

# Cluster Schedulers

Pietro Michiardi

Eurecom

# Introduction and Preliminaries

# Overview of the Lecture

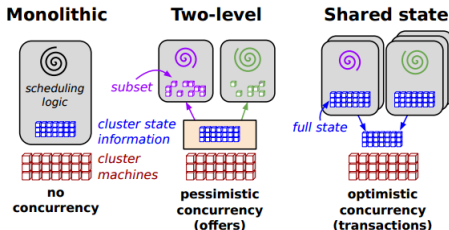
- **General overview of cluster scheduling principles**
  - ▶ General objectives
  - ▶ A taxonomy
  - ▶ Current architectures
- **In depth presentation of three representative examples**
  - ▶ Yarn
  - ▶ Mesos
  - ▶ Borg (Kubernetes)

# Cluster Scheduling Principles

## Objectives

- **Large-scale clusters are expensive, so it is important to use them well**
  - ▶ Cluster utilization and efficiency are key indicators for good resource management and scheduling decisions
  - ▶ Translates directly to cost arguments: better scheduling → smaller clusters
- **Multiplexing to the rescue**
  - ▶ Multiple, heterogeneous mixes of application run concurrently
  - The scheduling problem is a challenge
- **Scalability bottlenecks**
  - ▶ Cluster and workload sizes keep growing
  - ▶ Scheduling complexity roughly proportional to cluster size
  - Schedulers must be scalable

## Current Scheduler Architectures

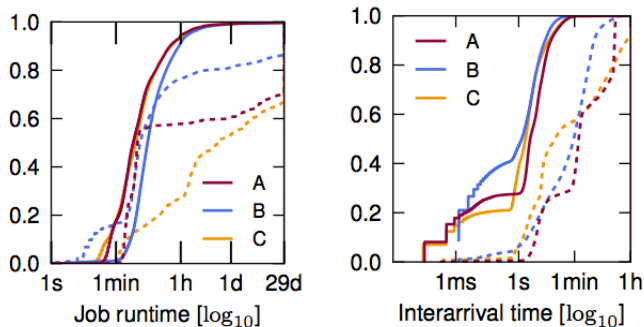


- **Monolithic:** use a centralized scheduling and resource management algorithm for all jobs
  - ▶ Difficult to add new scheduling policies
  - ▶ Do not scale well to large cluster sizes
- **Two-level:** single resource manager that grants resources to independent “framework schedulers”
  - ▶ Flexibility in accommodating multiple application frameworks
  - ▶ Resource management and locking are conservative, which can hurt cluster utilization and performance

## Typical workloads to support

- **Cluster scheduler must support heterogeneous workloads and clusters**
  - ▶ Clusters are made of several generations of machines
  - ▶ Workloads evolve in time, and can be made of a variety of applications
- **Rough categorization of job types**
  - ▶ Batch jobs: e.g., MapReduce computations
  - ▶ Service jobs: e.g., end-user facing web service
- **Knowing your workload is fundamental!**
  - ▶ Next, some examples from real cluster traces
  - ▶ The rationale: measurements can inform scheduling design

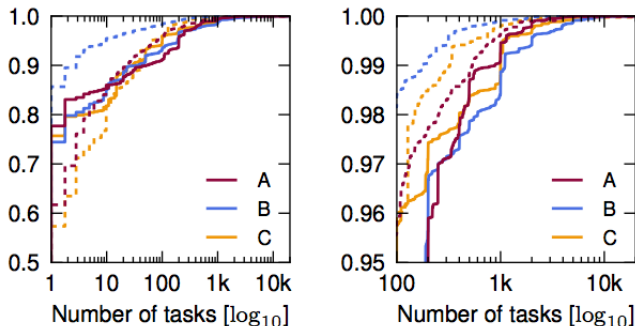
## Real cluster trace: workload characteristics



- Solid lines: batch jobs, dashed lines: service jobs



## Real cluster trace: workload characteristics



- Solid lines: batch jobs, dashed lines: service jobs

## Short Taxonomy of Scheduling Design Issues

- **Scheduling Work Partitioning:** how to distribute work across frameworks
  - ▶ Workload-oblivious load-balancing
  - ▶ Workload partitioning and specialized schedulers
  - ▶ Hybrid
- **Resource choice:** which cluster resources are available to concurrent frameworks
  - ▶ All resources available
  - ▶ A subset of cluster resources is granted (or offered)
  - ▶ NOTE: preemption primitives help scheduling flexibility at the cost of potentially wasting work

## Short Taxonomy of Scheduling Design Issues

- **Interference:** what to do when multiple frameworks attempt at using the same resources
  - ▶ Pessimistic concurrency control: make sure to avoid conflicts, by partitioning resources across frameworks
  - ▶ Optimistic concurrency control: hope for the best, otherwise detect and undo conflicting claims
- **Allocation Granularity:** task “scheduling” policies
  - ▶ Atomic, all-or-nothing gang scheduling: e.g. MPI
  - ▶ Incremental placement, hoarding: e.g. MapReduce
- **Cluster-wide behavior:** some requirements need global view
  - ▶ Fairness across frameworks
  - ▶ Global notion of priority

## Summary of design knobs

<i>Approach</i>	<i>Resource choice</i>	<i>Interference</i>	<i>Alloc. granularity</i>	<i>Cluster-wide policies</i>
Monolithic	all available	none (serialized)	global policy	strict priority (preemption)
Statically partitioned	fixed subset	none (partitioned)	per-partition policy	scheduler-dependent
Two-level (Mesos)	dynamic subset	pessimistic	hoarding	strict fairness

## Comparison of cluster scheduling approaches

## Architecture Details

### ● High-level summary of scheduling objectives

- ▶ Minimize the job queueing time, or more generally, the system response time (queueing + service times)
- ▶ Subject to
  - ★ Priorities among jobs
  - ★ Per-job constraints
  - ★ Failure tolerance
  - ★ Scalability

### ● Scheduling architectures

- ▶ Monolithic schedulers
- ▶ Statically partitioned schedulers
- ▶ Two-level schedulers
- ▶ Shared-state schedulers (cf. Omega paper)

# Monolithic Schedulers

- **Single centralized instance**

- ▶ Typical of HPC setting
- ▶ Implements all policies in a single code base
- ▶ Applies the same scheduling algorithm to all incoming jobs

- **Alternative designs**

- ▶ Support multiple code paths for different jobs
- ▶ Each path implements a different scheduling logic
- Difficult to implement and maintain

# Statically Partitioned Schedulers

- **Standard “Cloud-computing” approach**

- ▶ Underlying assumption: each framework has complete control over a set of resources
- ▶ Depend on statically partitioned, dedicated resources
- ▶ Examples: Hadoop 1.0, Quincy scheduler

- **Problems with static partitioning**

- ▶ Fragmentation of resources
- ▶ Sub-optimal cluster utilization

## Two-level Schedulers

- **Obviates the problems of static partitioning**

- ▶ Dynamic allocation of resources to concurrent frameworks
- ▶ Use a “logically centralized” coordinator to decide how many resources to grant

- **A first example: Mesos**

- ▶ Centralized resource allocator, with dynamic cluster partitioning
- ▶ **Available** resources are *offered* to competing frameworks
- ▶ **Avoids interference by exclusive offers**
- ▶ Frameworks lock resources by accepting offers → **pessimistic concurrency control**
- ▶ No global cluster state available to frameworks

- **Another tricky example: YARN**

- ▶ Centralized resource allocator (RM), with per-job framework master (AM)
  - ▶ AM only provides job management services, not proper scheduling
- YARN is closer to a monolithic architecture

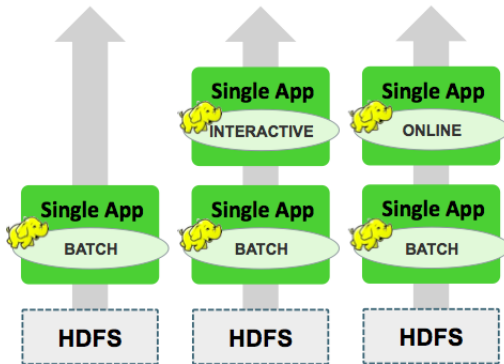


# Representative Cluster Schedulers

# YARN

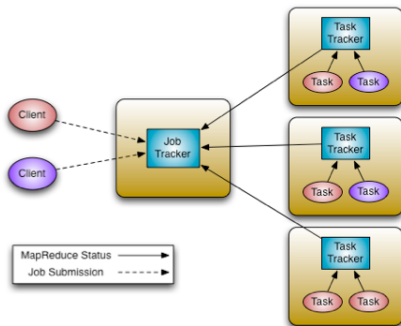
# Introduction and Motivations

## Hadoop 1.0: Focus on Batch applications



- **Built for batch applications**
  - ▶ Supports only MapReduce applications
- **Different silos for each usage pattern**

