Kafka in Jail

Running Kafka in container orchestrated clusters

Sean Glover, Lightbend
@seg1o

Lightbend

Who am I?

I'm Sean Glover

- Senior Software Engineer at Lightbend
- Member of the Fast Data Platform team
- Contributor to various projects in the Kafka ecosystem including Kafka, reactive-kafka, DC/OS Commons SDK

Contact Details

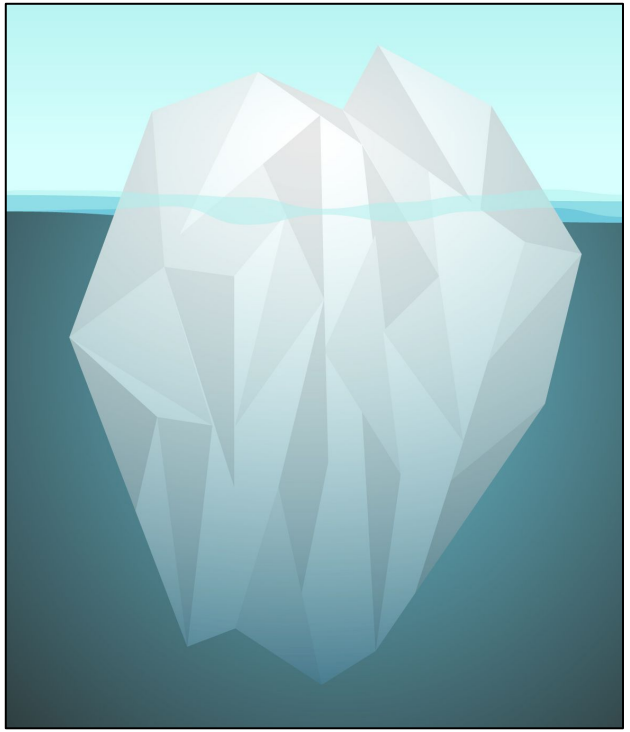
- @seg1o
- in/seanaglover
- sean.glover@lightbend.com

Operations Is Hard

“Technology will make our lives easier”

What technology makes running technology easier?

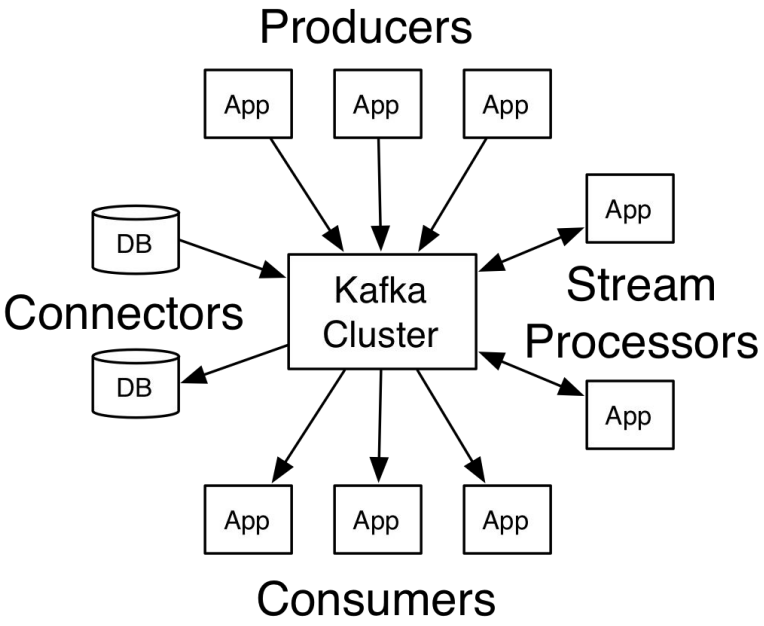
How much operations work can we automate?



Designed by Freepik

What is Kafka in one slide

- Distributed streaming platform
- Simple architecture, smart clients
- A Topic is comprised of 1 or more Partitions
- A Partition can have 0 or more Replicas
- **High Volume** - Partitions spread across cluster
- **Fault Tolerant** - Partition replicas, consistency guarantees, disk persistence
- **Fast** - Zero-copy, PageCache, Append-only



Kafka Documentation

Motivating Example: Zero-downtime Kafka Upgrade

Motivating Example: Upgrading Kafka

High level steps to upgrade Kafka

1. Rolling update to explicitly define broker properties `inter.broker.protocol.version` and `log.message.format.version`
2. Download new Kafka distribution and perform rolling upgrade 1 broker at a time
3. Rolling update to upgrade `inter.broker.protocol.version` to new version
4. Upgrade Kafka clients
5. Rolling update to upgrade `log.message.format.version` to new version

Motivating Example: Upgrading Kafka

Any update to the Kafka cluster must be performed in a serial “rolling update”. The complete Kafka upgrade process requires 3 “rolling updates”

Each broker update requires

- Secure login
- Configuration linting - Any change to a broker requires a rolling broker update
- Graceful shutdown - Send SIGINT signal to broker
- Broker initialization - Wait for Broker to join cluster and signal it’s ready

This operation is **error-prone to do manually** and **difficult to model declaratively** using generalized infrastructure automation tools.

Automation

“If it hurts, do it more frequently, and bring the pain forward.”

- Jez Humble, Continuous Delivery

Automation of Operations

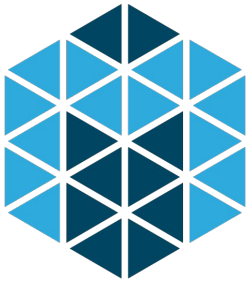
Upgrading Kafka is just one of many complex operational concerns. For example)

- Initial deployment
- Manage ZooKeeper
- Replacing brokers
- Topic partition rebalancing
- Decommissioning or adding brokers

How do we automate complex operational workflows in a reliable way?

