for IPC on Linux
  - One-to-one messaging (methods), plus pub/sub abilities
- fleet uses godbus to communicate with systemd
  - Sending commands: `StartUnit`, `StopUnit`
  - Retrieving current state of units

systemd is great!

- Automatically handles:
  - Process daemonization
  - Resource isolation/containment (`cgroups`)
  - Health-checking, restarting failed services
  - Logging (journal)
    - applications can just write to stdout, systemd adds metadata
  - Timers, inter-unit dependencies, socket activation, ...

fleet + systemd

- systemd takes care of things so we don't have to
- fleet configuration is just systemd unit files
- fleet extends systemd features to the cluster-level, and adds some of its own (using `[X-Fleet]`)
  - Template units (run *n* identical copies of a unit)
  - Global units (run a unit everywhere in the cluster)
  - Machine metadata (run only on certain machines)

systemd is… not so great

- **Problem**: unreliable pub/sub
  - fleet agent initially used a systemd D-Bus subscription to track unit status
  - Every change in unit state in systemd triggers an event in fleet (e.g. “publish this new state to the cluster”)
  - Under heavy load, or byzantine conditions, unit state changes would be dropped
  - As a result, unit state in the cluster became stale

systemd is... not so great

- **Problem**: unreliable pub/sub
- **Solution**: polling for unit states
  - Every *n* seconds, retrieve state of units from systemd, and synchronize with cluster
  - Less efficient, but much more reliable
  - Any state inconsistencies are quickly fixed

systemd (and docker) are… not so great

- **Problem**: poor integration with Docker
  - Docker is *de facto* appplication container manager
  - Docker and systemd do not always play nicely together..
  - Both Docker and systemd manage cgroups and processes, and the results are mixed

systemd (and docker) are… not so great

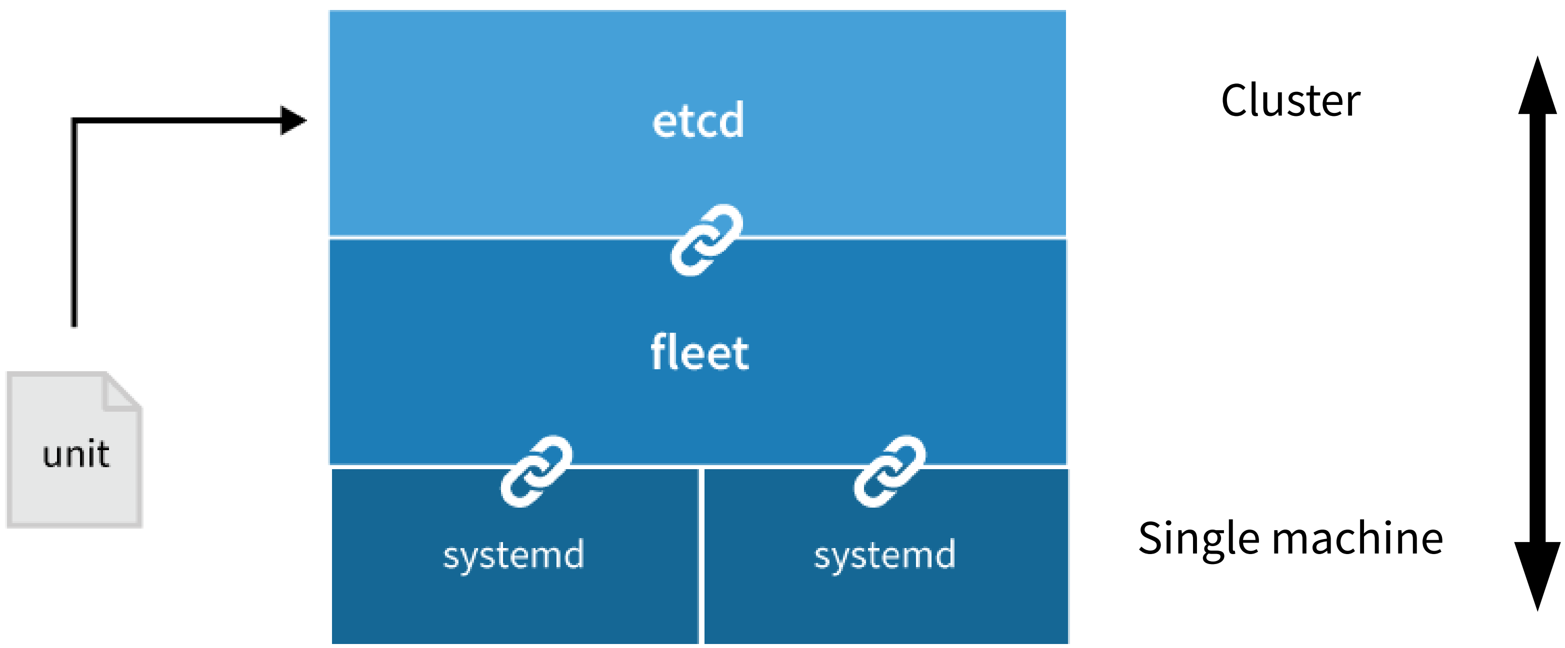
- **Problem**: poor integration with Docker
- **Example**: sending SIGTERM to a container
  - Given a simple container:

```
[Service]
ExecStart=/usr/bin/docker run busybox /bin/sh -c \
          "while true; do echo Hello World; sleep 1; done"
```