# Hadoop 1.0: Architecture (reloaded)



## • JobTracker

- ▶ Manages cluster resources
- ▶ Performs Job scheduling
- ▶ Performs Task scheduling

## • TaskTracker

- ▶ Per machine agent
- ▶ Manages Task execution

## Hadoop 1.0: Limitations

- **Only supports MapReduce, no other paradigms**

- ▶ Everything needs to be cast to MapReduce
- ▶ Iterative applications are slow

- **Scalability issues**

- ▶ Max cluster size roughly 4,000 nodes
- ▶ Max concurrent tasks, roughly 40,000 tasks

- **Availability**

- ▶ System failures destroy running and queued jobs

- **Resource utilization**

- ▶ Hard, static partitioning of resources in Map or Reduce slots
- ▶ Non-optimal resource utilization

## Next Generation Hadoop

### **Single Use System**

*Batch Apps*

## **HADOOP 1.0**

### **MapReduce**

(cluster resource management  
& data processing)

### **HDFS**

(redundant, reliable storage)



### **Multi Purpose Platform**

*Batch, Interactive, Online, Streaming, ...*

## **HADOOP 2.0**

### **MapReduce**

(data processing)

### **Others**

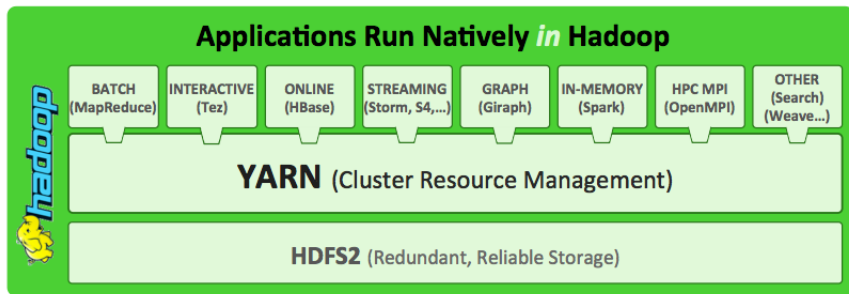
### **YARN**

(cluster resource management)

### **HDFS2**

(redundant, highly-available & reliable storage)

# The YARN ecosystem



- **Store all data in one place**
  - ▶ Avoids costly duplication
- **Interact with data in multiple ways**
  - ▶ Not only in batch mode, with the rigid MapReduce model
- **More predictable performance**
  - ▶ Advanced scheduling mechanisms



## Key Improvements in YARN (1)

### ● **Support for multiple applications**

- ▶ Separate generic resource brokering from application logic
- ▶ Define protocols/libraries and provide a framework for custom application development
- ▶ Share same Hadoop Cluster across applications

### ● **Improved cluster utilization**

- ▶ Generic resource container model replaces fixed Map/Reduce slots
- ▶ Container allocations based on locality and memory
- ▶ Sharing cluster among multiple application

### ● **Improved scalability**

- ▶ Remove complex app logic from resource management
- ▶ State machine, message passing based loosely coupled design
- ▶ Compact scheduling protocol

## Key Improvements in YARN (2)

- **Application Agility**

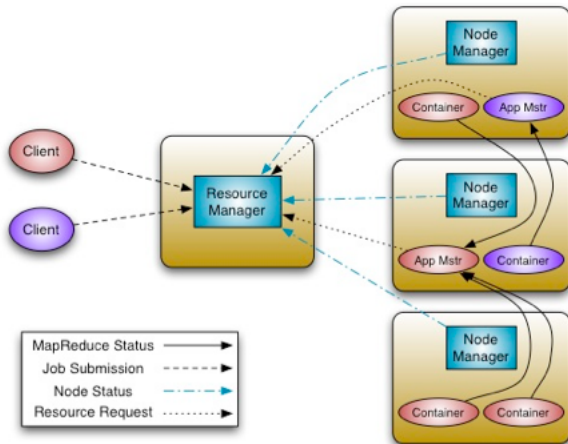
- ▶ Use Protocol Buffers for RPC gives wire compatibility
- ▶ Map Reduce becomes an application in user space
- ▶ Multiple versions of an app can co-exist leading to experimentation
- ▶ Easier upgrade of framework and applications

- **A data operating system: shared services**

- ▶ Common services included in a pluggable framework
- ▶ Distributed file sharing service
- ▶ Remote data read service
- ▶ Log Aggregation Service

# YARN Architecture Overview

# YARN: Architecture Overview



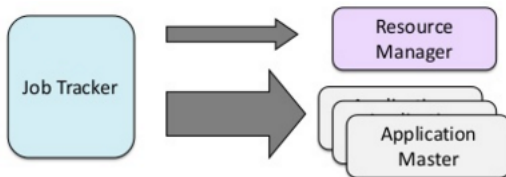
## YARN: Design Decisions

- **No static resource partitioning**

- ▶ There are no more slots
- ▶ Nodes have resources, which are allocated to applications when requested

- **Separate resource management from application logic**

- ▶ Cluster-wide resource allocation and management
- ▶ Per-application master component
- ▶ Multiple applications → multiple masters



## YARN Daemons

- **Resource Manager (RM)**

- ▶ Runs on master node
- ▶ Global resource manager and scheduler
- ▶ Arbitrates system resources between **competing** applications

- **Node Manager (NM)**

- ▶ Run on slave nodes
- ▶ Communicates with RM
- ▶ Reports utilization

- **Resource containers**

- ▶ Created by the RM upon request
- ▶ Allocate a certain amount of resources on slave nodes
- ▶ Applications run in one or more containers

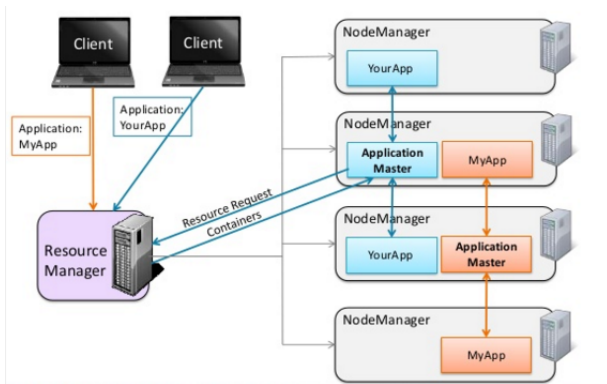
- **Application Master (AM)**

- ▶ One per application, **application specific**<sup>1</sup>
- ▶ Requests more containers to execute application tasks
- ▶ Runs in a container

---

<sup>1</sup>Every new application requires a new AM to be designed and implemented!

## YARN: Example with 2 Applications



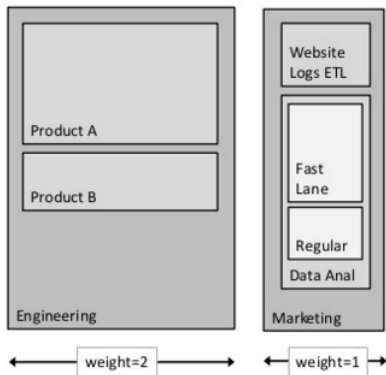
# YARN Core Components



## YARN Schedulers (1)

- **Schedulers are a pluggable component of the RM**
  - ▶ In addition to existing ones, advanced scheduling is supported
- **Current supported schedulers**
  - ▶ The Capacity scheduler
  - ▶ The Fair scheduler
  - ▶ Dominant Resource Fairness
- **What's different w.r.t. Hadoop 1.0?**
  - ▶ Support any YARN application, not just MapReduce
  - ▶ No more slots, tasks are scheduled based on resources
  - ▶ Some terminology change

