sunil@mesosphere.io

Deploying Containers in Production and at Scale

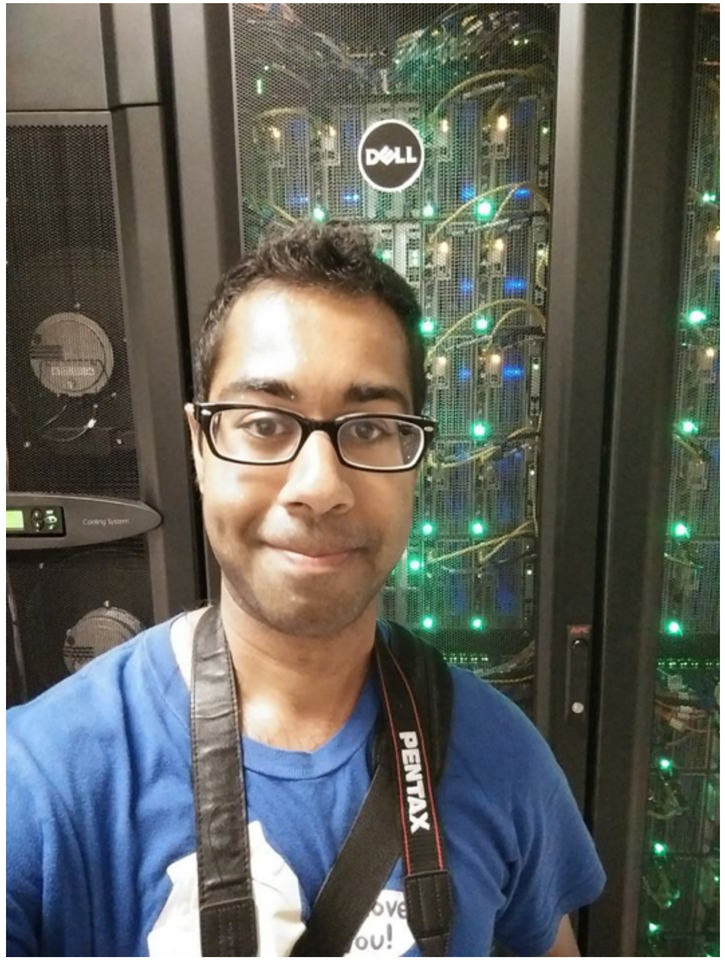
MESOSPHERE

1. Mesosphere and the DCOS

2. Running a Production Cluster: Four Themes

About Me

- Engineer at Mesosphere who wants to make life easier for our users.
- Continuing fascination with datacenters.
- Managed an 800TB cluster once upon a time, now I just talk to people with large clusters!



Dan

DATACENTER OPERATOR

- Wants happy Datacenter machines.
- Seeks to always have enough headroom.
- Prefers to avoid 3am wakeup calls.
- Aims to provide top-level services - like app deployment platforms, CI, databases - to everyone else.
- Doesn’t care about what individual workloads are actually doing: that’s for developers to worry about.