Container Orchestrated Clusters

Cluster Resource Managers



MESOS

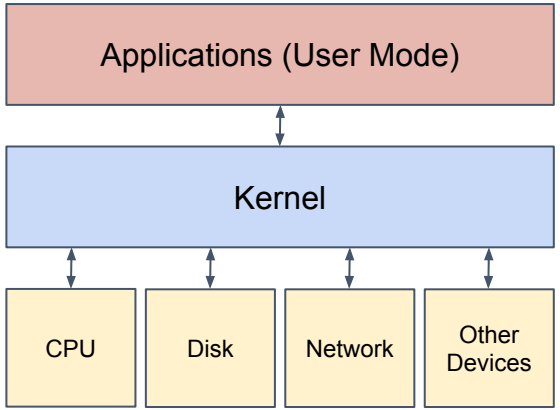
DC/OS

kubernetes

Cluster Resource Managers and Operating System

Cluster Resource Managers are similar to an Operating System Kernel

- Isolate processes and filesystem with Access Control Lists
- Share CPU cores with CPU slicing
- Prioritize (allocate CPU) with process priority (nice)
- Limit disk with filesystem quotas
- Drivers for attached devices



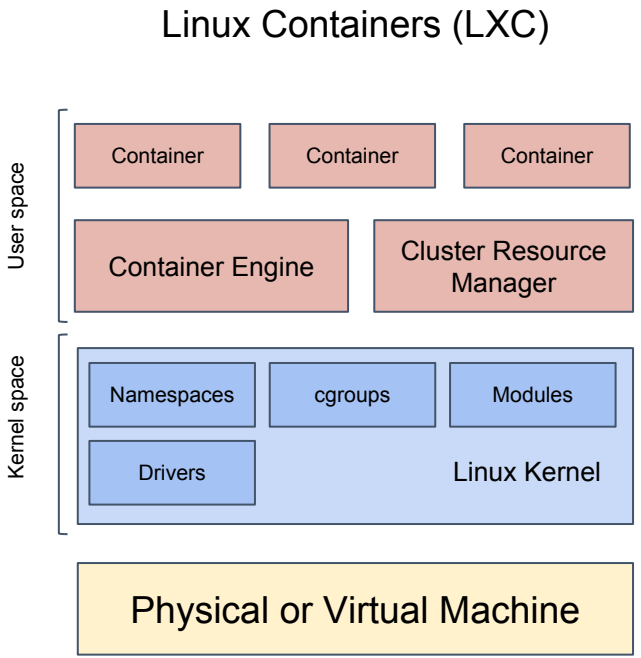
Task Isolation with Containers

- Cluster Resource Manager's use **Linux Containers** to constrain resources and provide isolation
- **cgroups** constrain resources
- **Namespaces** isolate file system/process trees
- Docker is just a project to describe and share containers efficiently
- Mesos supports Docker and Mesos containerizers
- Containers are available for several platforms

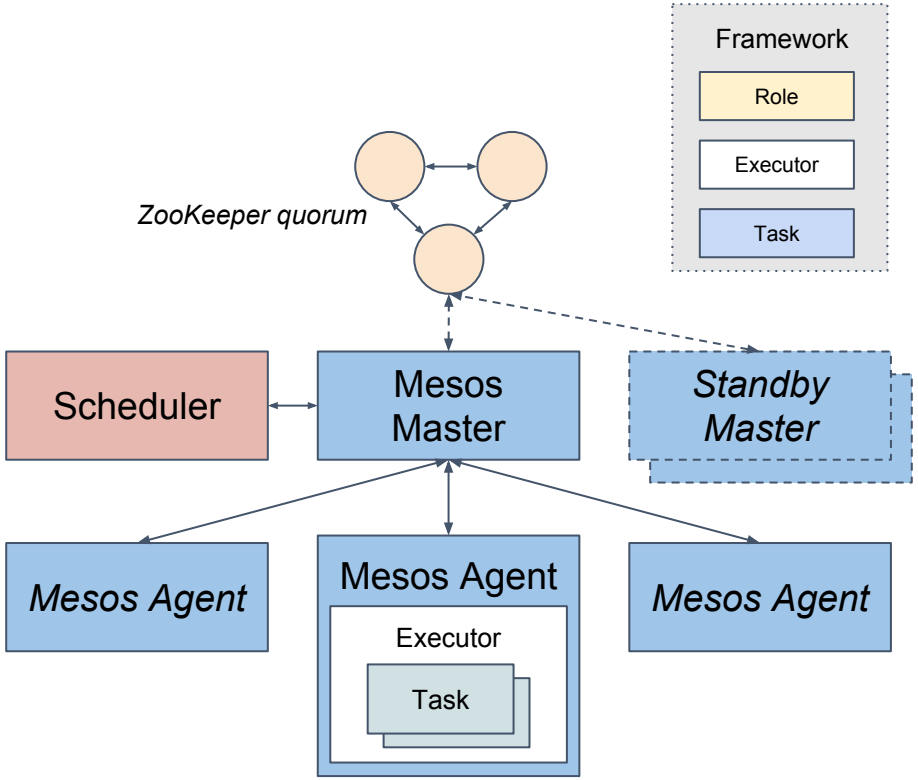
Jail

Linux Container (LXC)