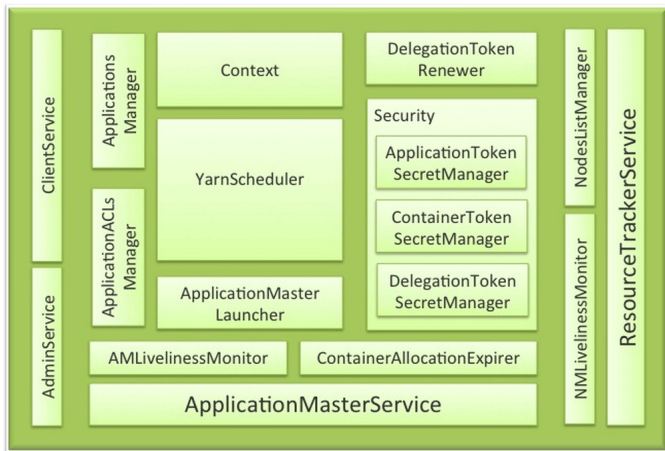
## YARN Schedulers (2)



- **Hierarchical queues**

- ▶ Queues can contain sub-queues
- ▶ Sub-queues share resources assigned to queues

# YARN Resource Manager: Overview



## YARN Resource Manager: Operations

- **Node Management**

- ▶ Tracks hearbeats from NMs

- **Container Management**

- ▶ Handles AM requests for new containers
- ▶ De-allocates containers when they expire or application finishes

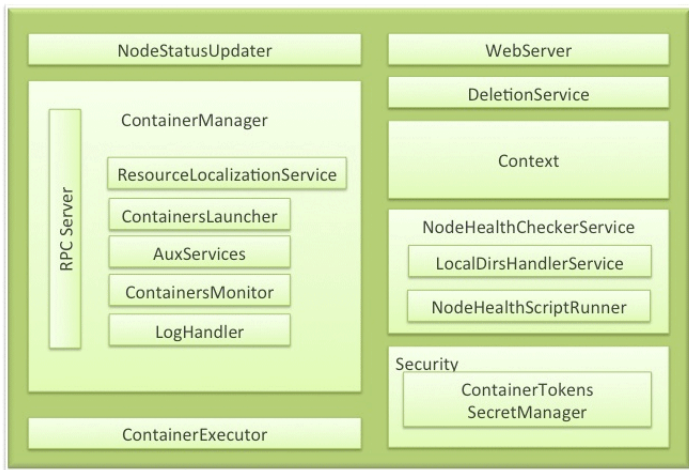
- **AM Management**

- ▶ Creates a container for every new AM, and tracks its health

- **Security Management**

- ▶ Kerberos integration

## YARN Node Manager: Overview



## YARN Node Manager: Operations

- **Manages communications with the RM**

- ▶ Registers, monitors and communicates node resources
- ▶ Sends heartbeats and container status

- **Manages processes in containers**

- ▶ Launches AMs on request from the RM
- ▶ Launches application processes on request from the AMs
- ▶ Monitors resource usage
- ▶ Kills processes and containers

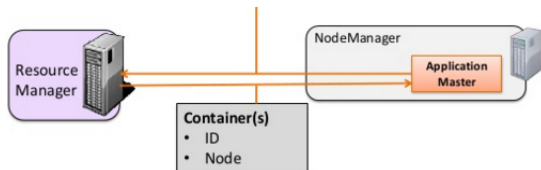
- **Provides logging services**

- ▶ Log aggregation and roll over to HDFS

# YARN Resource Request

## Resource Request

- **Resource name:** hostname, rackname, \*
- **Priority:** within the same application, not across apps
- **Resource requirements:** memory, CPU, and more to come...
- **Number of containers**



# YARN Containers

## Container Launch Context

- **Container ID**
- **Commands to start application task(s)**
- **Environment configuration**
- **Local resources:** application/task binary, HDFS files



# YARN Fault Tolerance

## ● Container failure

- ▶ AM re-attempts containers that complete with exceptions or fail
- ▶ Applications with too many failed containers are considered failed

## ● AM failure

- ▶ If application or AM fail, the RM will re-attempt the whole application
- ▶ Optional strategy: job recovery
  - ★ If false, all containers are re-scheduled
  - ★ If true, uses state to find which containers succeeded and which failed, to re-schedule only failed ones

## YARN Fault Tolerance

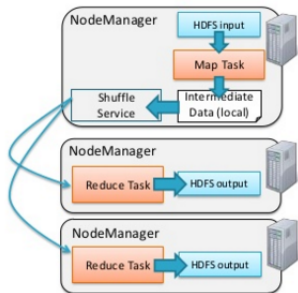
### ● NM failure

- ▶ If NM stops sending heartbeats, RM removes it from active node list
- ▶ Containers on the failed node are re-scheduled
- ▶ AM on the failed node are re-submitted completely

### ● RM failure

- ▶ No application can be run if RM is down
- ▶ Can work in active-passive mode (just like the NN of HDFS)

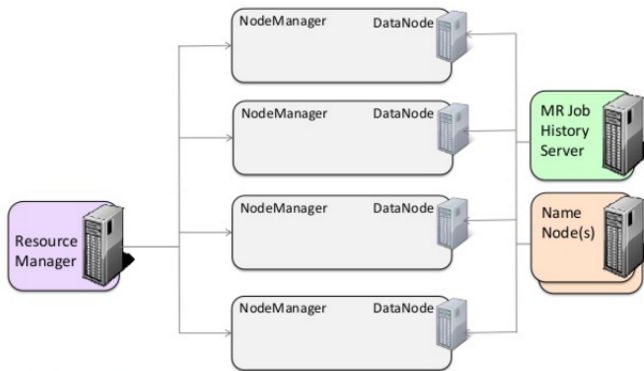
## YARN Shuffle Service



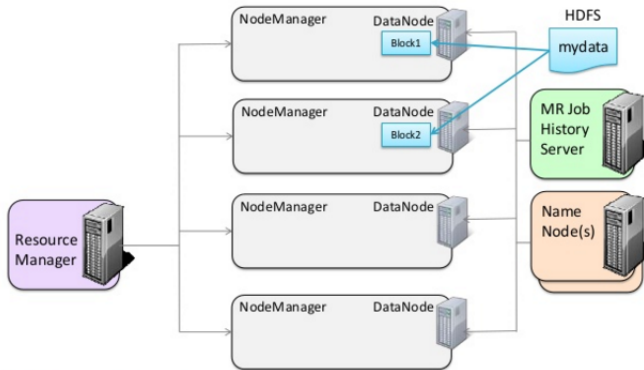
- **The Shuffle mechanism is now an auxiliary service**
  - ▶ Runs in the NM JVM as a persistent service