systemd (and docker) are... not so great

- **Problem**: poor integration with Docker
- **Solution**: ... work in progress
  - `system-docker` - small application that moves cgroups from Docker under systemd's control
  - Use Docker for image management, but `systemd-nspawn` for runtime (e.g. CoreOS's `toolbox`)
  - Docker standalone mode (start container directly rather than through daemon)

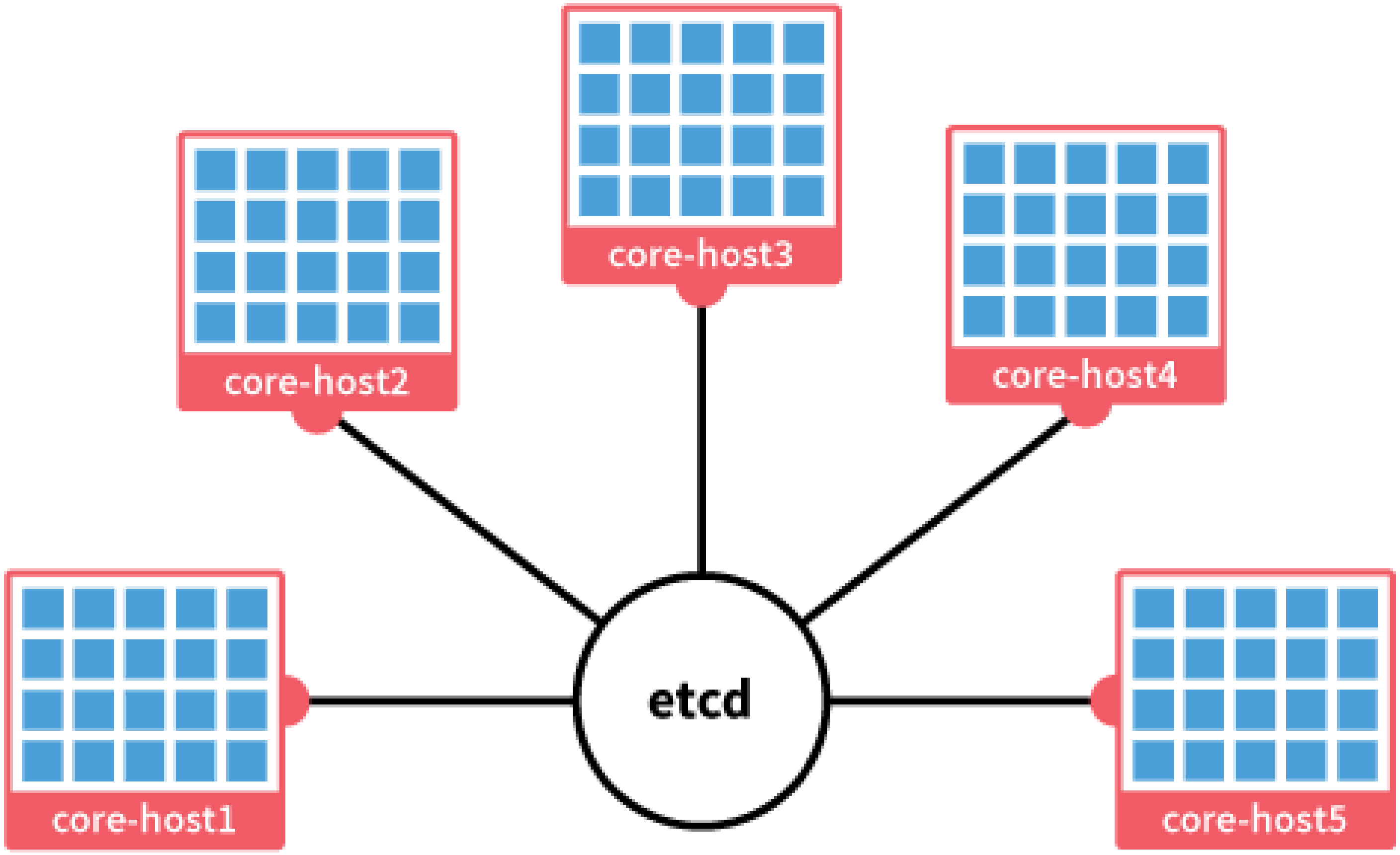
fleet – high level view



etcd

- A consistent, highly available key/value store
- Facilitates shared configuration, distributed locking
- Driven by Raft
  - Consensus protocol similar to Paxos
  - Designed for understandability and simplicity
- Popular and widely used
  - Simple HTTP API + libraries in Go, Java, Python, Ruby, ...

etcd connects CoreOS hosts



fleet + etcd

- Every CoreOS installation includes etcd and fleet, so fleet can always talk to a local instance of etcd
- fleet needs a consistent view of the cluster to make scheduling decisions: etcd provides this view
- All *unit files*, *unit state*, *machine state* and *scheduling information* is stored in etcd

etcd is great!

- Fast and simple API
- Handles all cluster-level communication so we don't have to
- Powerful primitives:
  - *Compare-and-Swap* allows for atomic operations and implementing locking behaviour
  - Watches provide event-driven behaviour

etcd is... not so great

- **Problem**: unreliable watches
  - fleet initially used a purely event-driven architecture
  - watches in etcd used to trigger events
  - Unfortunately, many places for things to go wrong...

Unreliable watches

- Example:
  - etcd is undergoing a leader election, watches do not work during this period
  - Change occurs (e.g. a machine leaves the cluster)
  - Event is missed and fleet never recovers

etcd is... not so great

- **Problem**: unreliable watches
  - fleet initially used a purely event-driven architecture
  - watches in etcd used to trigger events
- **Problem**: limited event history
  - Can “watch” from an arbitrary point in the past, but..
  - History is a limited window!
  - With a busy cluster, watches can fall out of this window

Limited event history

- Example:
  - etcd holds history of last 1000 events
  - fleet sets watch at *i=100* to check for machine loss
  - Meanwhile, many changes occur in other parts of the keyspace, advancing index to *i=1500*
  - Leader election/network hiccup occurs and severs watch
  - fleet tries to recreate watch at *i=100* and fails:

```
err="401: The event in requested index is outdated and
cleared (the requested history has been cleared [1500/100])
```

etcd is... not so great

- **Problem**: unreliable watches
  - Missed events lead to unrecoverable situations
- **Problem**: limited event history
  - Can't always replay entire event stream
- **Solution**: move to “reconciler” model