Windows Container



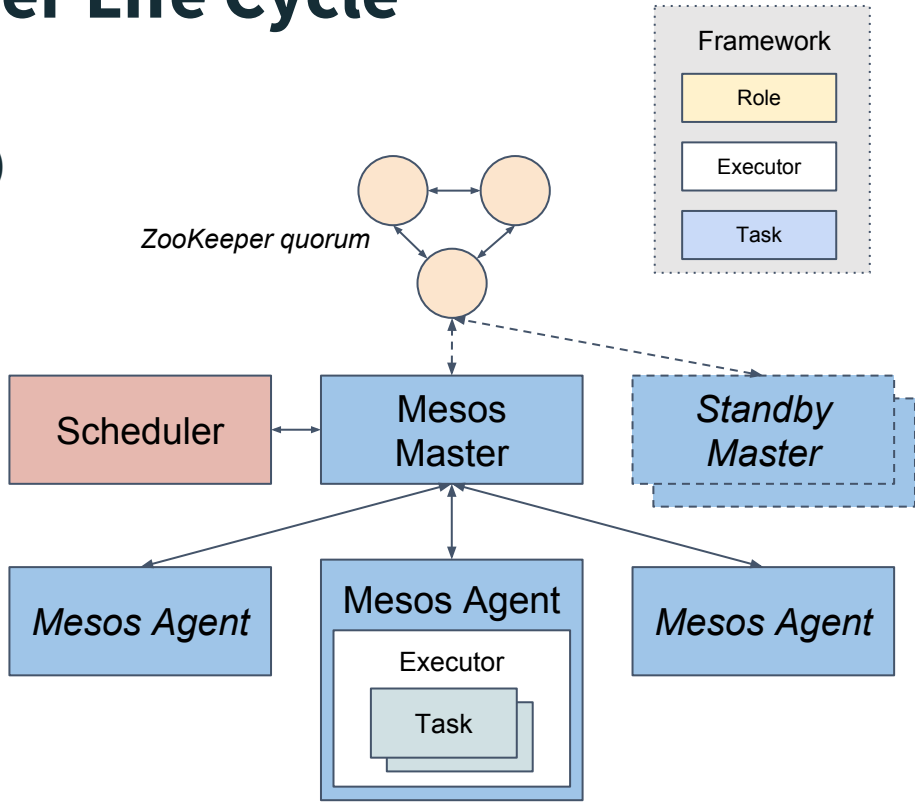
Apache Mesos Architecture



Apache Mesos Resource Offer Life Cycle

Allocating resources to an application (framework)

1. Agents inform Master of resources available
2. Scheduler makes resource request to Master for resources on behalf of Framework
3. Master responds with resource offers of agents that have available resources
4. Scheduler chooses a resource offer based on some logic and informs Master of choice
5. Master delegates schedulers response to appropriate Agent to run tasks on.



Schedulers in Apache Mesos

What are schedulers?

- Similar to system init daemons like `systemd`
- `systemd` is a system *scheduler* that keeps services running smoothly
  - Run a process on boot
  - Restart a process on failure
  - Unified admin interface
  - Gracefully stop
- A Mesos scheduler schedules tasks on behalf of an application framework
- Schedulers provide features like task restart strategy, stable identity, and scaling
- Scheduler projects for generalized workloads: Marathon, Aurora
- Custom schedulers are required for complex and app-specific workloads

The Marathon Scheduler

- Open Source scheduler by Mesosphere
- Easy way to deploy on Mesos
- Provides similar guarantees as Kubernetes resources
- Useful for most workloads, but still limits user to declarative specification

Marathon Application Definition

```
{
    "id": "basic-1",
    "cmd": "`./cool-script.sh`",
    "cpus": 0.1,
    "mem": 10.0,
    "instances": 1,
    "fetch": [
        {
          "uri": "https://example.com/app/cool-script.sh",
          "executable": true
        }
    ]
}
```

DC/OS Commons SDK

- A toolkit to help build schedulers for apps with complex operational needs
- The goal of DC/OS Commons is to make it easier to develop, package, and use services within DC/OS
  - Commons SDK Java library to build a custom scheduler (details next slide)
  - `ServiceSpec` YAML to describe service declaratively
  - Shared configuration specification with DC/OS Catalog
  - Extend DC/OS CLI with a sub command
- DC/OS Commons is used by to build production-ready frameworks to run Kafka, HDFS, ElasticSearch, and Cassandra

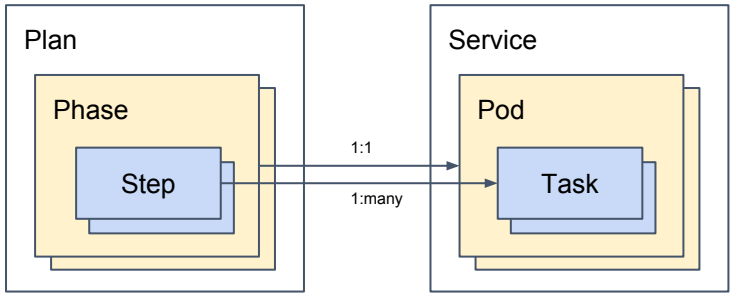
DC/OS Commons SDK Scheduler

- A YAML `ServiceSpec` is a starting point for the scheduler
- The scheduler is a **JVM application** written in Java
- Programmatically alter the `ServiceSpec` at runtime
  - Custom logic as a result of a task failing
  - Run a plan based on a runtime state change
  - Run anomaly detection and do preventative maintenance
- RESTful API for Metrics and Pod & Plan actions

DC/OS Commons SDK

The `ServiceSpec` is both a YAML specification and a JVM type hierarchy that can be used together.

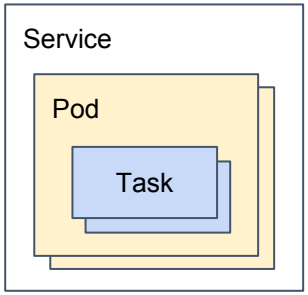
Two main components of a `ServiceSpec` are pods and plans



Although similar to Kubernetes pods DC/OS commons pods have different concerns.

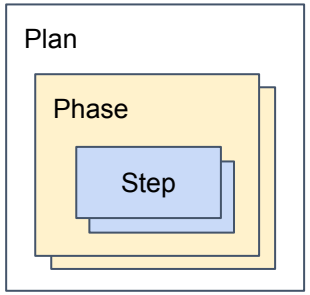
DC/OS Commons Pods

- A task represents a single operation or command
- A Goal defines the end state of the task. `RUNNING` or `ONCE`
- Resource constraints are applied with a cgroup
- **`readiness-check`**'s are conditions that must evaluate successfully to move to the goal state. `RUNNING` or `COMPLETED`
- `health-check`'s signal to the scheduler that the task is unhealthy and must be restarted




```
name: "hello-world"
pods:
  hello:
    count: 1
    tasks:
      server:
        goal: RUNNING
        cmd: "echo hello && sleep 1000"
        cpus: 0.1
        memory: 256
        health-check:
          cmd: "./check-up"
          interval: 5
          grace-period: 30
          max-consecutive-failures: 3
          delay: 0
          timeout: 10
        readiness-check:
          cmd: "./readiness-check"
          interval: 5
          delay: 0
          timeout: 10
```

