Running Hadoop on Mesos at Uber

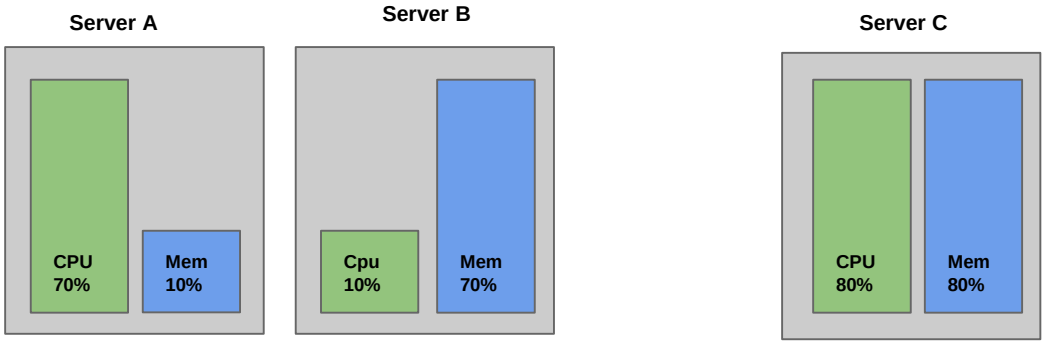
UBER

Agenda

- Problem
- Mesos
- Frameworks
- Design & Implementation
- Operating & Management
  - Multi-Framework
  - Configuration
  - Monitoring

Problem

Hardware efficiency



- In this example server A is using most of its CPU and little memory and server B is using most of its memory and little CPU.
- If we were able to colocate the services that are running on server A and B onto server C the efficiency of C would be better utilized.

Problem

Time based efficiency

- In this example cluster 1 is utilized during work hours and is mostly idle after work hours, while cluster 2 is idle during work hours and utilized after work hours.
- If we were to run both work loads on the same cluster the utilization of the cluster would be consistent over the course of the day.

Problem

Common platform between cloud and physical datacenter

- The cloud offers many conveniences that may be better than a physical datacenter.
- With Mesos we are able to have a common platform regardless of hardware.

http://mesos.apache.org/

What is Mesos?

A distributed systems kernel

Mesos is built using the same principles as the Linux kernel, only at a different level of abstraction. The Mesos kernel runs on every machine and provides applications (e.g., Hadoop, Spark, Kafka, Elastic Search) with API's for resource management and scheduling across entire datacenter and cloud environments.

Mesos only handles resource allocation so a framework is needed to schedule the use of those resources.

Apache MESOS™

Why Mesos?

Apache Mesos abstracts CPU, memory, storage, and other compute resources away from machines (physical or virtual), enabling fault-tolerant and elastic distributed systems to easily be built and run effectively.

What this means for us is we can better allocate machine resources across our fleet while preserving isolation between services. We can scale up and down a service easily and we add fault-tolerance across all our services without overhead per service.

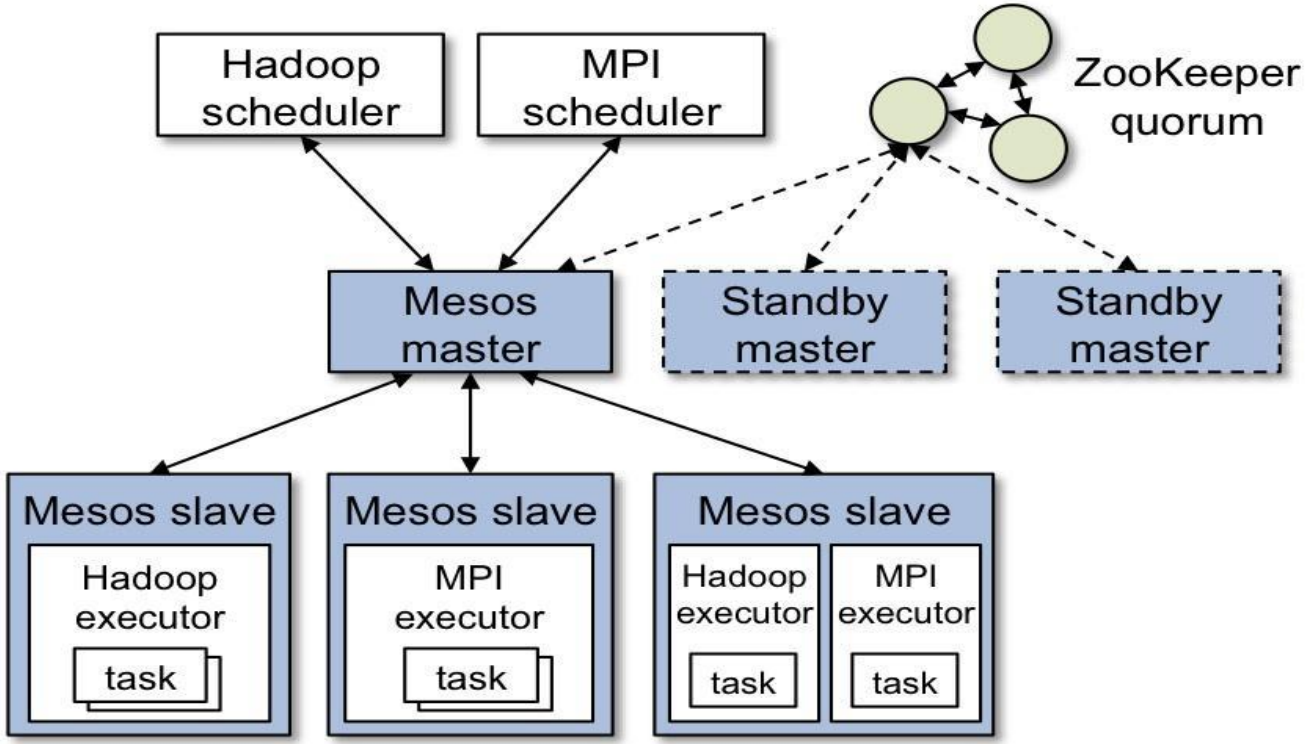
For example, if a server fails the tasks that are running on that server they can be rescheduled to another server. If a service needs more processing temporarily we can add more resources quickly then reduce them when they are no longer needed.