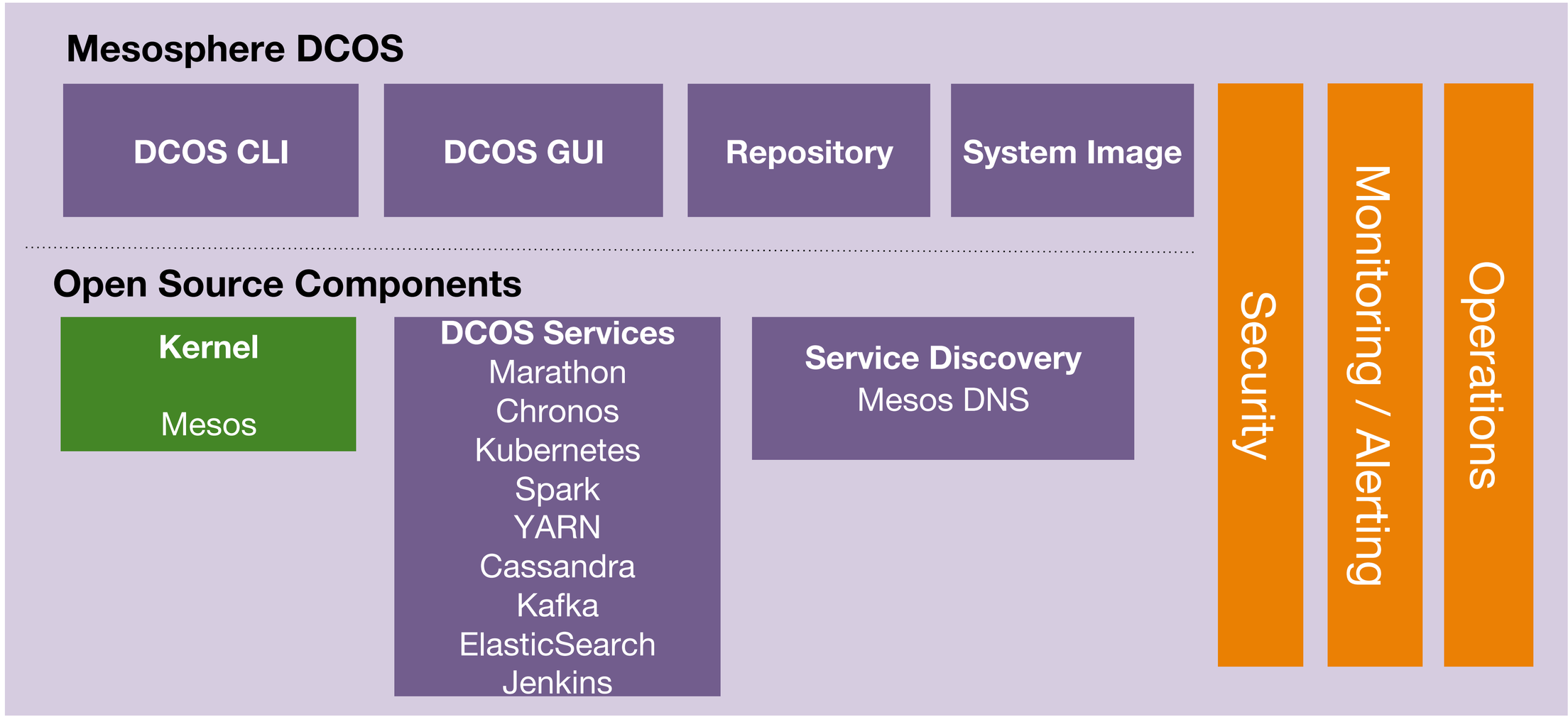
Mesosphere and the DCOS

operating system

"a collection of software that manages the computer hardware resources and provides common services for computer programs"