DC/OS Commons Plans



- Plans define workflows of an operation (i.e. initial deployment)
- Deployment strategy is applied at phase and step levels
  - `serial` - each phase/step executed one at a time
  - `parallel` - all phase/step executed at the same time
- Plans can be automatically or manually triggered
- phases are 1:1 with a pod
- A single step is `COMPLETE` once all tasks reach their Goal
- Plan is executed from top to bottom
- Each step is defined with an ordinal value or `default` special case
- Plan is complete once all phases are `COMPLETE`

```yaml
name: "hello-world"
pods:
  hello:
    [...]
    tasks:
      init:
        [...]
      main:
        [...]
plans:
  deploy:
    strategy: serial
    phases:
      hello-phase:
        strategy: serial
        pod: hello
        steps:
          - default: [[init], [main]]
```

DC/OS Commons `ServiceSpec` Example

```yaml
name: "hello-world"
pods:
  hello:
    count: 2
    resource-sets:
      hello-resources:
        cpus: 1.0
        memory: 256
        volume:
          path: hello-data
          size: 5000
          type: ROOT
    tasks:
      init:
        goal: ONCE
        cmd: "./init"
        resource-set: hello-resources
      main:
        goal: RUNNING
        cmd: "./main"
        resource-set: hello-resources
```

...

...

```yaml
plans:
  deploy:
    strategy: serial
    phases:
      hello-phase:
        strategy: serial
        pod: hello
        steps:
          - default: [[init], [main]]
```

The DC/OS Kafka Package

DC/OS Kafka Package

Finally we can discuss the DC/OS Kafka package. The best production-ready Apache Kafka implementation you can use today on a container orchestrated cluster. *

Features include:

- One command install into an existing DC/OS cluster
- Rolling broker configuration updates
- Broker upgrades
- Broker replacement/move
- Stable broker/agent identity
- Scaling brokers up
- Virtual IP for client connections
- Topic administration
- TLS security
- Support for DC/OS metrics
- Multiple cluster installs

** The `confluent-operator` project for Kubernetes was recently announced and could change the game. At this time few details are known about its features and license model.*

DC/OS Kafka Package

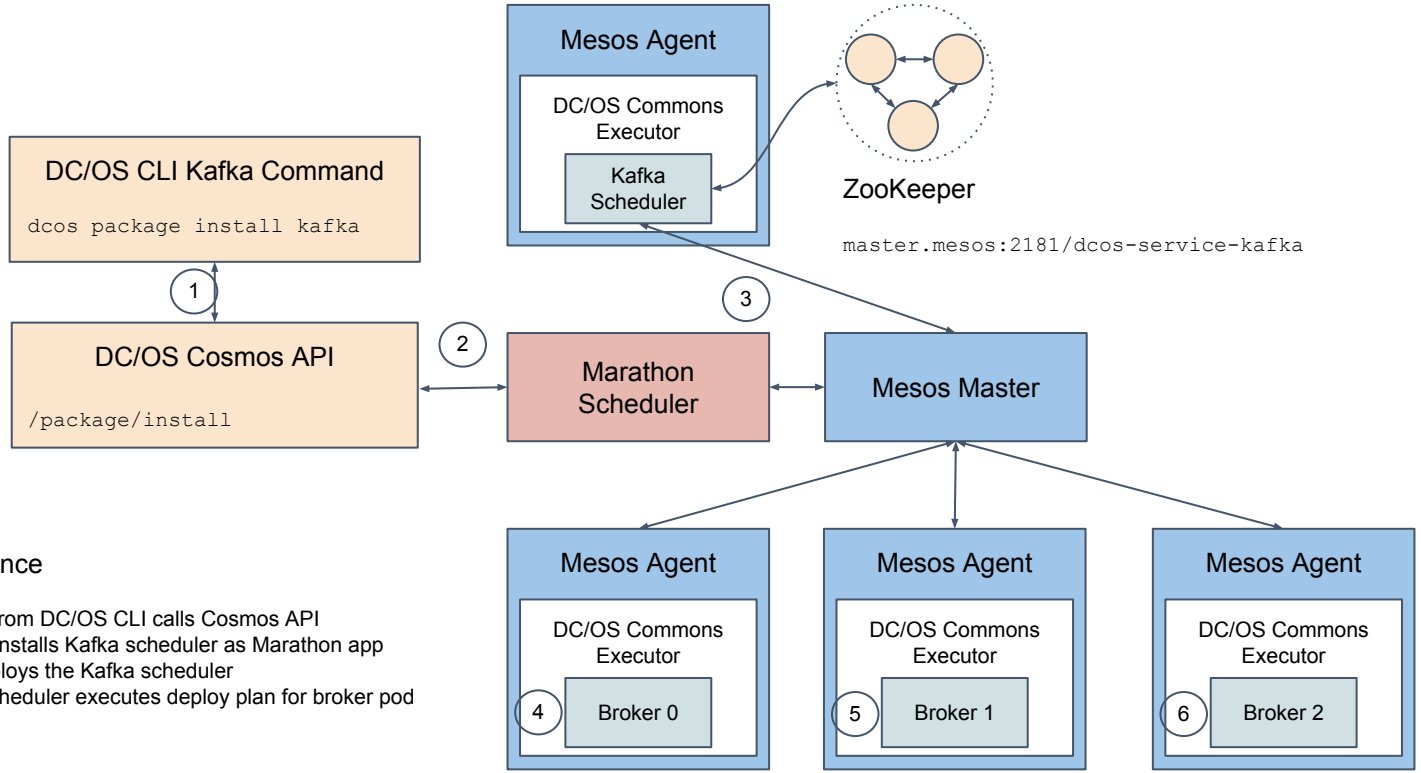
- DC/OS Kafka and Commons SDK are Open Source
- One configuration for Kafka and cluster
- Installed from DC/OS Catalog
- Most operations are triggered by a human operator using the DC/OS CLI
- Scheduler exposes API for CLI to issue commands to running cluster
- Kafka clusters can run side by side each with its own scheduler