Apache MESOS™

Mesos supports Docker and Mesos containers for running tasks. Mesos is configurable for one container over another or you can configure mesos to try one container first and if it fails it will try the next.

Containers isolate a task from other running tasks, control task's resources, and allows tasks to run in different environments than other tasks.

Apache MESOS™



UBER

Frameworks

Scheduling and processing layer

https://open.mesosphere.com/frameworks/

Apache AURORA™

http://aurora.apache.org/

Aurora is a Mesos framework responsible for scheduling long-running services and cron jobs.

Aurora runs applications and services across a shared pool of machines, and is responsible for keeping them running forever. When machines experience failure, Aurora intelligently reschedules those jobs onto healthy machines.

Apache AURORA™

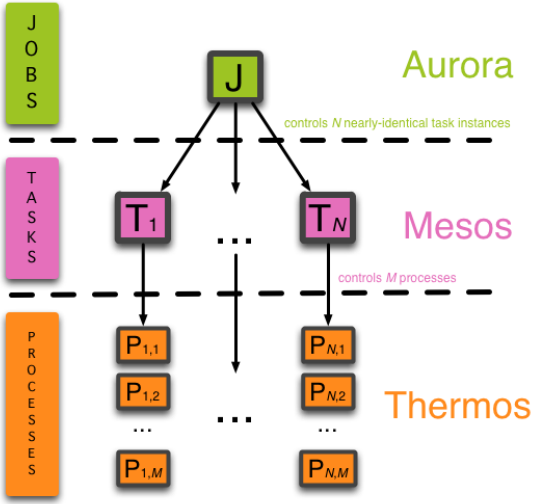
- Service jobs:
  - A service job will run forever and get automatically rescheduled if the job gets killed. Service jobs have additional metrics collected.

- Ad-hoc jobs:
  - An ad-hoc job will run once when scheduled.

- Cron jobs:
  - A cron job will run a job at a scheduled time.

Apache AURORA™

How does Aurora work?

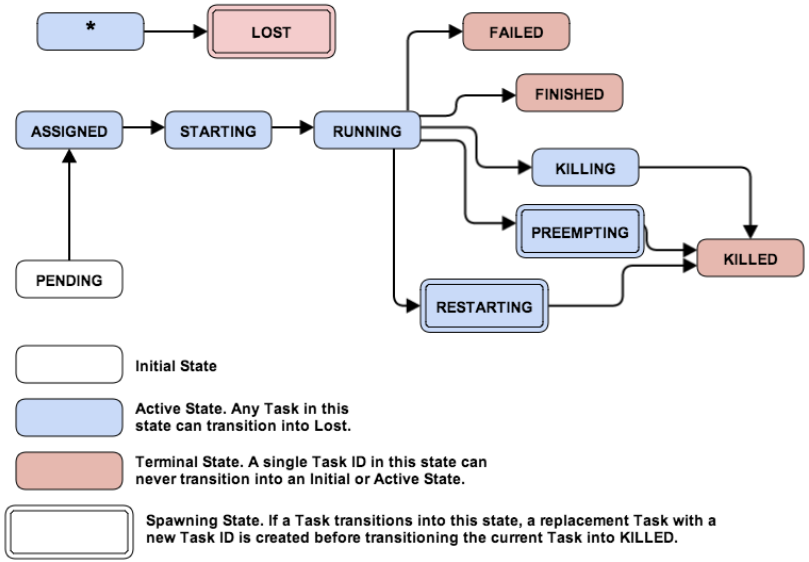


- Aurora manages jobs made of tasks.
- Mesos manages tasks made of processes.
- Thermos manages processes.
- All defined in .aurora configuration file.