## YARN Application Example

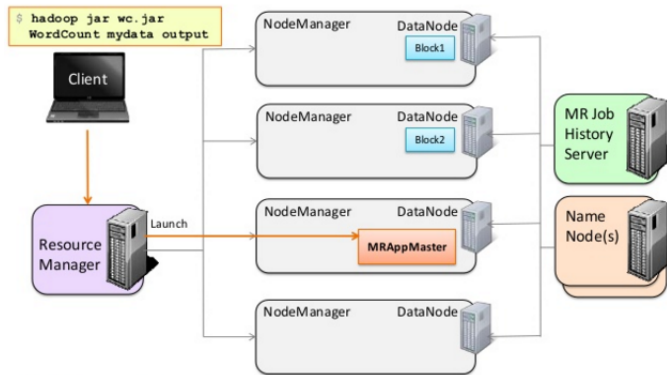
## YARN WordCount execution



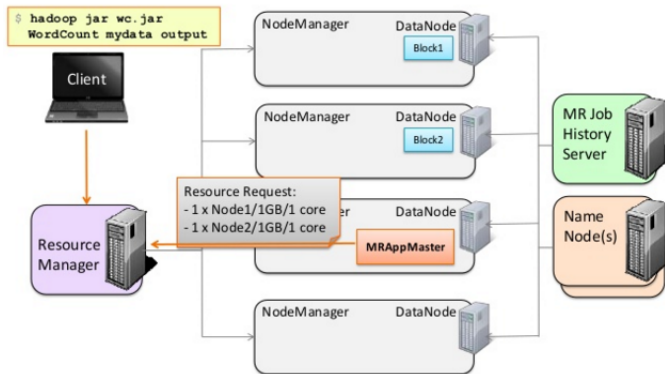
## YARN WordCount execution



## YARN WordCount execution

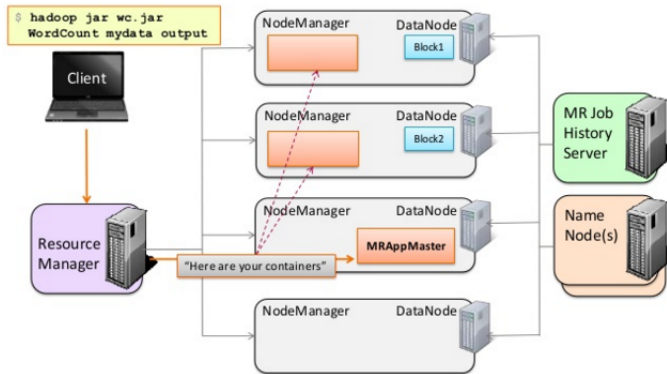


## YARN WordCount execution

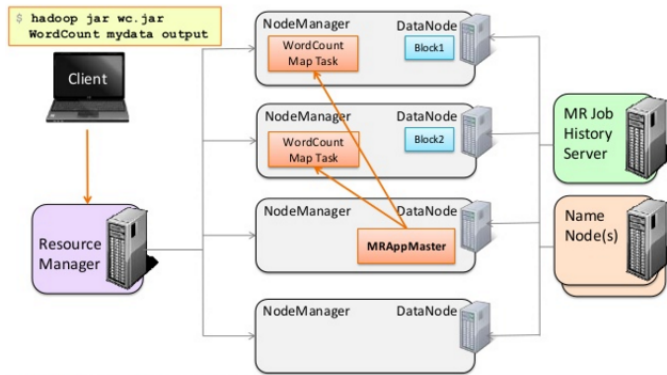




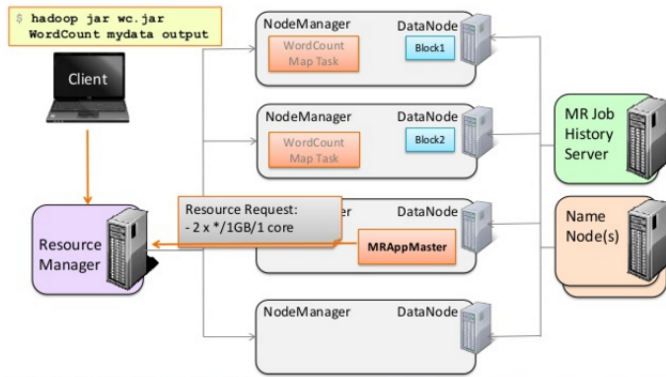
## YARN WordCount execution



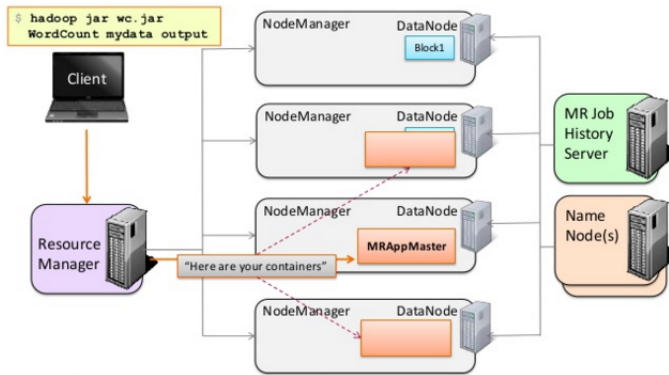
## YARN WordCount execution



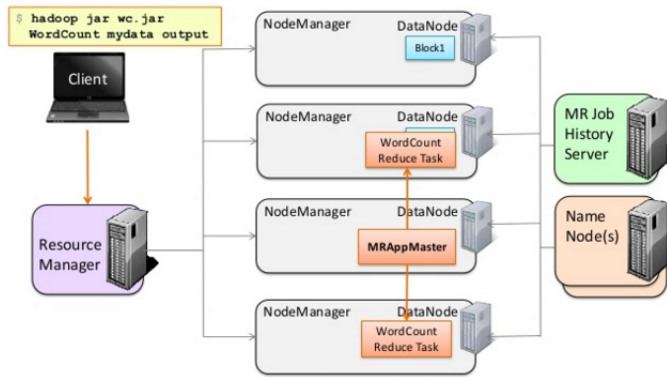
## YARN WordCount execution



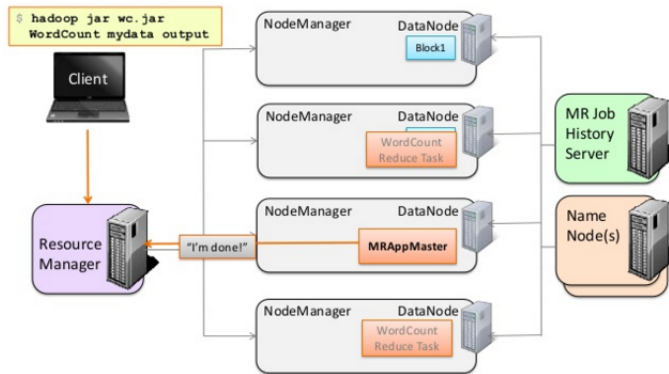
## YARN WordCount execution



## YARN WordCount execution



## YARN WordCount execution



# Mesos

# Introduction



## Introduction and Motivations

- **Clusters of commodity servers major computing platform**
    - ▶ Modern Internet Services
    - ▶ Data-intensive applications
  - **New *frameworks* developed to “program the cluster”**
    - ▶ Hadoop MapReduce, Apache Spark, Microsoft Dryad
    - ▶ Pregel, Storm, ...
    - ▶ and many more
  - **No *one-size fit them all***
    - ▶ Pick the right frameworks for the application
    - ▶ Run multiple frameworks at the same time
- **Multiplexing cluster resources among frameworks**
- ▶ Improves cluster utilization
  - ▶ Allows sharing of data without the need to replicate it

## Common Solutions to Share a Cluster

- **Common practice to achieve cluster sharing**

- ▶ *Static* partitioning
- ▶ Traditional virtualization

- **Problems of current approaches**

- ▶ Mismatch between allocation granularities
- ▶ No mechanism to allocate resources to short-lived tasks

→ **Underlying hypothesis for Mesos**

- ▶ Cluster frameworks operate with short tasks
- ▶ Cluster resources free up quickly
- ▶ This allows to achieve **data locality**

## Mesos Design Objectives

*Mesos: a thin resource sharing layer enabling fine-grained sharing across diverse frameworks*

### ● Challenges

- ▶ Each supported framework has different scheduling needs
- ▶ Scalability is crucial (10,000+ nodes)
- ▶ Fault-tolerance and high availability

### ● Would a centralized approach work?

- ▶ Input: framework requirements, instantaneous resource availability, organization policies
- ▶ Output: global schedule for all tasks of all jobs of all frameworks

## Mesos Key Design Principles

- **Centralized approach does not work**

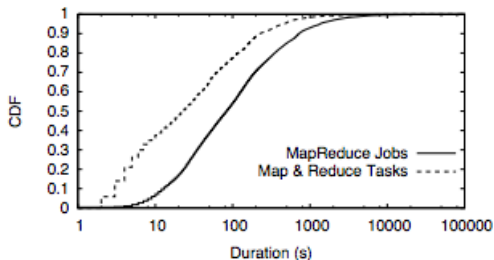
- ▶ Complexity
- ▶ Scalability and resilience
- ▶ Moving framework-specific scheduling to a centralized scheduler requires expensive refactoring

- **A decentralized approach**

- ▶ Based on the abstraction of a *resource offer*
- ▶ Mesos decides **how many** resources to offer to a framework
- ▶ The framework decides **which** resources to accept and which tasks to run on them

## Target Workloads

## Target Environment



- **Typical workloads in “Data Warehouse” systems**

- ▶ Heterogeneous MapReduce jobs, *production* and *ad-hoc* queries
- ▶ Large scale machine learning
- ▶ SQL-like queries

# Architecture

## Design Philosophy

- **Data center operating system**

- ▶ Scalable and resilient core exposing low-level interfaces
- ▶ High-level libraries for common functionalities