datacenter operating system

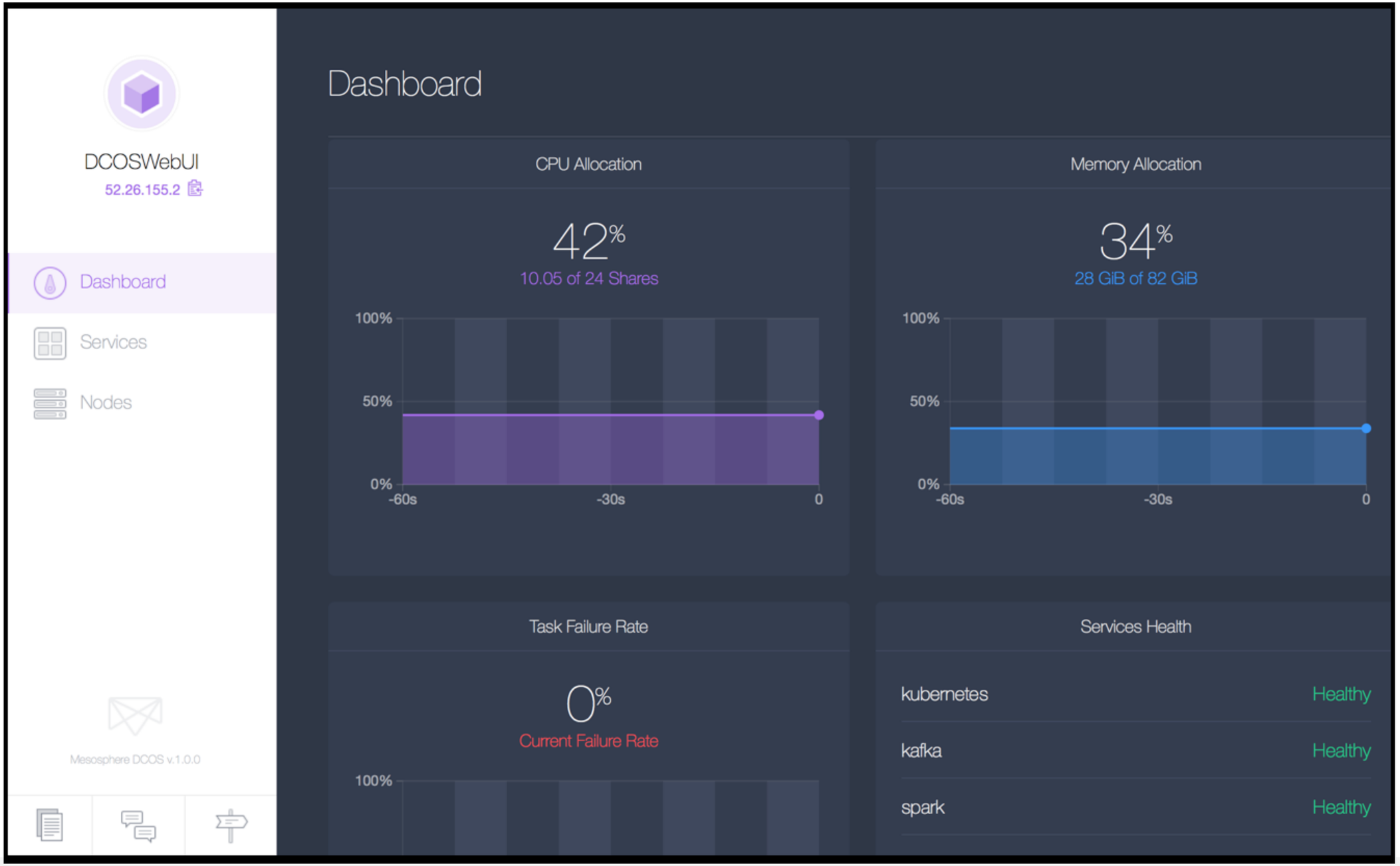
"a collection of software that manages the *datacenter* computer hardware resources and provides common services for computer programs"



Datacenter Operating System

Introduction to DCOS

- Native support for Docker containers
- Build around multiple open source projects:
  - Apache Mesos (kernel)
  - Mesosphere Marathon (init service)
  - Mesos DNS (service discovery)



The Command Line for your Datacenter

- Easiest way to install distributed systems into a cluster
- One command installs of Spark, Cassandra, HDFS, etc.
  - `dcos package install spark`
- More packages on their way!
  - Myriad (YARN scheduler)
  - ElasticSearch
- Provides tools to debug and monitor a DCOS cluster