UBER

Apache AURORA™

Job lifecycle

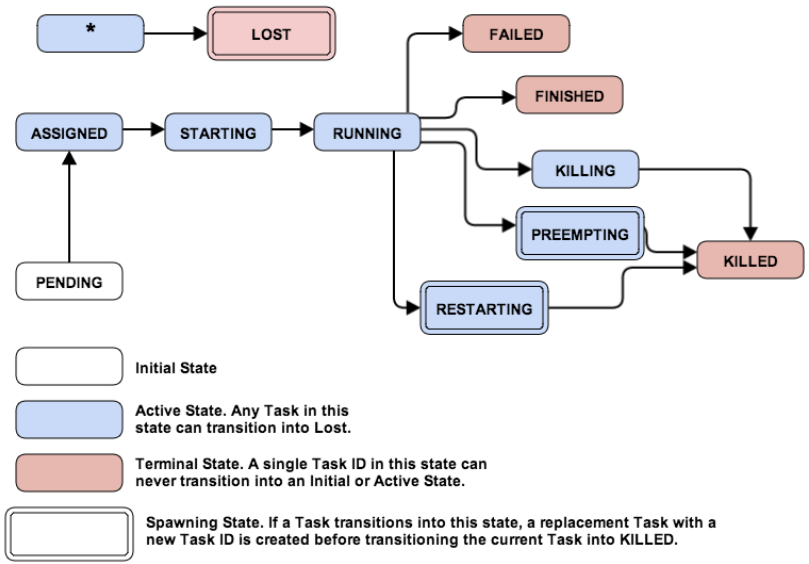


- When Aurora reads a configuration file and finds a job definition, it:
  1. Evaluates the job definition.
  2. Splits the job into its constituent tasks.
  3. Sends those tasks to the scheduler.
  4. The scheduler puts the tasks into PENDING state, starting each task's lifecycle.

UBER

Apache AURORA™

Job lifecycle

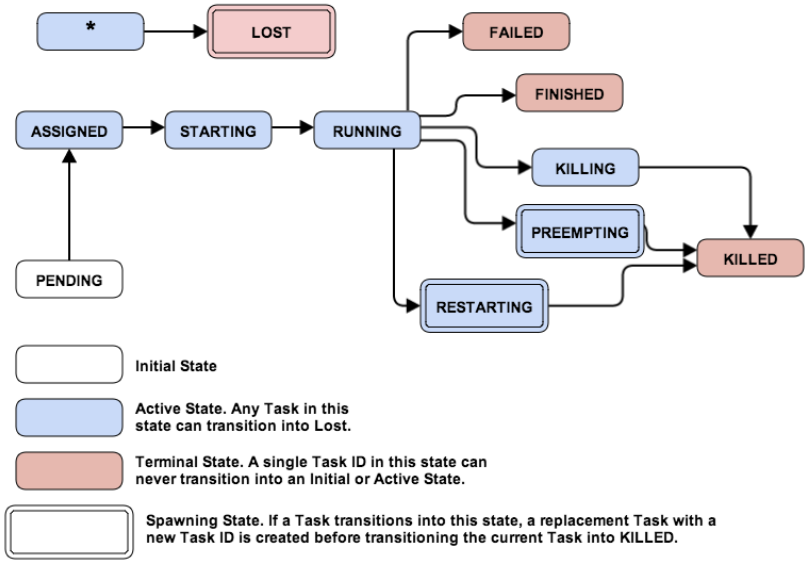


- While a task is in a PENDING state the scheduler will search for machines satisfying that task's resource requirements and configuration constraints. When it finds a match the task is ASSIGNED.
- If a task is set as production and it's stuck in the PENDING state due to lack of resources a non-production task can get preempted to free up resources for a production task. The preempted task will get moved into a PREEMPTING state then KILLED.