- **Minimal interface to support resource sharing**

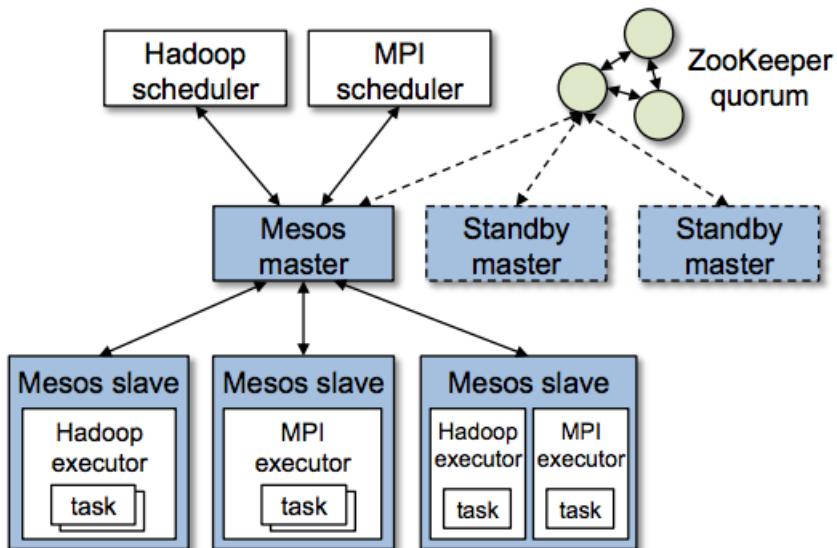
- ▶ Mesos manages cluster resources
- ▶ Frameworks control task scheduling and execution

- **Benefits of two-level approach**

- ▶ Frameworks are independent and can support diverse scheduling requirements
- ▶ Mesos is kept simple, minimizing the rate of change to the system



## Architecture Overview



## Architecture Overview

### The Mesos Master

- Uses **Resource Offers** to implement fine-grained sharing
- Collects resource utilization from slaves
- Resource offer: list of free resources on multiple slaves

### First-level Scheduling

- Master decides how many resources to offer a framework
- Implements a cluster-wide allocation policy:
  - ▶ Fair Sharing
  - ▶ Priority based

## Architecture Overview

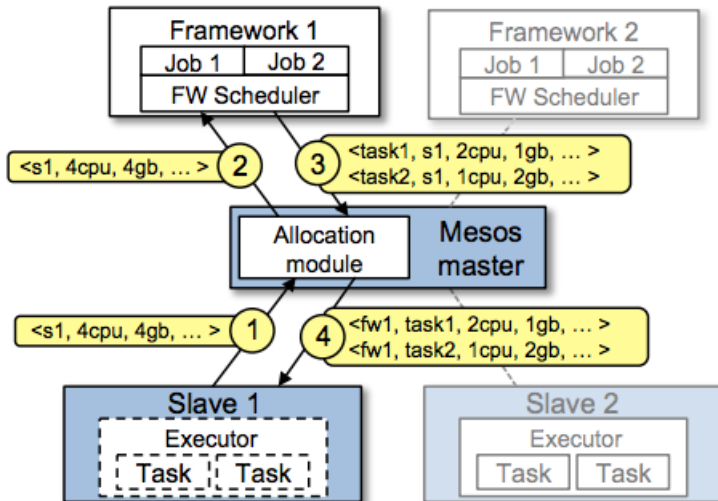
### Mesos Frameworks

- Framework **scheduler**
  - ▶ Registers to the master
  - ▶ Selects **which offer to accept**
  - ▶ Describes the tasks to launch on accepted resources
- Framework **executor**
  - ▶ Launched on Mesos slaves executing on accepted resources
  - ▶ Takes care of running the framework's tasks

### Second-level Scheduling

- One framework scheduler per application
- Framework decides how to execute a job and its tasks
- **NOTE:** The actual task execution is requested by the master

## Resource Offer Example



## Consequences of the Mesos Architecture

- **Mesos makes no assumptions on Framework requirements**
  - ▶ This is unlike other approaches, which requires the cluster scheduler to understand application constraints
  - ▶ This does not mean that users are not required to express their applications' constraints
- **Rejecting offers**
  - ▶ It is the framework that decides to reject a resource offer that does not satisfy application constraints
  - ▶ Frameworks can **wait** for offers to satisfy constraints

### Arbitrary and complex resource constraints

- Delegate logic and control to individual frameworks
- Mesos also implements **filters** to optimize resource offers

## Resource Allocation

- **Pluggable allocation module**

- ▶ Max-min Fairness
- ▶ Strict priority

- **Fundamental assumption**

- ▶ Tasks are short
- Mesos only reallocates resources when tasks finish

- **Example**

- ▶ Assume a Framework's share is 10% of the cluster
- ▶ It needs to wait 10% of the mean task length to receive its share

## Resource Revocation

- **Short vs. long lived tasks**

- ▶ Some jobs (e.g. streaming) may have long tasks
- ▶ In this case, Mesos can **Kill** running tasks

- **Preemption primitives**

- ▶ Require knowledge about potential resource usage by frameworks
- ▶ Killing might be wasteful, although not critical (e.g. MapReduce)
- ▶ Some applications (e.g. MPI) might be harmed

- **Guaranteed allocation**

- ▶ Minimum set of resources granted to a framework
- ▶ If below guaranteed allocation → never kill tasks
- ▶ If above guaranteed allocation → kill any tasks

## Performance Isolation

- **Isolation between executors on the same slave**

- ▶ Achieved through low-level OS primitives
- ▶ Pluggable isolation modules to support a variety of OS

- **Currently supported mechanisms**

- ▶ Limit CPU, memory, network and I/O bandwidth of a process tree
- ▶ Linux Containers and Solaris Cages

- **Advantages and limitations**

- ▶ Better isolation than current approach, process-based
- ▶ Fine grained isolation is not yet fully functional



## Mesos Scalability

### ● Filter mechanism

- ▶ Short-circuit the rejection process, avoids unnecessary communication
- ▶ Filter type 1: restrict which slave machines to use
- ▶ Filter type 2: check resource availability on slaves

### ● Incentives to speed-up the resource offer mechanism

- ▶ Mesos counts offers to a framework toward its allocation
- ▶ Frameworks have to answer and/or filter as quickly as possible

### ● Rescinding offers

- ▶ Mesos can decide to invalidate an offer to a framework
- ▶ This avoids blocking and misbehavior

## Mesos Fault Tolerance

- **Master designed with *Soft State***

- ▶ List of active slaves
- ▶ List of registered frameworks
- ▶ List of running tasks

- **Multiple masters in a hot-standby mode**

- ▶ Leader election through Zookeeper
- ▶ Upon failure detection new master is elected
- ▶ Slaves and executors help populating the new master's state

- **Helping frameworks to tolerate failure**

- ▶ Master sends “health reports” to framework schedulers
- ▶ Master allows multiple schedulers for a single framework

**System behavior: a very rough Mesos “model”**

## Overview: Mesos in a nutshell

### ● Ideal workloads for Mesos

- ▶ Elastic frameworks, supporting scaling up and down seamlessly
- ▶ Task durations are homogeneous (and short)
- ▶ No strict preference over cluster nodes

### ● Frameworks with cluster node preferences

- ▶ Assume frameworks prefer different (and possibly disjoint) nodes
- ▶ Mesos can emulate a centralized scheduler
- ▶ Cluster and Framework wide fair resource sharing

### ● Heterogeneous task durations

- ▶ Mesos can handle coexisting short and long lived tasks
- ▶ Performance degradation is acceptable

## Definitions

### ● Workload characterization

- ▶ *Elasticity*: elastic workloads can use resources as soon as they are acquired, and release them as soon as tasks finish; in contrast, rigid frameworks (e.g. MPI) can only start a job when **all** resources have been acquired, and do not work well with scaling
- ▶ *Task runtime distribution*: both homogeneous and not

### ● Resource characterization

- ▶ *Mandatory*: resource that a framework **must** acquire to work. Assumption: mandatory resources < guaranteed share
- ▶ *Preferred*: resources that a framework **should** acquire to achieve better performance, but are not necessary for the job to work

# Performance Metrics

## ● Performance metrics

- ▶ *Framework ramp-up time*: time it takes a new framework to achieve its fair share
- ▶ *Job completion time*: time it takes a job to complete. Assume one job per framework
- ▶ *System utilization*: total cluster resource utilization, with focus on CPU and memory

## Homogeneous Tasks

- Cluster with  $n$  slots and a framework  $f$  entitled with  $k$  slots
- Task runtime distribution: uniform and exponential
- Mean task duration  $T$
- Job duration:  $\beta k T$