Installing DC/OS Kafka Package

We can deploy a simple 3 broker cluster with a single command

```
$ dcos package install kafka
```

Installing DC/OS Kafka Package



Install Sequence

1. Install Kafka from DC/OS CLI calls Cosmos API
2. Cosmos API installs Kafka scheduler as Marathon app
3. Marathon deploys the Kafka scheduler
4, 5, 6. Kafka scheduler executes deploy plan for broker pod

Installing DC/OS Kafka Package

Track deploy progress

```
$ dcos kafka plan status deploy
deploy (serial strategy) (COMPLETE)
└─ broker (serial strategy) (COMPLETE)
   ├─ kafka-0:[broker] (COMPLETE)
   ├─ kafka-1:[broker] (COMPLETE)
   └─ kafka-2:[broker] (COMPLETE)
```

View broker client connection info

```
$ dcos kafka endpoints broker
{
  "address": [
    "10.0.13.209:1025",
    "10.0.5.88:1025",
    "10.0.7.216:1025"
  ],
  "dns": [
    "kafka-0-broker.kafka.autoip.dcos.thisdcos.directory:1025",
    "kafka-1-broker.kafka.autoip.dcos.thisdcos.directory:1025",
    "kafka-2-broker.kafka.autoip.dcos.thisdcos.directory:1025"
  ],
  "vip": "broker.kafka.l4lb.thisdcos.directory:9092"
}
```

DC/OS Kafka Operational Concerns

Automatic Reconciliation

A key function of the Kafka scheduler is to keep broker tasks running

- Scheduler runs as standalone task made highly available by Marathon
- Scheduler state is backed by ZooKeeper
- When a broker task stops for any reason, the scheduler will automatically restart it
  - Right away if the agent is still available
  - As soon as the agent recovers from shutdown or network partition
- Brokers have identity, maintained by scheduler
- Each broker is “sticky” to a local persistent volume which contains data logs
- Any update to configuration will trigger reconciliation process to make current state reach desired state

Rolling Configuration Updates

Updates are applied using the same deploy plan as the install.

- Package configuration is supplied and templated into `broker.properties`
- Scheduler updates each broker in serial by restarting each task one at a time with new configuration applied

Full configuration can be found from project repo, or using CLI command:

```
dcos kafka describe
```

The default `delete_topic_enable` config

```
...
"delete_topic_enable": {
  "title": "delete.topic.enable",
  "description": "Enables delete topic. Delete topic
    through the admin tool will have no effect if
    this config is turned off",
  "type": "boolean",
  "default": false
},
...
```

Enable `delete_topic_enable` with an override file `kafka-options.json`

```
{
  "kafka": {
    "delete_topic_enable": true
  }
}
```

Apply configuration changes with the DC/OS CLI

```
$ dcos kafka update start --options=kafka-options.json
```

Scaling up Brokers

Brokers can be added by making a configuration change too.

- Config not mapped to `broker.properties`, existing broker tasks are not restarted
- Config is mapped to `ServiceSpec` broker pod count
- Scheduler reconciles cluster/`ServiceSpec` by adding deploying additional brokers using `deploy` plan

The default `brokers.count` config

```
...
"count": {
  "description": "Number of brokers to run",
  "type": "number",
  "default": 3
},
...
```

Increase `brokers.count` with an override file `kafka-options.json`

```
{
  "brokers": {
    "count": 5
  }
}
```

Apply configuration changes with the DC/OS CLI

```
$ dcos kafka update start --options=kafka-options.json
```

Find new broker IPs and hostnames using endpoints command

```
$ dcos kafka endpoints broker
```

Rolling Broker Upgrades

Upgrading brokers uses the same rolling update process as previous operations.

Similar to a rolling configuration update, upgrade will do the following.

- Restart Kafka scheduler with new version
- Restart each broker task serially with new version, while maintaining identity (no data loss)

Pick a compatible package version

```
$ dcos kafka update package-versions
Current package version is: "2.2.0-1.0.0"
Package can be upgraded to: ["2.3.0-1.0.0"]
Package can be downgraded to: ["2.1.0-1.0.0"]
```

Install a new version of the Kafka subcommand on the DC/OS CLI (1-to-1 with a package version)

```
$ dcos package uninstall --cli kafka
$ dcos package install --cli kafka --package-version="2.3.0-1.0.0"
```

Initiate the upgrade process

```
$ dcos kafka update start --options=<new options json>
    --package-version="2.3.0-1.0.0"
```

To upgrade using the same workflow as in Motivating Example we perform rolling updates for `inter.broker.protocol.version` *and* `log.message.format.version` *before and after the actual broker update.*

Broker Movement

Replacing a broker is useful when its underlying agent is dead, but it can also be used to facilitate broker movement across the cluster!

- Generate a partition reassignment plan to move partitions from old broker
- Find throughput rates of partitions to move
- Pick a throttle value between average throughput rate and upper network limit
- Apply plan with a throttle to reduce network traffic
- Replace broker
- Regenerate partition reassignment plan to migrate partitions to new broker.