UBER

Apache AURORA™

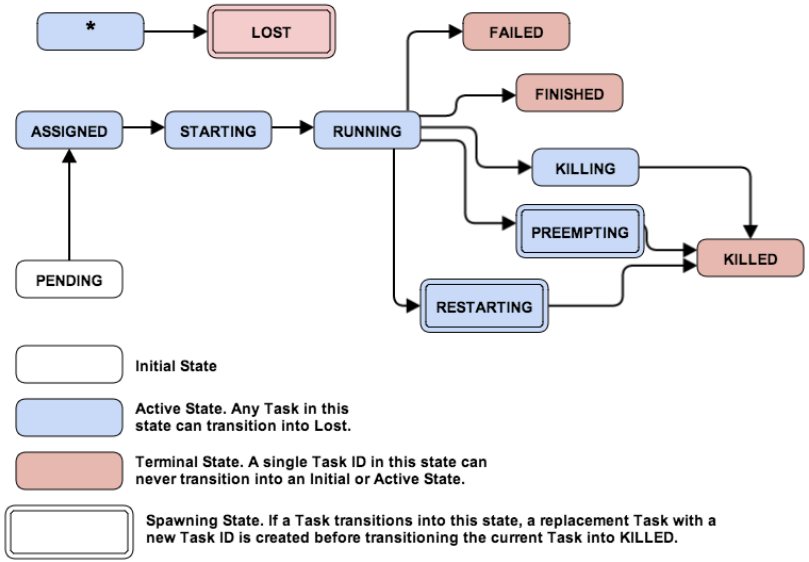
Job lifecycle



- ASSIGNED tasks are sent to the agent which spawn an executor responsible for the task's lifecycle. When the task is accepted it goes into the STARTING state.
- If a task stays ASSIGNED for too long, the scheduler forces that task into a LOST state and creates a new task in its place that's sent into PENDING state.
- Aurora keeps a special metric for this state: Mean time to assigned (MTTA) which can be useful for evaluating constraints.

Apache AURORA™

Job lifecycle

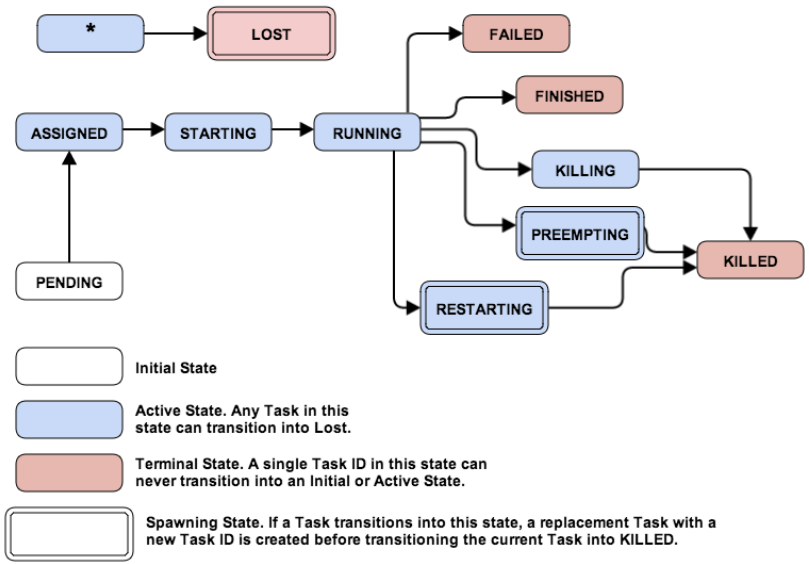


- STARTING initializes a task sandbox then Thermos begins to invoke the process and updates the scheduler putting the task into RUNNING state.
- If a task stays STARTING for too long, the scheduler forces that task into a LOST state and creates a new task in its place that's sent into PENDING state.

UBER

Apache AURORA™

Job lifecycle

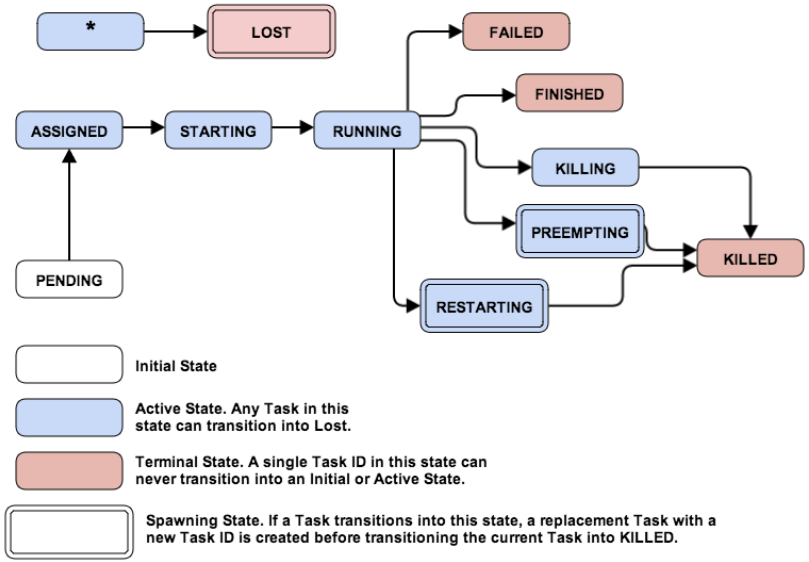


- A task says in the RUNNING state until it is naturally or forcefully terminated.
    - Natural termination occurs when a task completes: if the exit status is 0 it is FINISHED, if it's completed after reaching its failure limits it's FAILED.
    - Forceful termination occurs when a kill command is issued to a job moving it to KILLING then KILLED. It can also occur when the scheduler is forced to restart the task which puts the task in RESTARTING, launching a new task in PENDING, while taking the existing task from RESTARTING to KILLED.