→ If  $f$  has  $k$  slots, then job duration is  $\beta T$

	Elastic Framework		Rigid Framework	
	Constant dist.	Exponential dist.	Constant dist.	Exponential dist.
Ramp-up time	$T$	$T \ln k$	$T$	$T \ln k$
Completion time	$(1/2 + \beta)T$	$(1 + \beta)T$	$(1 + \beta)T$	$(\ln k + \beta)T$
Utilization	1	1	$\beta/(1/2 + \beta)$	$\beta/(\ln k - 1 + \beta)$

## Placement preferences

Consider two cases:

- **There exist a configuration satisfying all frameworks constraints**
  - ▶ The system will eventually converge to the state in which the optimal allocation is achieved, and this in at most one  $T$  interval
- **No such allocation exists, e.g. demand is larger than supply**
  - ▶ **Lottery Scheduling** to achieve a **weighted fair allocation**
  - ▶ Mesos offers a slot to framework  $i$  with probability

$$\frac{s_i}{\sum_{i=1}^m s_i}$$

- ▶ where  $s_i$  is framework's  $i$  **intended** allocation, and  $m$  is the total number of frameworks registered to Mesos



## Heterogeneous Tasks

- **Assumptions**

- ▶ Workloads with tasks that are either long or short
- ▶ Mean duration of long task is longer than short ones

- **Worst case scenario**

- ▶ All nodes required by a “short job” are filled with long tasks, which means it has to wait for a long time

- **How likely is the worst case?**

- ▶ Assume  $\phi < 1$ , where  $\phi$  fraction of long tasks
- ▶ Assume a cluster with  $S$  available slots per node
- Probability for a node to be filled with long tasks is  $\phi^S$
- ▶  $S = 8$  and  $\phi = 0.5$  gives a 0.4% chance

## Limitations of Distributed Scheduling

### ● Fragmentation

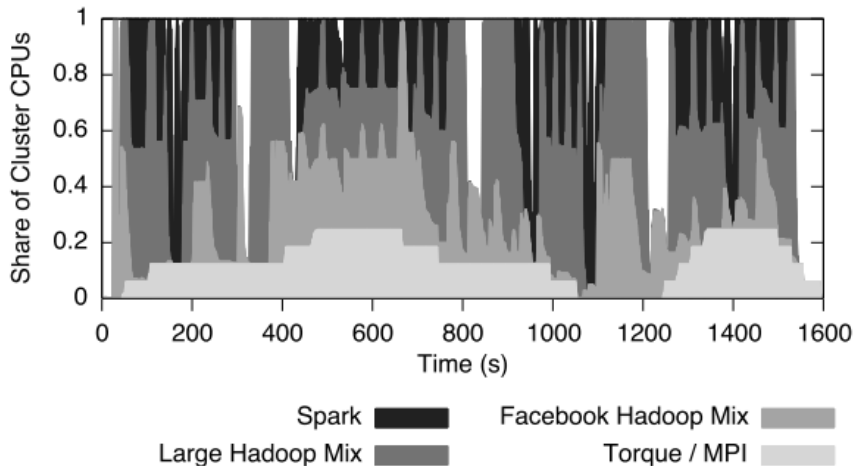
- ▶ Provokes under utilization of system resources
- ▶ Distributed collection of frameworks might not achieve the same “packing” quality of a centralized scheduler
- This is mitigated by having clusters of “big” nodes (many CPUs, many cores) running “small” tasks

### ● Starvation

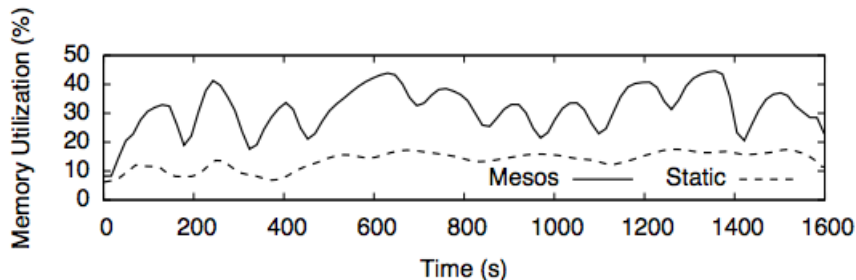
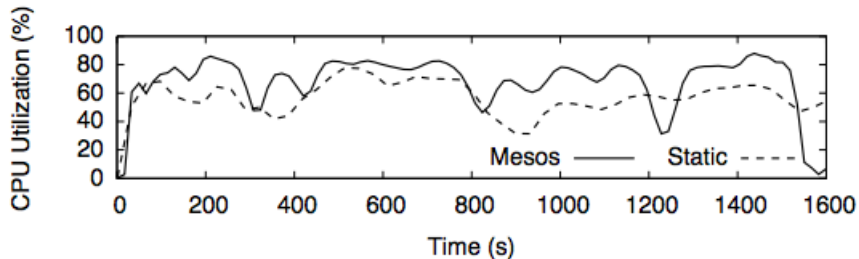
- ▶ Large jobs may wait indefinitely for slots to become free
- ▶ Small tasks from small jobs might monopolize the cluster
- This is mitigated by a *minimum offer size* mechanism

# Experimental Mesos Performance Evaluation

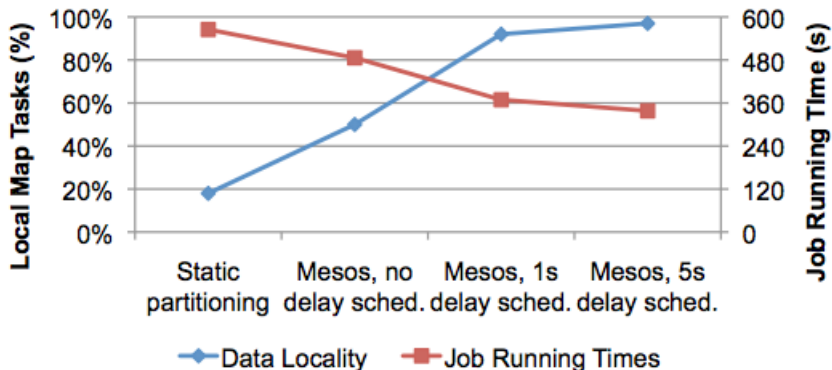
## Resource Allocation



## Resource Utilization



## Data Locality



# Borg

# Introduction



# Introduction and Objectives

- **Hide the details of resource management**
  - ▶ Let users instead focus on application development
- **Operate applications with high reliability and availability**
  - ▶ Tolerate failures within a datacenter and across datacenters
- **Run heterogeneous workloads and scale across thousands of machines**

# The User Perspective

## Terminology

- **Users develop applications called *jobs***
- **Jobs consists in one or more *tasks***
- **All tasks run the same binary**
- **Each job runs in a set of machines managed as a unit, called a *Borg Cell***

## Workloads

- **Two main categories supported**

- ▶ Long-running services: jobs that should “never” go down, and handle short-lived, latency-sensitive requests
- ▶ Batch jobs: delay-tolerant jobs that can take from few seconds to few days to complete
- ▶ Storage services: these are long-running services like above, that are used to store data

- **Workload composition in a cell is dynamic**

- ▶ It varies depending on the tenants using the cell
- ▶ It varies with time: diurnal usage pattern for end-user-facing jobs, irregular pattern for batch jobs

- **Examples**

- ▶ High-priority, `production` jobs → long-running services
- ▶ Low-priority, `non-production` jobs → batch jobs
- ▶ In a typical Borg Cell
  - ★ Prod Jobs: 70% of CPU allocation, representing 60% of CPU usage
  - ★ Non-prod Jobs: 55% of CPU allocation, representing 85% of CPU usage

## Clusters and Cells

- **Borg Cluster:** a set of machines connected by a high-performance datacenter-scale network fabric
  - ▶ The machines in a Borg Cell all belong to a *single* cluster
  - ▶ A cluster lives inside a datacenter building
  - ▶ A collection of building makes up a *Site*
- **Borg Machines:** physical servers dedicated to execute Borg applications
  - ▶ They are generally highly heterogeneous in terms of resources
  - ▶ They may expose a public IP address
  - ▶ They may expose advanced features, like SSD or GPGPU
- **Examples**
  - ▶ A typical cluster usually hosts one large cell and a few small-scale test cells
  - ▶ The *median cell size* is about 10k machines
  - ▶ Borg uses those machines to schedule application tasks, install their binaries and dependencies, monitor their health and restarting them if they fail