See current partition assignments and generate a proposed new mapping excluding broker 3

```
$ kafka-reassign-partitions --zookeeper
master.mesos:2181/dcos-service-kafka --topics-to-move-json-file
topics-to-move.json --broker-list "0,1,2,4" --generate

Current partition replica assignment
{
  "version":1,
  "partitions":[
    {"topic":"my_topic","partition":0,"replicas":[0,3]}
    ...
  ]
}
Proposed partition reassignment configuration
{
  "version":1,
  "partitions":[
    {"topic":"my_topic","partition":0,"replicas":[0,4]}
    ...
  ]
}
```

Execute the reassignment plan with a 10MB/s throttle

```
$ kafka-reassign-partitions --zookeeper
master.mesos:2181/dcos-service-kafka --reassignment-json-file
reassignment.json --execute --throttle 10000000
```

Destroy broker 3 and its identity with its agent and create it on a new instance

```
$ dcos kafka pod replace kafka-3-broker
```

Repeat steps to re-assign partitions back to new broker 3

Broker Replace & Movement

Replacing brokers is common with large busy clusters

```
$ dcos kafka pod replace kafka-3-broker
```

Broker replacement also useful to facilitate broker movement across the cluster

1. Research the max bitrate per partition for your cluster
2. Move partitions from broker to replace
3. Replace broker
4. Rebalance/move partitions to new broker

Broker Replace & Movement

1. Research the max bitrate per partition for your cluster

Run a controlled test

- Bitrate depends on message size, producer batch, and consumer fetch size
- Create a standalone cluster with 1 broker, 1 topic, and 1 partition
- Run producer and consumer perf tests using average message/client properties
- Measure broker metric for average bitrate

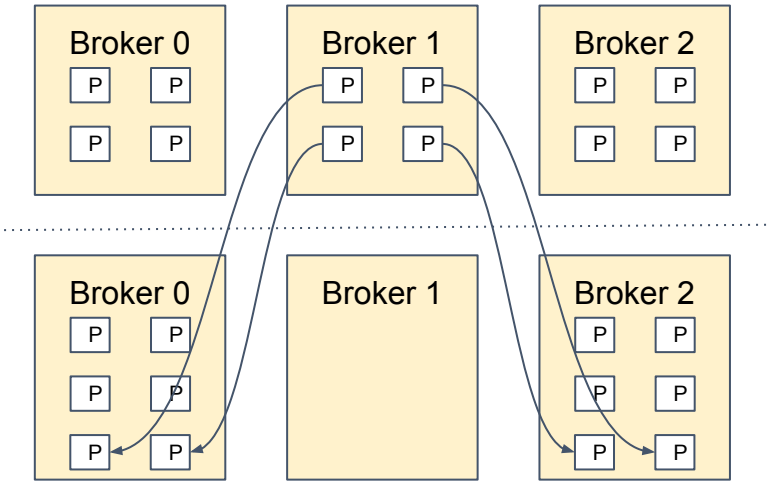
```
kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec
kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec
```

Broker Replace & Movement

2. Move partitions from broker to replace

Use Kafka partition reassignment tool

- Generate an assignment plan **without** old broker 1
- Pick a fraction of the measured max bitrate found in step 1 (Ex. 75%, 80%)
- Apply plan with bitrate throttle
- Wait till complete




```
kafka-reassign-partitions … --topics-to-move-json-file topics.json --broker-list "0,2" --generate

kafka-reassign-partitions … --reassignment-json-file reassignment.json --execute --throttle 10000000

kafka-reassign-partitions … --topics-to-move-json-file topics.json --reassignment-json-file reassignment.json --verify
```

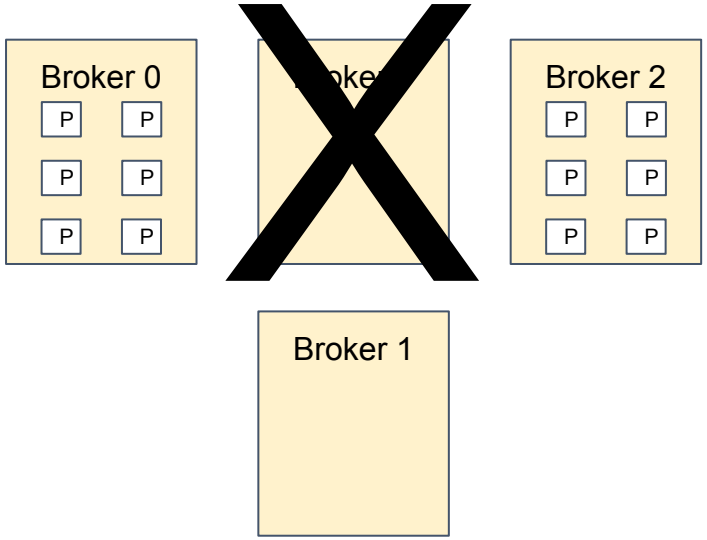
Broker Replace & Movement

3. Replace broker

Replace broker pod instance with DC/OS CLI

```
$ dcos kafka pod replace kafka-3-broker
```

- Old broker 1 instance is shutdown and resources deallocated
- Deploy plan provisions a new broker 1 instance
- New broker 1 is assigned same id as old broker 1: **1**

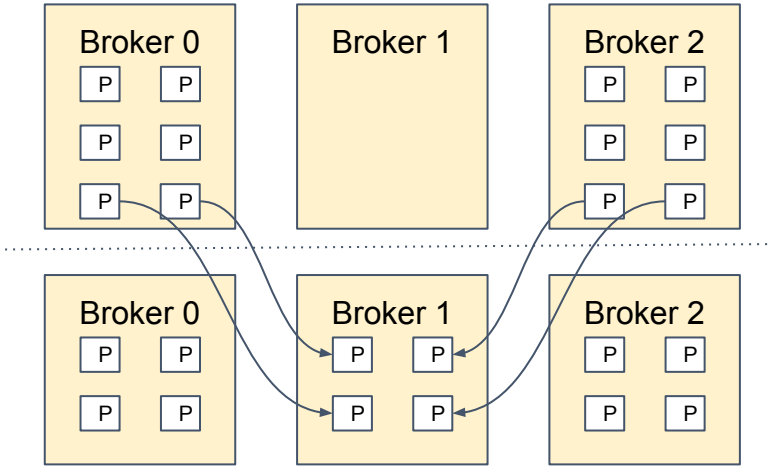


Broker Replace & Movement

4. Rebalance/move partitions to new broker

Use Kafka partition reassignment tool

- Generate an assignment plan **with** new broker 1
- Pick a fraction of the measured max bitrate found in step 1 (Ex. 75%, 80%)
- Apply plan with bitrate throttle
- Wait till complete



Managing State in Mesos

Local State in Container Orchestrated Clusters

Local state is the most difficult resource to manage in Container Orchestrated Clusters.