UBER

Apache AURORA™

Job lifecycle



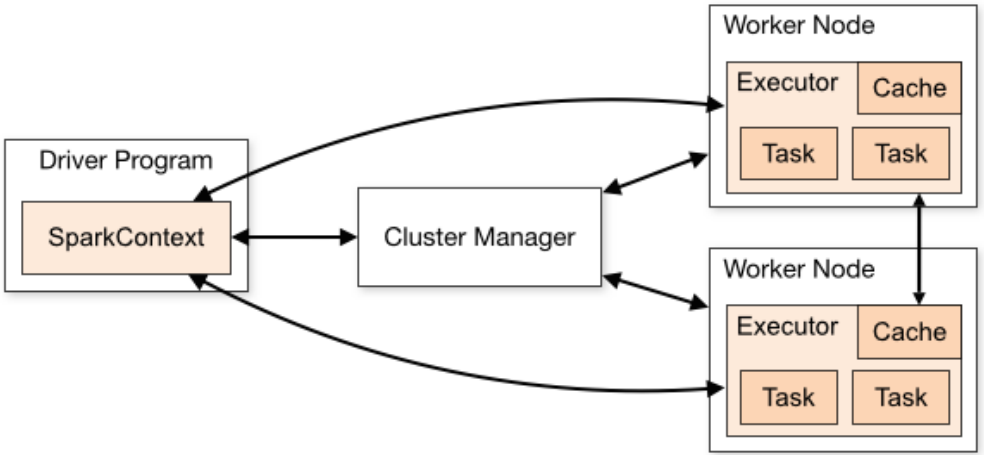
- If there is a state mismatch possibly from a network split, a state reconciliation process kills the errant RUNNING tasks.
- Aurora keeps a special metric for this state: Mean time to running (MTTR) which is the overall time to start executing user content.

http://spark.apache.org/

Apache Spark is a fast and general-purpose cluster computing system. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs. It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming.

Spark runs on Hadoop, Mesos, standalone, or in the cloud. It can access diverse data sources including HDFS, Cassandra, HBase, and S3.

Lightning-Fast Cluster Computing



Running Spark on Mesos

When using Mesos, the Mesos master replaces the Spark master as the cluster manager.

Now when a driver creates a job and starts issuing tasks for scheduling, Mesos determines what machines handle what tasks. Because it takes into account other frameworks when scheduling these many short-lived tasks, multiple frameworks can coexist on the same cluster without resorting to a static partitioning of resources.

Coarse-grained:

Will launch only one long-running Spark task on each Mesos machine, and dynamically schedule its own “mini-tasks” within it. The benefit is much lower startup overhead, but at the cost of reserving the Mesos resources for the complete duration of the application.

By default, it will acquire all cores in the cluster.

Fine-grained:

Each Spark task runs as a separate Mesos task. This allows multiple instances of Spark (and other frameworks) to share machines at a very fine granularity, where each application gets more or fewer machines as it ramps up and down, but it comes with an additional overhead in launching each task.

May be inappropriate for low-latency requirements.

http://incubator.apache.org/projects/myriad.html

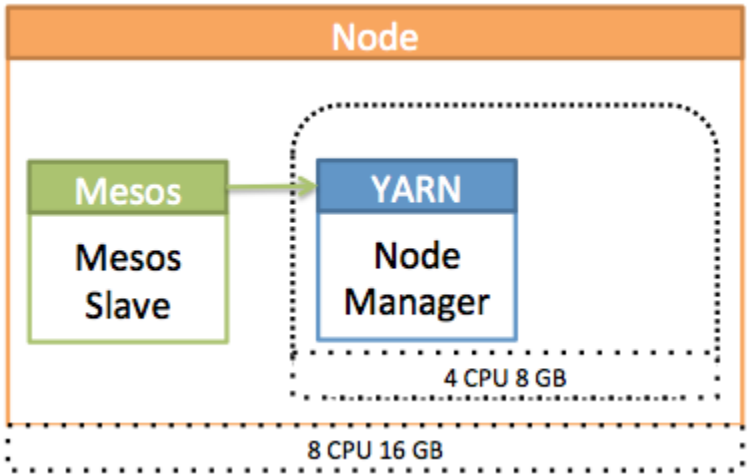
Myriad enables co-existence of Apache Hadoop YARN and Apache Mesos together on the same cluster and allows dynamic resource allocations across both Hadoop and other applications running on the same physical data center infrastructure.

Why Myriad?

With Myriad you can deploy, manage, and monitor a single cluster that supports both Docker-based microservices and Hadoop clusters deployed via Mesos frameworks.

Myriad can increase agility by provisioning multiple logical Hadoop clusters on a single physical Mesos cluster. Each logical cluster can be tailored to the end user, with a custom configuration and security policy, while running a specific version, and with either static or dynamic resources allocated to it.