## Jobs and Tasks

### ● Job Definition

- ▶ Name, owner and number of tasks
- ▶ *Constraints* to force tasks run on machines with particular *attributes*
- ▶ Constraints can be **hard** or **soft** (*i.e.*, preferences)
- ▶ Each task maps to a set of UNIX processes running in a container on a Borg machine in a Borg Cell

### ● Task Definition

- ▶ Task index within their parent job
- ▶ Resource requirements
- ▶ Generally, all tasks have the *same* definition
- ▶ Tasks can run on **any resource dimension**: there are no fixed-size slots or buckets

## Jobs and Tasks

- **The Borg Configuration Language**

- ▶ Declarative language to specify jobs and tasks
- ▶ Lambda functions to allow calculations
- ▶ Some application descriptions can be over 1k lines of code

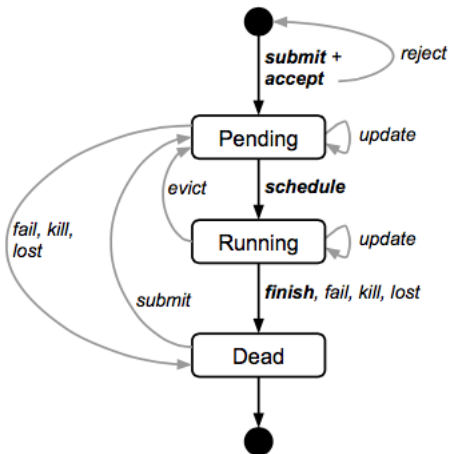
- **User interacting with live jobs**

- ▶ This is achieved mainly using RPC
- ▶ Users can update the specification of tasks, while their parent job is running
- ▶ Updates are non-atomic, executed in a rolling-fashion

- **Task updates “side-effects”**

- ▶ Always require restarts: e.g., pushing a new binary
- ▶ Might require migration: e.g., change in specification
- ▶ Never require restarts nor migrations: e.g., change in priority

# Jobs State Diagram





## Resource Allocations

### ● The Borg “Alloc”

- ▶ Reserved set of resources on **an individual machine**
- ▶ Can be used to execute one or more tasks, that equally share resources
- ▶ **Resources remain assigned whether or not they are used**

### ● Typical use of Borg Allocs

- ▶ Set resources aside for future tasks
- ▶ Retain resources between stopping and starting tasks
- ▶ Consolidate (gather) tasks from different jobs on the same machine

### ● Alloc Sets

- ▶ Group of allocs on different machines
- ▶ Once an alloc set has been created, one or more jobs can be submitted

## Priority, Quota and Admission Control

- **Mechanisms to deal with resource demand and offer**

- ▶ What to do when more work shows up than can be accommodated?
- ▶ Note: this is not *scheduling*, it is more admission control

- **Job priority**

- ▶ Non-overlapping *priority bands* for different uses
- ▶ This essentially means users must “manually” cluster their applications according to such bands
- Tasks from high-priority jobs can preempt low-priority tasks
- ▶ Cascade preemption is avoided by disabling it for same-band jobs

- **Job/User quotas**

- ▶ Used to decide which job to admit for scheduling
- ▶ Expressed as a vector of resource quantities

- **Pricing**

- ▶ Underlying mechanism to regulate user behavior
- ▶ Aligns user incentives to better resource utilization
- ▶ Discourages over-buying by over-selling quotas at lower priority

## Naming Services

- **Borg Name Service**

- ▶ A mechanism to assign a name to tasks
- ▶ Task name = Cell name, job name and task number

- **Uses the Chubby coordination service**

- ▶ Writes task names into it
- ▶ Writes also health information and status
- ▶ Used by Borg RPC mechanism to establish communication endpoints

- **DNS service inherits from BNS**

- ▶ Example: the 50th task in job “jfoo” owned by user “ubar” in a Borg Cell called “cc”
- ▶ `50.jfoo.ubar.cc.borg.google.com`

## Monitoring Services

- **Every task in Borg has a built-in HTTP server**

- ▶ Provides health information
- ▶ Provides performance metrics

- **Borg SIGMA**

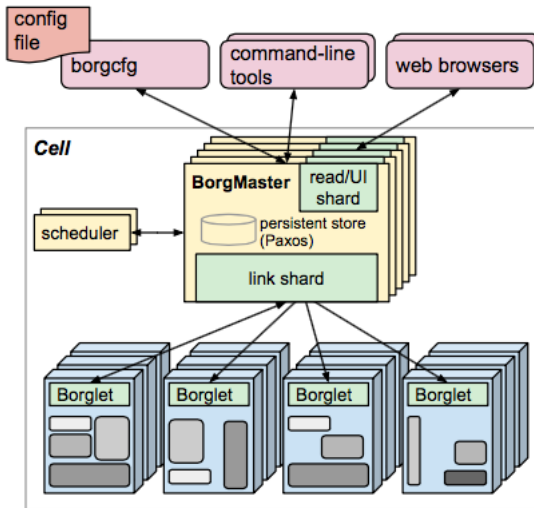
- ▶ Monitoring UI Service
- ▶ State of jobs, of cells
- ▶ Drill-down to task level
- ▶ **Why pending?**
  - ★ “Debugging” service
  - ★ Helps users with finding job specifications that can be easily scheduled

- **Billing services**

- ▶ Use monitoring information to compute usage-based charging
- ▶ Help users debug their jobs
- ▶ Used for capacity planning

# Borg Architecture

# Architecture Overview



# Architecture Overview

- **Architecture components**

- ▶ A set of physical machines
- ▶ A *logically centralized* controller, the **Borgmaster**
- ▶ An agent process running on all machines, the **Borglet**

## The Borgmaster: Components

- **The Borgmaster**

- ▶ One per Borg Cell
- ▶ Orchestrates cell resources

- **Components**

- ▶ The Borgmaster process
- ▶ The scheduler process

- **The Borgmaster process**

- ▶ Handles client RPCs that either mutate state or lookup for state
- ▶ Manages the state machines for all Borg “objects” (machines, tasks, allocs, etc...)
- ▶ Communicates with all Borglets in the cell
- ▶ Provides a Web-based UI



## The Borgmaster: Reliability

- **Borgmaster reliability achieved through replication**

- ▶ Single logical process, replicated 5 times in a cell
- ▶ Single master elected using Paxos when starting a cell, or upon failure of the current master
- ▶ Master serves as Paxos leader and cell state mutator

- **Borgmaster replicas**

- ▶ Maintain an **in-memory** fresh copy of the cell state
- ▶ Persist their state to a distributed Paxos-based store
- ▶ Help building the most up-to-date cell state when a new master is elected

# The Borgmaster: State

- **Borgmaster checkpoints its state**

- ▶ Time-based and event-based mechanism
- ▶ State include everything related to a cell

- **Checkpoint utilization**

- ▶ Restore the state to a functional one, e.g. before a failure or a bug
- ▶ Studying a faulty state and fixing it by hand
- ▶ Build a persistent log of events for future queries
- ▶ Use it for offline simulations

# The Fauxmaster

- **A high-fidelity simulator**

- ▶ It reads checkpoint files
- ▶ Full-fledged Borgmaster code
- ▶ Stubbed-out interfaces to Borglets

- **Fauxmaster operation**

- ▶ Accepts RPCs to make state machine changes
- ▶ Connects to simulated Borglets that replay real interactions from checkpoint files

- **Fauxmaster benefits**

- ▶ Help users debug their application
- ▶ Capacity planning, e.g. “How many new jobs of this type would fit in the cell?”
- ▶ Perform sanity checks for cell configurations, e.g. “Will this new configuration evict any important jobs?”