- Tasks are transient by default. If a task dies then its sandbox on disk is garbage collected to reclaim disk space.
- Tasks can be allocated CPU, memory, and network on any agent, but disk requires task “stickiness” to an agent. Identity.
- Using disk over a network provides the illusion of local disk at the cost of performance..
- Kafka should have the disk as close to the broker process as possible.

Apache Mesos uses Persistent Volumes for tasks that require local state.

Persistent Volumes

Persistent Volumes (PV) allow tasks to retain state across task instances.

By default the DC/OS Kafka package creates ROOT PV

- A simple directory that gets mounted to the task sandbox
- When a task fails and starts again, the directory is remounted to the new task sandbox

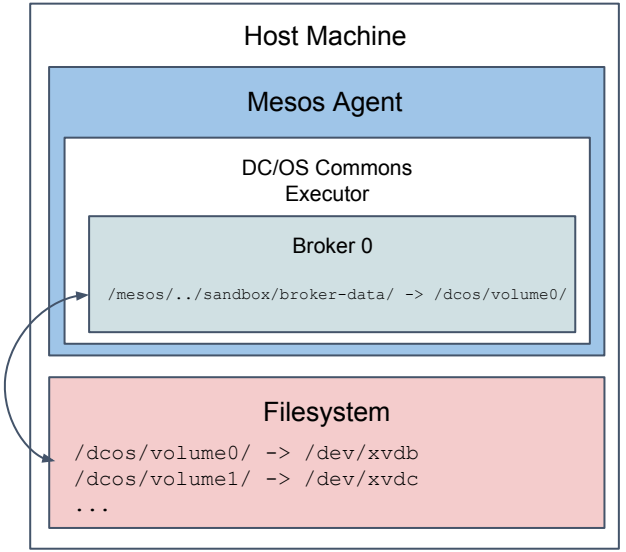
Problems with using ROOT PV

1. Directory created on a shared drive can cause contention between other tasks
2. A runaway task could fill the partition and impair any other task using it too

MOUNT Persistent Volume Type

In production MOUNT PVs should be used with DC/OS Kafka

- MOUNT PVs are treated like any other Mesos resource
- Their 2 main properties are size and the agent they're associated with.
- MOUNT PVs are made available by mounting block storage devices to the `/dcos/` directory using the `volume#` convention
- Agents will offer MOUNT's as part of the resource offer lifecycle to any task that requires one
- MOUNT's can be assigned to 1 task at a time



Kubernetes and The Operator Pattern

Kubernetes

- Kubernetes (K8s) is a popular choice for Cluster Resource Manager
- K8s is analogous to the DC/OS platform with its support for DNS/service discovery, ingress, load balancers, advanced scheduler features, and more.
- K8s has an easier to use UI (kubectl) and **faster ramp up time** than Mesos with DC/OS
- Mesos, Marathon, and Commons SDK basic DSL's overlap with one another, but K8s provides a simple well defined set of resource types.
- Basic resource types are declaratively defined as YAML. Therefore they're limited in their ability to describe advanced operational concerns.

The Operator Pattern

- The Operator Pattern is an architecture to extend Kubernetes
- K8s internally implements its features using this pattern
- Introduced by CoreOS several years ago, and now there are many operator-based projects
- It has two components
  - CustomResourceDefinition (CRD) - configuration
  - Controller application instance - operational logic
- A Controller is any application which works with the API server to satisfy the operational needs of what it is operationalizing.
- **It has the same function as a DC/OS Commons SDK-based scheduler**.
- The role of the controller is to actively reconcile the configured state from a CRD with the actual state of the service it operates

Simplified workflow of a controller performing Active Reconciliation

```
for {
  desired := getDesiredState()
  current := getCurrentState()
  makeChanges(desired, current)
}
```

strimzi

An operator-based Kafka on Kubernetes project

strimzi

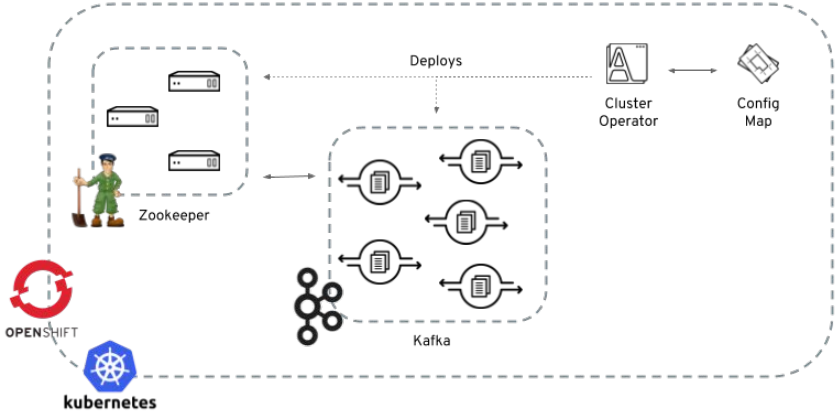
strimzi is an open source **operator-based** Apache Kafka project for Kubernetes and OpenShift

- Announced Feb 25th, 2018
- Evolved from non-operator project known as Barnabas by Paulo Patierno, RedHat
- Part of RedHat Developer Program
- Composed of two JVM-based controllers
  - Cluster Controller
  - Topic Controller

strimzi Architecture

Cluster Controller

- Uses ConfigMaps to define the cluster (cluster config, Broker config, ZK config, etc.)
- Reconciles state of ConfigMap with running state
- Deploys Topic Controller
- Supports two storage modes
  - Ephemeral (emptyDir)
  - Persistent (PersistentVolume's and Claims)
- Can host persistent KafkaConnect instances