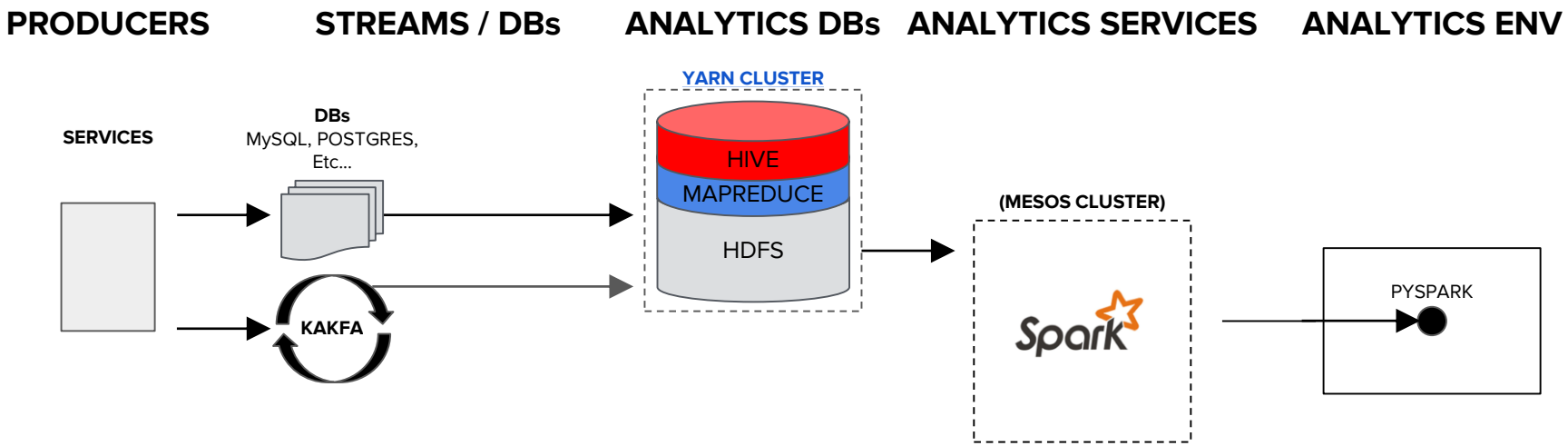
UBER



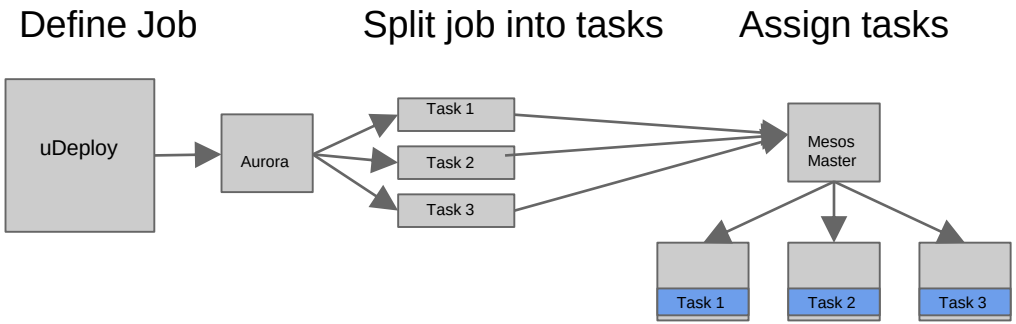
How Myriad works:

- The Mesos Slave processes advertises all of a node's resources (8 CPUs, 16 GB RAM) to the Mesos Master.
- The YARN Node Manager is started as a Mesos Task. This task is allotted (4 CPUs and 8 GB RAM) and the Node Manager is configured to only advertise 3 CPUs and 7 GB RAM.
- The Node Manager is also configured to mount the YARN containers under a Cgroup hierarchy - which stems from a Mesos task.