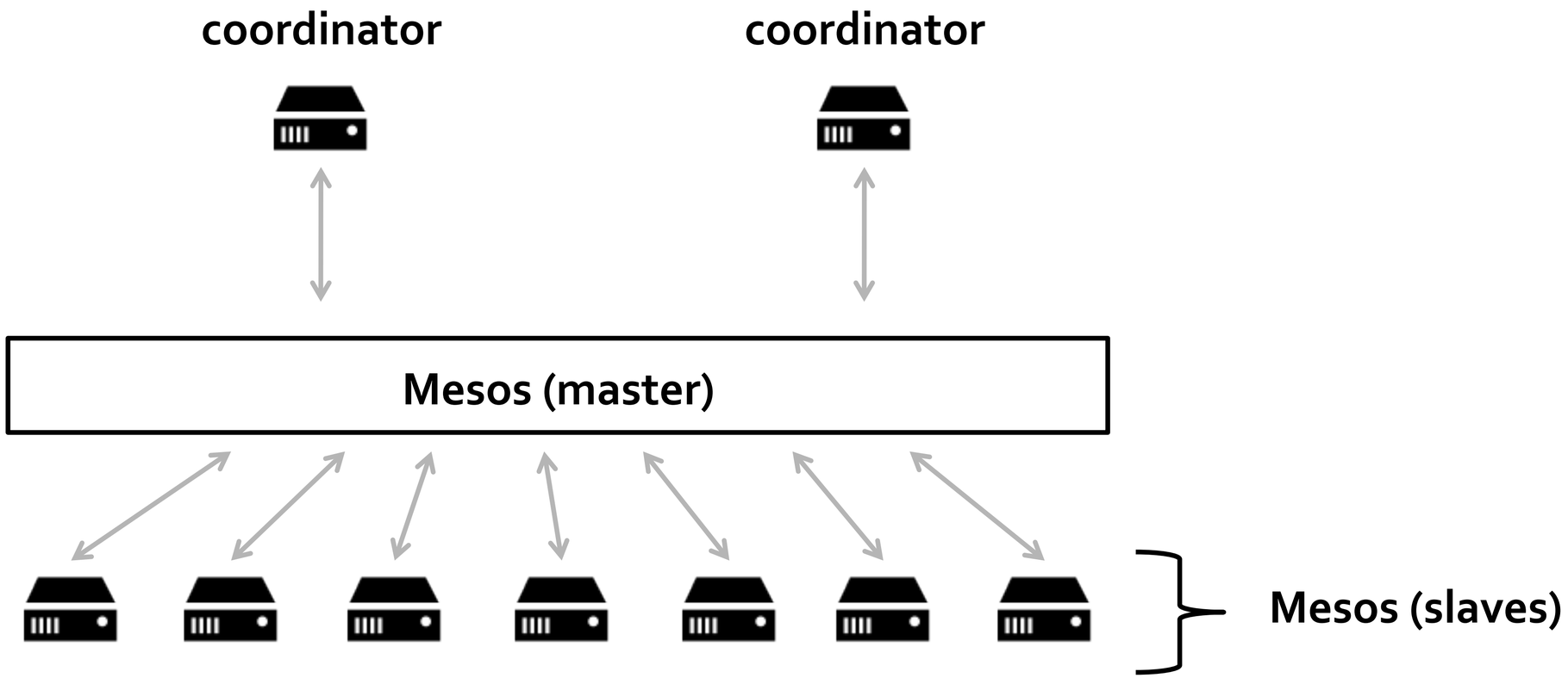
The Command Line for your Datacenter

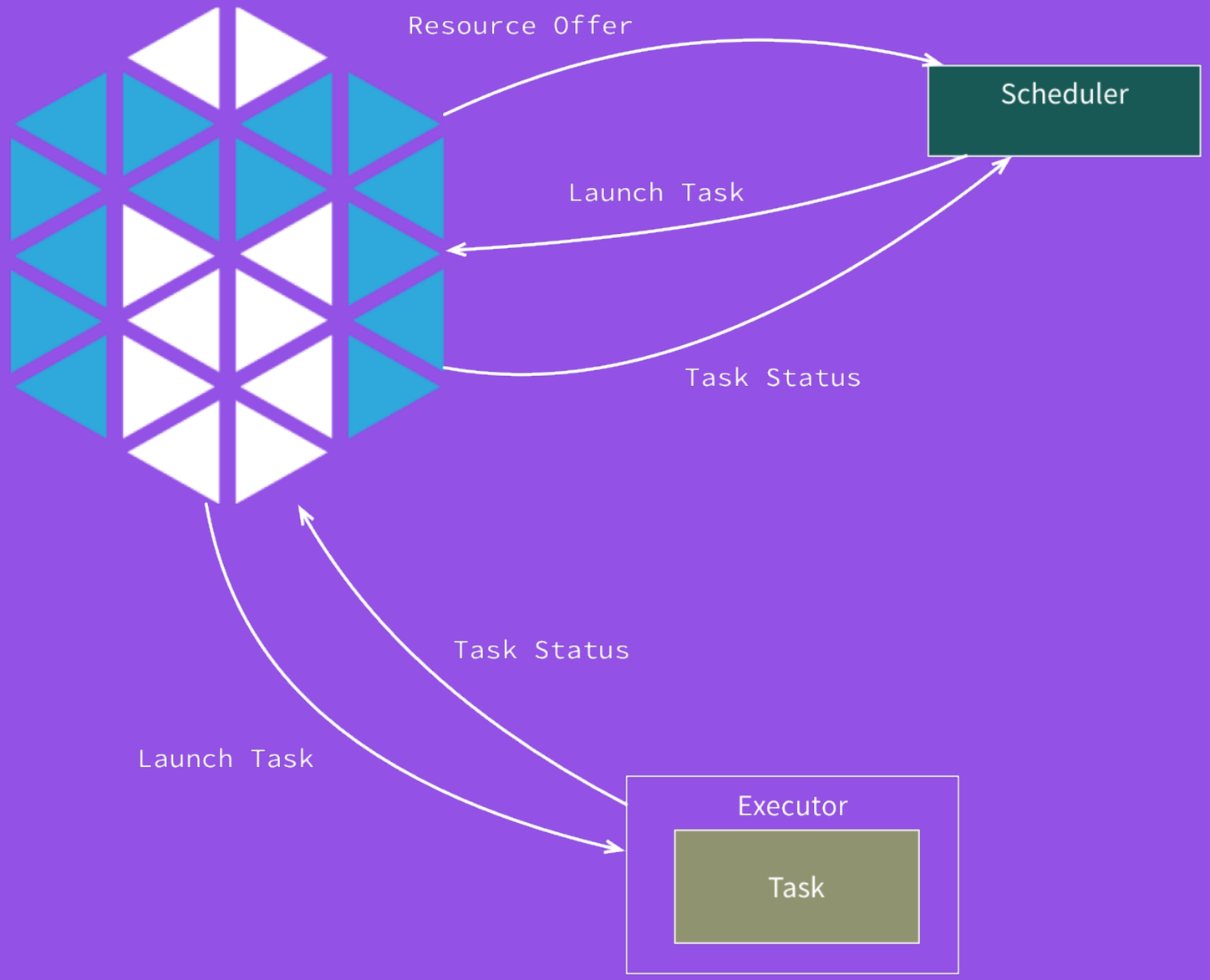
- Provides tools to debug and monitor a DCOS cluster
  - `dcos marathon app list`
  - `dcos service log spark`
- Open source (Apache 2 licensed)
- Extensible!