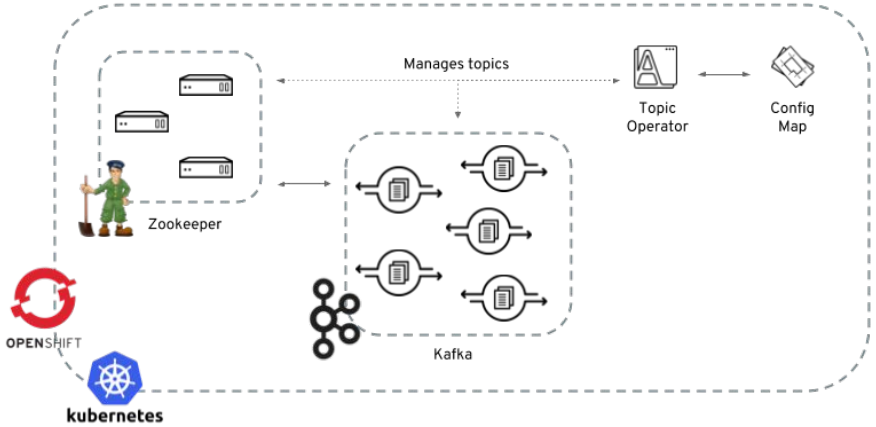


strimzi documentation http://strimzi.io/docs/0.4.0/

strimzi Architecture

Topic Controller

- Also uses ConfigMaps to manage topics on Kafka
- Bidirectional synchronization of Kafka topic state and ConfigMap state



strimzi documentation http://strimzi.io/docs/0.4.0/

strimzi

Project details

- Apache Kafka project for Kubernetes and OpenShift
- Licensed under Apache License 2.0
- Web site: http://strimzi.io/
- GitHub: https://github.com/strimzi
- Slack: strimzi.slack.com
- Mailing list: strimzi@redhat.com
- Twitter: @strimziio

Managing State in Kubernetes

`StatefulSet`'s Resource Type

- strimzi uses `StatefulSet`'s to manage Kafka Broker and ZooKeeper nodes
- `StatefulSet`'s have a number of useful properties for stateful services
  - Stable identity - Pods are "sticky" to nodes they're provisioned to
  - Stable persistent storage - a `PersistentVolume` is sticky to a pod
  - Ordered deployment and updates
  - Ordered graceful deletion and termination
  - Ordered automated rolling updates.
- The `StatefulSet` is comparable to pods defined in a DC/OS Commons-based service

PersistentVolume and Claims

- A `PersistentVolume` (PV) is very similar to a Mesos PersistentVolume
- Provides a way to abstract out block storage devices like other cluster resources
- `PersistentVolume`'s are provisioned by the Kubernetes cluster operator
- `PersistentVolumeClaim`'s are used by the user/service/application to request the assignment of a PersistentVolume
- `StorageClass`'s let you provide more parameters that can be used as criteria for what kind of `PersistentVolume` a pod requires
- `PersistentVolume`'s support a number of different block storage technologies, not just what you can mount to a local system. Ex)
  - GlusterFS
  - Ceph
  - Local filesystem
  - Cloud storage: EBS, Azure Disk, etc.

Conclusion

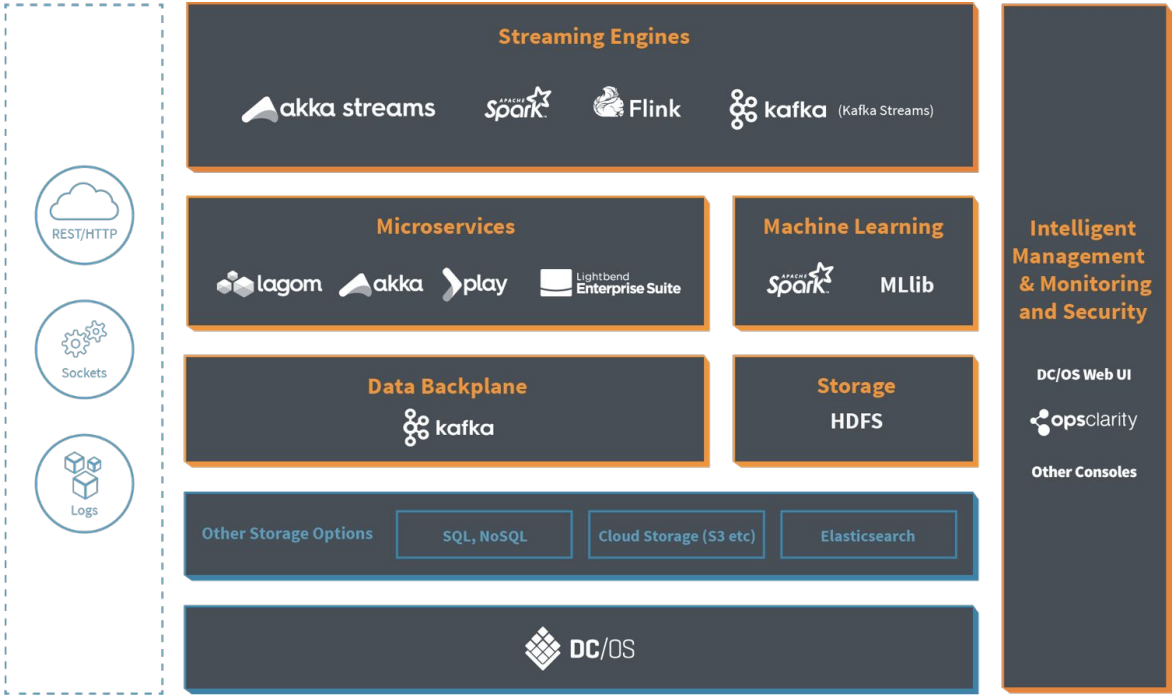
To Recap

In summary

- Containers are very useful to encapsulate any workload.
- Advanced operational needs can be satisfied with schedulers (Mesos) and controllers (K8s)
- Schedulers and Controllers let you operate “smart”/automated Kafka clusters, which lessen the burden of infrastructure or Kafka-expertise
- Movement of Kafka Brokers is possible, but requires careful planning

Putting Kafka in Jail (containers) actually liberates it!

Lightbend Fast Data Platform



http://lightbend.com/fast-data-platform

Thank You!

Sean Glover

@seg1o

in/seanaglover

sean.glover@lightbend.com