Spark on Mesos

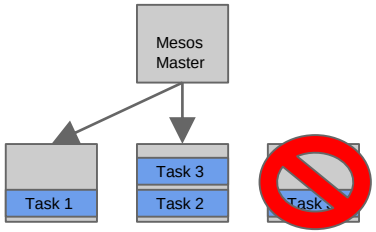


Aurora on Mesos

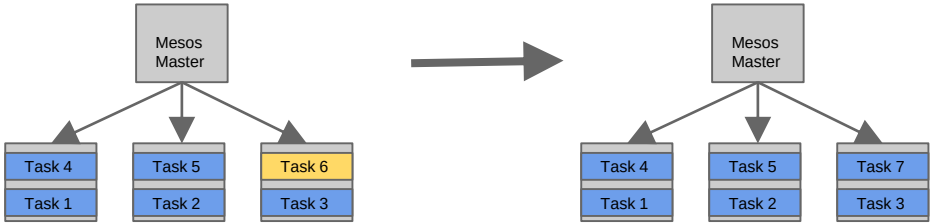


Aurora on Mesos

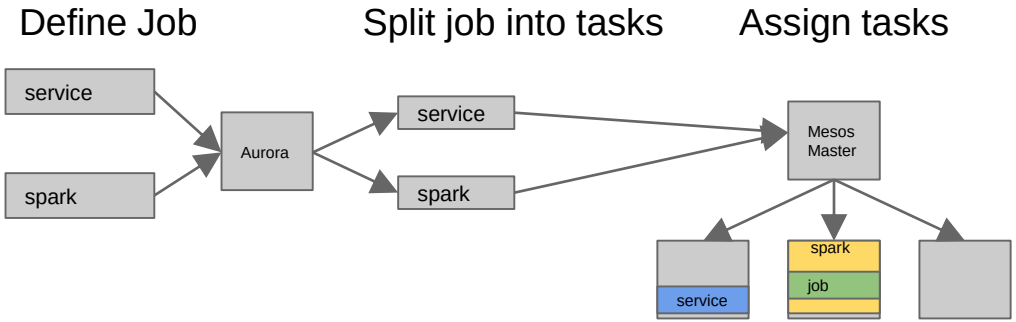
Reassign tasks



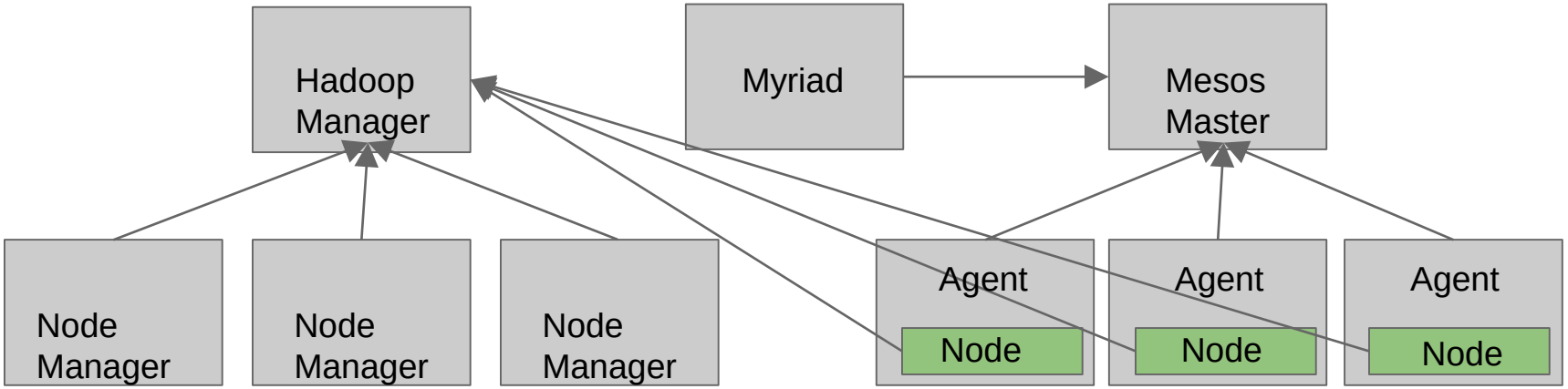
Preempting tasks



Spark & Aurora on Mesos



Using Myriad for elastic compute



Aurora job running Spark

```
 ...
environment: "environment"
jobName: "jobName"
isService: false
numCpus: 1.0
ramMb: 5120
diskMb: 5120
priority: 0
maxTaskFailures: 1
production: false
….
executorConfig:
  name: "AuroraExecutor"
 ...
container:
  mesos: not set
  docker:
    image:
"docker_repository_namename:5000/sparkie:image"
      parameters: []
 ...
instanceId: 0
status: RUNNING
failureCount: 0
```

```
…..
taskEvents: [
    Item[0] =
      timestamp: 1460736716573
      status: PENDING
      message: not set
      scheduler: "scheduler_name",
    Item[1] =
      timestamp: 1460736716595
      status: ASSIGNED
      message: not set
      scheduler: "scheduler_name",
    Item[2] =
      timestamp: 1460736725352
      status: STARTING
      message: "Initializing sandbox."
      scheduler: "scheduler_name",
    Item[3] =
      timestamp: 1460736726432
      status: RUNNING
      message: not set
      scheduler: "scheduler_name"
```

Multi-framework considerations

Static reservation:

Reserve resources on an agent for a framework
Allows fine grain configuration
Hard to manage at large scale

Dynamic reservation:

Agent offers cpu(*): 12
Offers goes from Master to Framework
Framework reserve resource “cpu(role):6” to Master
Master sends reservation to Agent
Agent returns offer of cpu(*):6, cpu(role):6
Those offers are given to the Frameworks

UBER

Quota:

A quota is sent for a role on Mesos for the whole cluster

Example:
aurora-role: 1000 cpu and 1000 gb

Better for large clusters
Can use more than the guaranteed amount
Future will have quota limit as well

Oversubscribing resources

Agent offers cpu(*):12 cpu(revocable, *):6
Agent is now offering 18 cpu instead of 12

Tasks using revocable CPU will be killed to make sure non-revocable tasks have resources.