Apache Mesos: Datacenter Kernel


Siri: Today
Third Generation
powered by

Level of Indirection

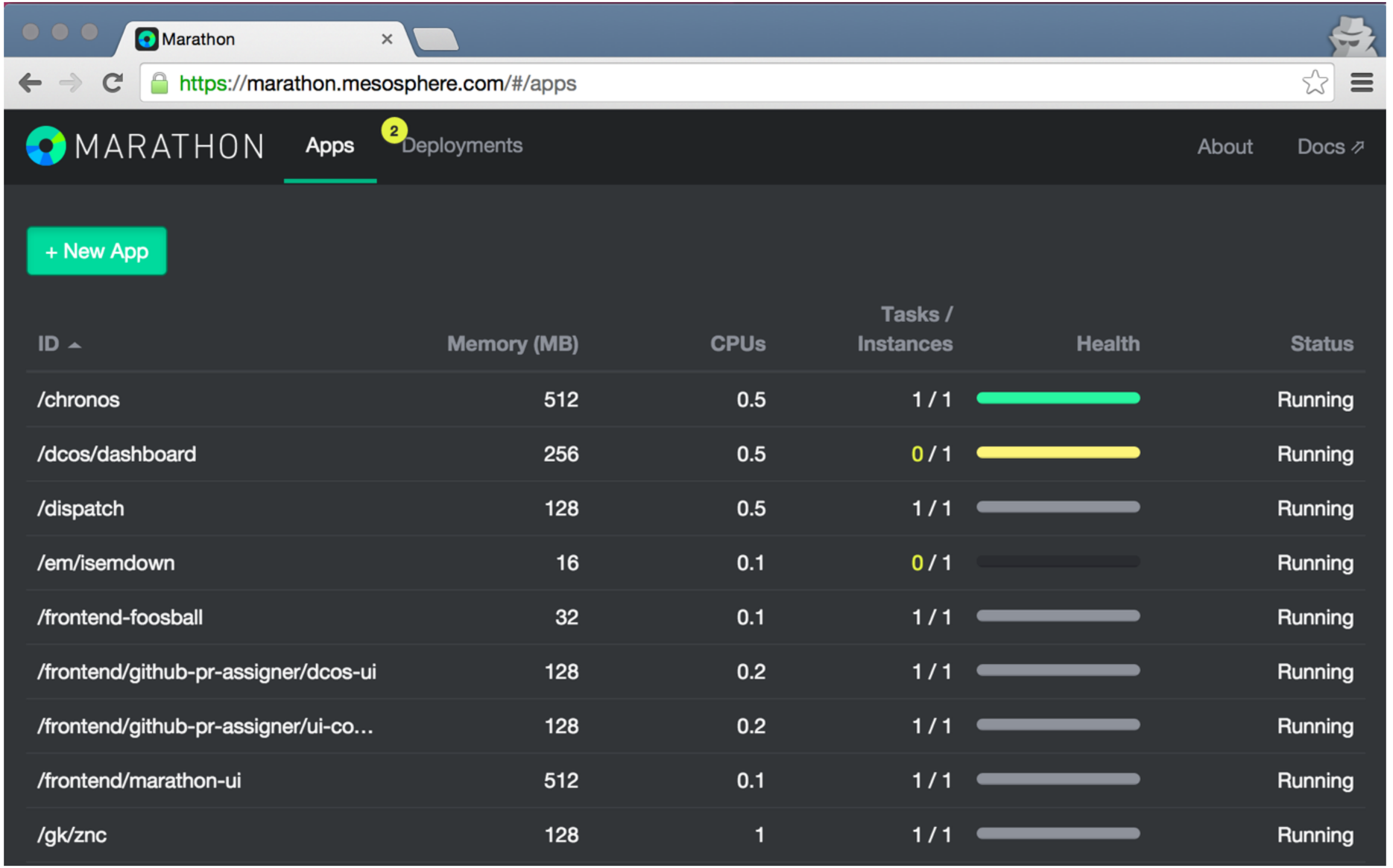


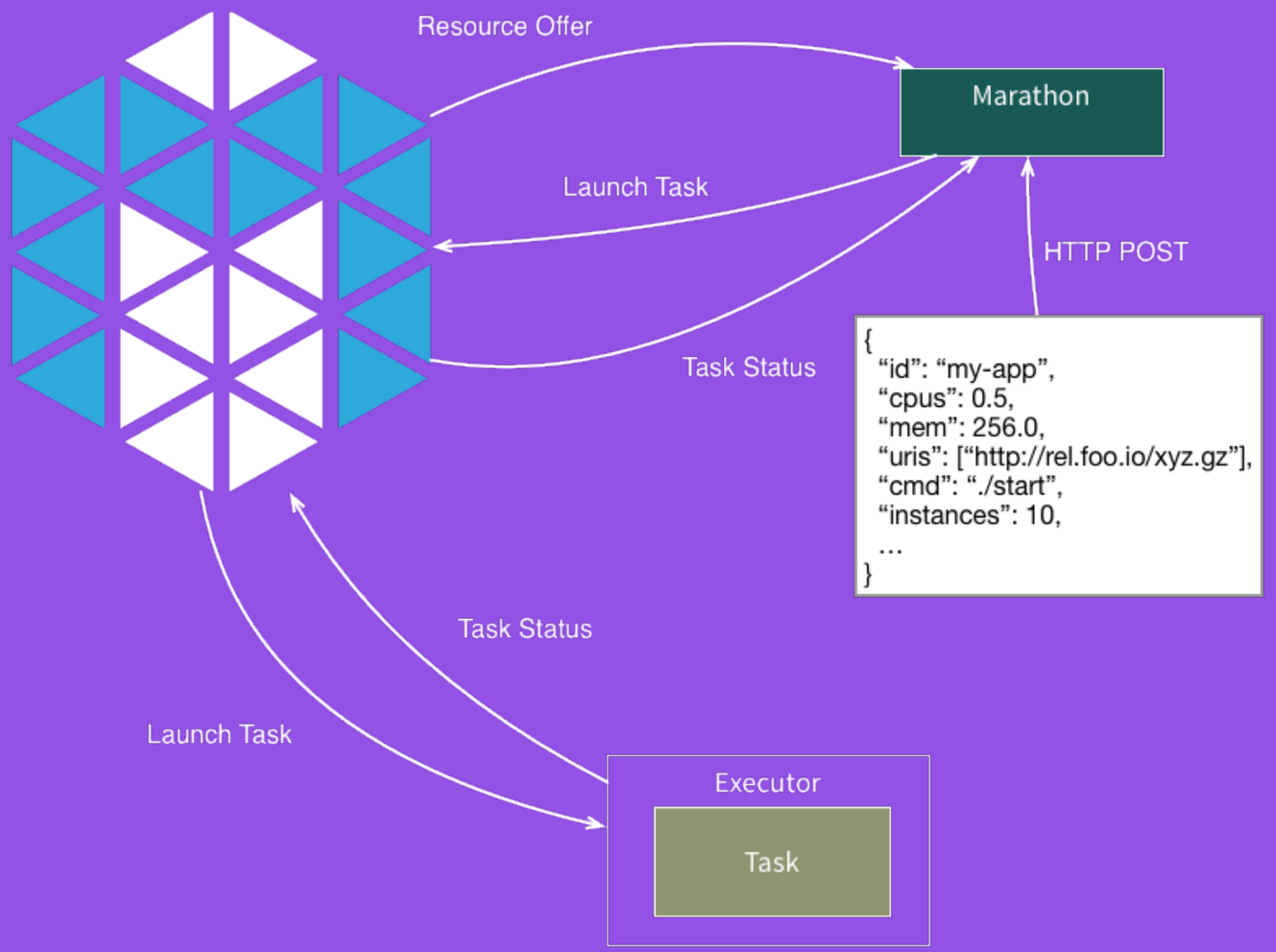


Overview & Users

- A top-level Apache project
- A cluster resource negotiator
- Scalable to 10,000s of nodes
- Fault-tolerant, battle-tested
- An SDK for distributed apps

Marathon: Init System





Features

- Start, stop, scale, update apps
- Nice web interface, API
- Highly available, no SPoF
- Native Docker support
- Rolling deploy / restart
- Application health checks
- Artifact staging

Running a Production Cluster: Four Themes

Running a Production Cluster: Four Themes

1. Dependency Management
2. Deployment
3. Service Discovery
4. Monitoring & Logging

1. Dependency Management

1. Dependency Management

a) Configuration of Servers

b) Application Dependencies

Configuration of Servers

- Still need to configure the underlying system image but it’s now much simpler!
- Use Chef or Puppet.

Use a configuration management system to build your underlying machines.

Application Dependencies

- Docker works really well!
- For non Dockerized applications, using a tarball is crude but works well.

Application developers should make no assumptions about the underlying system. Containers make this easy.

2. Deployment

2. Deployment

We need two things:

1. An artifact repository
2. A container orchestration system (i.e. Mesos)

2. Deployment

a) Developer Workflow
b) Private Registries
c) Resource Limits
d) Resource Homogeneity
e) Noisy Neighbours
f) High Availability

Developer Workflow

- Use a source control system to track application and job definitions. These can either live in a central repository or in each projects' repository.

Make use of source control and continuous integration tooling to provide an audit log of what's being deployed to your cluster.

Private Registries

- Lots of machines pulling down containers. Docker Hub just won't suffice. You'll want to use a private registry backed by something like HDFS or S3.

Run an internal registry backed by a distributed file system.

Resource Limits

- Containers need to be sized appropriately.
- Running an application on a virtual machine allows the application to grow as much as needed. Container resource limits will be enforced by killing the task.
- Some languages are better than this than others (e.g. Java)

Think harder about how much of various resources your application really needs.

Resource Homogeneity

- CPUs perform at different rates! Generally 1 core = 1 share but one core doesn’t necessarily equal another core.
- Same goes for memory!

Leave some slack in your resource limits when deploying an application to account for performance differences between servers.

Noisy Neighbours

- Just like VMs, containers suffer from the issues of noisy neighbours.
- Colocation between services is more frequent and interference becomes a really big problem. Networking isolation is still poor.
- Stanford's David Lo has done some great research into what workloads work well with each other.