Reconciler model

In a loop, run periodically until stopped:

1. Retrieve `current state` and `desired state` from datastore (e.g. etcd)
2. Calculate necessary actions to change `current state` --> `desired state`
3. Perform actions

Reconciler model

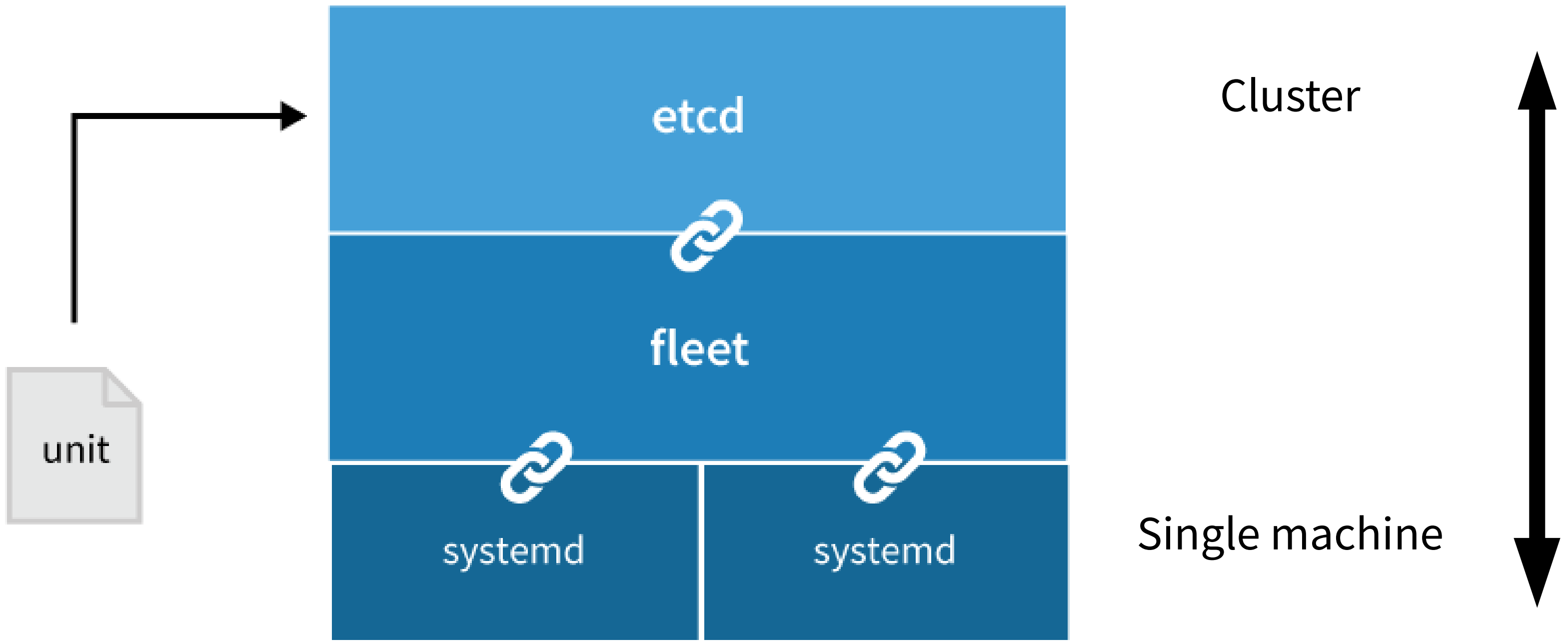
Example: fleet's engine (scheduler) looks something like:

```
for {
        select {
        case <- stopChan:
                return
        case <- time.After(5 * time.Minute):
                units = fetchUnits()
                machines = fetchMachineStates()
                reschedule(units, machines)
        }
}
```

etcd is... not so great

- **Problem**: unreliable watches
  - Missed events lead to unrecoverable situations
- **Problem**: limited event history
  - Can't always replay entire event stream
- **Solution**: move to “reconciler” model
  - Less efficient, but extremely robust
  - Still many paths for optimisation (e.g. using watches to trigger reconciliations)

fleet – high level view



fleet summed up:

- systemd is our init system
- etcd provides cluster awareness

- Or, to put it more simply:

systemd runs things
etcd coordinates things

golang

- Standard language for all CoreOS projects (above OS)
  - etcd
  - fleet
  - locksmith (semaphore for reboots during updates)
  - etcdctl, updatectl, coreos-cloudinit, ...
- fleet is ~10k LOC (and another ~10k LOC tests)

Go is great!

- Fast!
  - to write (concise syntax)
  - to compile (builds typically <1s)
  - to run tests (O(seconds), including with race detection)
  - Never underestimate power of rapid iteration
- Simple, powerful tooling
  - Built in package management, code coverage, etc.

Go is great!

- Rich standard library
  - “Batteries are included”
  - e.g.: completely self-hosted HTTP server, no need for reverse proxies or worker systems
- Static compilation into a single binary
  - Ideal for a minimal OS with no libraries

Go is… not so great

- **Problem**: managing third-party dependencies
  - modular package management but: *no versioning*
  - `import "github.com/coreos/fleet"` - which SHA?
- **Solution**: vendoring :-/
  - Copy entire source tree of dependencies into repository
  - Slowly maturing tooling: goven, third_party.go, Godep

Go is… not so great

- **Problem**: (relatively) large binary sizes
  - “relatively”, but… CoreOS is the minimal OS
  - ~10MB per binary, many tools, quickly adds up
- **Solutions**:
  - upgrading golang!
    - go1.2 to go1.3 = ~25% reduction
  - sharing the binary between tools

Sharing a binary

- client/daemon often share much of the same code
  - Encapsulate multiple tools in one binary, symlink the different command names, switch off command name
  - Example: `fleetd/fleetctl`

```
func main() {
    switch os.Args[0] {
        case “fleetctl”:
            Fleetctl()
        case “fleetd”:
            Fleetd()
    }
}
```

Before:

```
 9150032  fleetctl
 8567416  fleetd
```

After:

```
11052256  fleetctl
       8  fleetd -> fleetctl
```

Go is... not so great

- **Problem**: young language => immature libraries
  - CLI frameworks
  - godbus, go-systemd, go-etcd :-(
- **Solutions:**
  - Roll your own (e.g. `fleetctl`'s command line)
  - Keep it simple

fleetctl CLI

```
type Command struct {
        Name        string
        Summary     string
        Usage       string
        Description string
        Flags       flag.FlagSet

        Run func(args []string) int
}
```

Wrap up

- CoreOS Linux
  - Minimal OS with clustering built-in
  - Containerized applications --> a[u]tom[at]ic updates
- fleet
  - Simple, powerful cluster-level orchestration
  - Glue between system-level init system (systemd) and cluster-level awareness (etcd)
  - golang++

Questions?

References

- CoreOS updates
  https://coreos.com/using-coreos/updates/
- Omaha protocol
  https://code.google.com/p/omaha/wiki/ServerProtocol
- Raft algorithm
  http://raftconsensus.github.io/
- fleet
  https://github.com/coreos/fleet
- etcd
  https://github.com/coreos/etcd

Brief side-note: locksmith

- Reboot manager for CoreOS
- Uses a semaphore in etcd to co-ordinate reboots
- Each machine in the cluster:
    1. Downloads and applies update
    2. Takes lock in etcd (using Compare-And-Swap)
    3. Reboots and releases lock

Template units

- Easily spin up multiple near-identical units
  - Want ten copies of your web serving application running in the cluster?

```
fleetctl start webserver@{0..10}.service
```

-

Global units

- Run one instance of a unit on every machine
- Ideal for shared services: monitoring, logging, ...
- Machine-level metadata filtering
- e.g. to limit where an application can run:

```
[X-Fleet]
Global=true
MachineMetadata=location=chicago
```