Can also add revocable for a framework:
Agent offers cpu(*):12 cpu(revocable, role): 6

Configuration via puppet

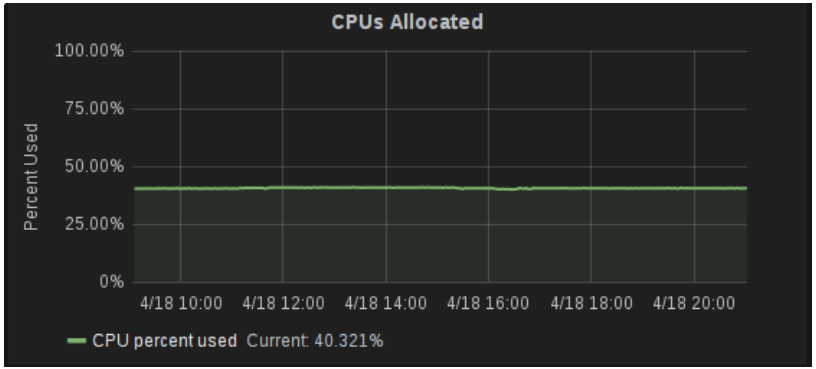
```
ensure_resource(
      'file',
      '/etc/mesos/zk',
      {
        content => $zookeeper_urls,
        require => Package['mesos'],
      }
    )

 '/etc/mesos-master/quorum':
      ensure => $file_ensure,
      content => strip(" ${quorum} "),
      require => Package['mesos'],
      before => Uber::System::Service['mesos-master'];

 '/etc/mesos-slave/containerizers':
      ensure => $file_ensure,
      content => join($containerizers, ','),
      require => $containerizer_resources,
      before => Uber::System::Service['mesos-slave'];
```

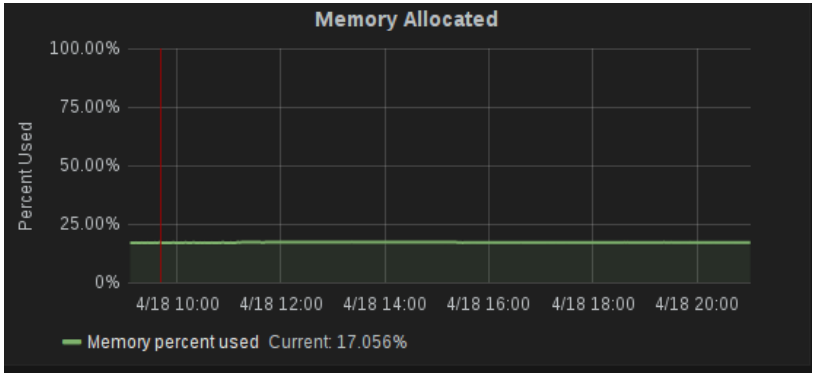
We pass file contents as command line arguments to either the Master or Agent based one which directory the file is in.

- /etc/mesos:
  - Both Master and Agent
- /etc/mesos-master:
  - Master only
- /etc/mesos-slave:
  - Agent only

Using puppet we can dynamically control these files making configuration easier.

Cluster monitoring

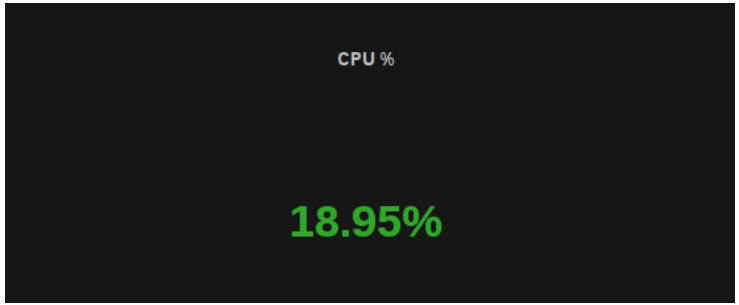


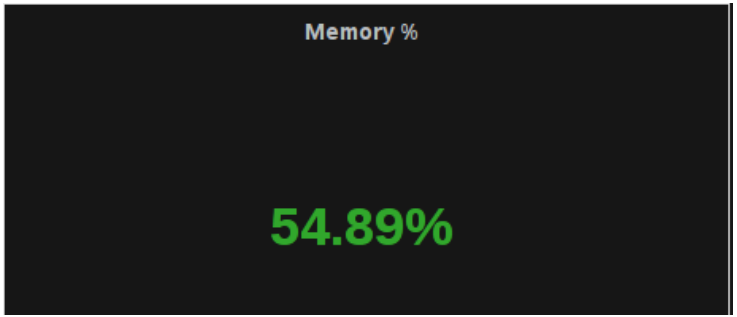


Cluster level monitoring is aggregated from all the agents in the cluster.

The cluster reports how many resources are allocated not used.

Container monitoring





Container level monitoring shows how many resources are used of the amount allocated.

This is a good way to see if the container is over allocated.

Questions?

Thank you!