Leak some slack in your resource limits when deploying an application to account for noisy neighbours. Consider co-location constraints (or machine roles) to avoid worst case interference.

High Availability

- A container based architecture will not make your applications more resilient.
- Mesos and Marathon are built to handle rolling upgrades.
- However it's up to the application itself to handle failover and persistence of state.

It's up to the application writer to build in high availability functionality. ZooKeeper is a good start.

3. Service Discovery

3. Service Discovery

Two approaches:

1. Static ports
2. Dynamic ports

Static Ports

- Each instance service is given a unique hostname and runs on the same, well known, port.
- In order to co-locate multiple instances of service on same physical host, it is necessary to allocate one IP per container.
- Typically using DNS A-records.

Less manual configuration but with static ports, unless you have one IP per container, you are limited to one instance of an application per machine.

Dynamic Ports

Routing to services running on unique ports usually requires maintaining a secondary, out-of-band, process:

1. Using a DNS server and SRV records. Application must be able to read SRV records. Most languages don't have good support for this (Go does).
2. Use a proxy or iptables that is fed by a secondary process (e.g. ServiceRouter) to remap well known ports to dynamically allocated ports.

Dynamic Ports

Applications must be written to accept ports dynamically. This may not be possible with legacy applications - which limits you to running one instance per host.

DNS based approaches work well if your applications can handle SRV records.

A combination of approaches will most likely be required.

Dynamic Ports (ZooKeeper/etc.d based)

- Use ZooKeeper or etc.d as a directory service / source of truth to store port mapping information.
- Load is significant and if clients misbehave then these services may have too many open connections.

Ensure that ZooKeeper/etc.d clients are well behaved. Stick a distributed cache in front of ZooKeeper to reduce load.

Is Not Load Balancing

- Service discovery mechanisms primarily handle reachability of one service by another and don't typically route requests in an intelligent way.

Add some intelligence to your service discovery mechanism or use an external load balancer (e.g. ELB).

4. Monitoring & Logging

4. Monitoring & Logging

Different when using containers:

1. Limited access to runtime environment
2. Metrics are different

Utilisation vs Allocation

- It's hard to size applications correctly!
- Monitor running containers for CPU and memory usage to make sure they're correctly sized.

Monitor CPU and memory of running containers to ensure applications are correctly sized.

Application Metrics

- Applications may be using all of their allocated capacity.
- This doesn't mean that they're necessarily mis-sized though.

Monitor application level metrics like throughput and latency to get a more meaningful idea of how your application is performing.

Health Checks

- Health checks allow the container management system to automatically cycle and route around tasks that may be still be running but are broken at an application level.
- Use these in combination with system/machine level monitoring to keep track of the state of a cluster.

Make health checks a mandatory part of the application deployment process.

Tooling

- It's not feasible to ssh into machines.
- Must provide tooling that allows users to introspect their containers. Mesos allows users to access their tasks' sandboxes (and the new DCOS command line interface provides similar functionality).

Make it easy for developers to access log output.

Tooling

- It's not feasible to ssh into machines.
- Must provide tooling that allows users to introspect their containers. Mesos allows users to access their tasks' sandboxes (and the new DCOS command line interface provides similar functionality).

Make it easy for developers to access log output.

Consistency FTW

- Logging becomes significantly more important to debug application failures when you're running many containers on various hosts.
- Ensure logging is approached in a standard way across applications and that log output is sufficiently descriptive to debug errors.

Mandate that applications log in a common way, either using a library or enforced best practices.

Aggregate Logs

- Good practice in general to view logging output across a cluster.
- If a machine dies, you'll lose logs.
- Aggregate these logs centrally and make them accessible to the user.

Aggregate logs and expose these to your application developers.

Summary

Summary

1. Mesos and Marathon provide a great starting point.
2. Docker with a container orchestration system makes it easier to treat machines as “cattle”.
3. Resource requirements need more thought.
4. Developers need tooling to help debug application failures.
5. No right answer (yet) for service discovery.

Thank you!

Slides will be online at:

mesosphere.github.io/presentations

Special thanks to:

- Ben Hindman
- Brenden Matthews
- Sam Eaton
- Tyler Neely