# Scheduling

- **Queue based mechanism**

- ▶ New submitted jobs (and their tasks) are stored in the Paxos store (for reliability) and put in the *pending queue*

- **The scheduler process**

- ▶ Operates at the task level, not the job level
- ▶ Scans **asynchronously** the pending queue
- ▶ Assigns tasks to machines that satisfy constraints and that have enough resources

- **Pending task selection**

- ▶ Scanning proceeds from high to low priority tasks
- ▶ Within the same priority class, scheduling uses a round-robin mechanism
- Ensures fairness
- Avoids head-of-line blocking behind large jobs

# Scheduling Algorithm

- **The scheduling algorithm has two main processes**

- ▶ **Feasibility checking:** find a set of machines that
  - ★ Meet tasks' constraints
  - ★ Have enough available resources, including those that are currently assigned to low-priority tasks that can be evicted
- ▶ **Scoring:** among the set returned by the previous process, rank such machines to
  - ★ Minimize the number and priority of preempted tasks
  - ★ Prefer machines with a local copy of tasks binaries and dependencies
  - ★ Spread tasks across failure and power domains
  - ★ Pack and spread tasks, mixing high and low priority ones on the same machine to allow high-priority tasks to eventually expand

## More on the scoring mechanism

- **Worst-fit scoring: spreading tasks**

- ▶ Single cost value across heterogeneous resources
- ▶ Minimize the change in cost when placing a new task
- Leaves headroom for load spikes
- But leads to fragmentation

- **Best-fit scoring: “waterfilling” algorithm**

- ▶ Tries to fill machines as tightly as possible
- Leaves empty machines that can be used to place large tasks
- But difficult to deal with load spikes as the headroom left in each machine depends highly on load estimation

- **Hybrid**

- ▶ Tries to reduce the amount of stranded resources
- ▶ Performs better than best-fit

## Task startup latency

- **Task startup latency is a very important metric to optimize**
  - ▶ Time from job submission to a task running
  - ▶ Highly variable
  - ▶ E.g.: at Google, median was about 25s
- **Techniques to reduce latency**
  - ▶ The main culprit for high latency is binary and package installations
  - ▶ Idea: place tasks on machines that already have dependencies installed
  - ▶ Packages and binaries distributed using a BitTorrent-like protocol

# The Borglet

## ● The Borglet

- ▶ Borg agent present on every machine in a cell
- ▶ Starts and stop tasks
- ▶ Restarts failed tasks
- ▶ Manages machine resources interacting with the OS
- ▶ Maintains and rolls over debug logs
- ▶ Report the state of the machine to the Borgmaster

## ● Interaction with the Borgmaster

- ▶ Pull-based mechanism: heartbeat-like messages every few seconds
- Borgmaster perform flow and rate control to avoid message storms
- ▶ Borglet continues operation even if communication to Borgmaster is interrupted
- ▶ A failed Borglet is blacklisted and all tasks are rescheduled



## Borglet to Borgmaster communication

- **How to handle control message overhead?**

- ▶ Many Borgmaster replicas receive state updates
- ▶ Many Borglets communicate concurrently

- **The link shard mechanism**

- ▶ Each borgmaster replica communicates with a subset of the cell Borglets
  - ▶ Partitioning is computed at each leader election
  - ▶ Borglets report full state, but the link shard mechanism aggregate state information
- Differential state update, to reduce the load at the master

## Scalability

- **A typical Borgmaster resource requirements**

- ▶ Manages 1000s of machines in a cell
- ▶ Arrival rates of 10,000 tasks per minute
- ▶ 10+ cores, 50+ GB of RAM

- **Decentralized design**

- ▶ Scheduler process separate from Borgmaster process
- ▶ One scheduler per Borgmaster replica
- ▶ Scheduling is somehow decentralized
- ▶ State change communicated from replicas to elected Borgmaster, that finalizes the state update

- **Additional techniques to achieve scalability**

- ▶ Score caching
- ▶ Equivalence class
- ▶ Relaxed randomization

## **Borg Behavior: Experimental Perspective**

Also, additional details on how Borg works...

## Availability

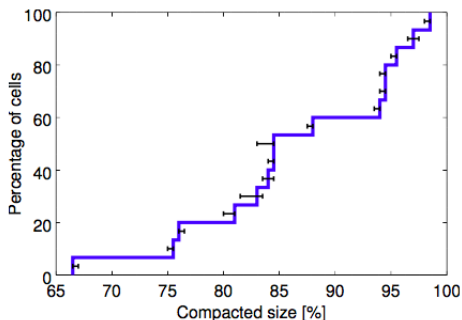
- **In large scale systems, failure are the norm not the exception**
  - ▶ Everything can fail, and both Borg and its running applications must deal with this
- **Baseline techniques to achieve high availability**
  - ▶ Replication
  - ▶ Storing persistent state in a distributed file system
  - ▶ Checkpointing
- **Additional techniques**
  - ▶ Automatic rescheduling of failed tasks
  - ▶ Mitigating correlated failures
  - ▶ Rate limitation
  - ▶ Avoid duplicate computation
  - ▶ Admission control to avoid overload
  - ▶ Minimize external dependencies for task binaries



## System Utilization

- **The primary goal of a cluster scheduler is to achieve high utilization**
  - ▶ Machines, network fabric, power, cooling ... represent a significant financial investment
  - ▶ Increasing utilization by a few percent can save millions!
- **A sophisticated metric: Cell Compaction**
  - ▶ Replaces the typical “average utilization” metric
  - ▶ Provides a fair, consistent way to compare scheduling policies
  - ▶ Translates directly into cost/benefit result
  - ▶ Computed as follows:
    - ★ Given a workload in a point in time (so this is not trace driven simulation)
    - ★ Enter a loop of workload packing
    - ★ At each iteration, remove physical machines from the cell
    - ★ Exit the loop when the workload can no longer fit the cell size
- **Use the Fauxmaster to produce experimental results**

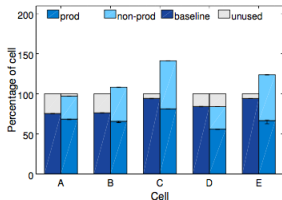
## System Utilization: Compaction



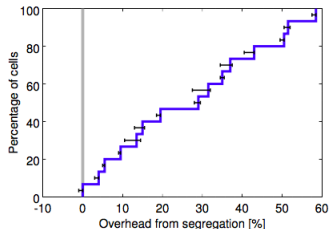
- This graph shows how much smaller cells would be if we applied compaction to them

## System Utilization: Cell Sharing

- **Fundamental question: to share or not to share?**
  - ▶ Many current systems apply static partitioning: one cluster **dedicated only** to prod jobs, one cluster for non-prod jobs
- **Benefits from sharing**
  - ▶ Borg can reclaim resources reserved by “anxious” prod jobs



(a) The left column for each cell shows the original size and the combined workload; the right one shows the segregated case.

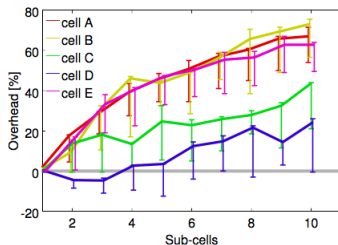


(b) CDF of additional machines that would be needed if we segregated the workload of 15 representative cells.

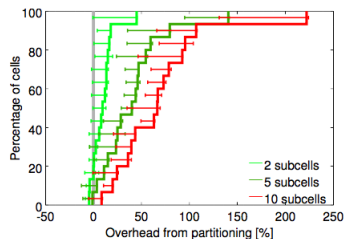
## System Utilization: Cell Sizing

### ● Fundamental question: large or small cells?

- ▶ Large cells to accommodate large jobs
- ▶ Large cells also avoid fragmentation



(a) Additional machines that would be needed as a function of the number of smaller cells for five different original cells.

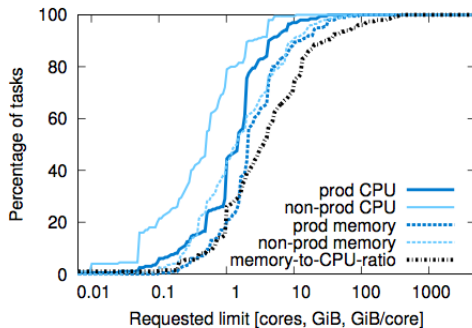


(b) A CDF of additional machines that would be needed to divide each of 15 different cells into 2, 5 or 10 cells.



## System Utilization: Fine-grained Resource Requests

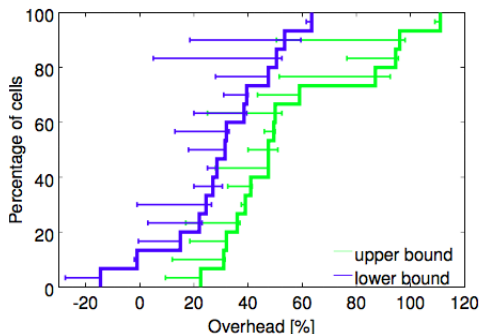
- **Borg users specify Job requirements in terms of resources**
  - ▶ For CPU: this is done in milli-core
  - ▶ For RAM and disk: this is done in bytes



## System Utilization: Fine-grained Resource Requests

- **Would fixed size containers (or slot) be good?**

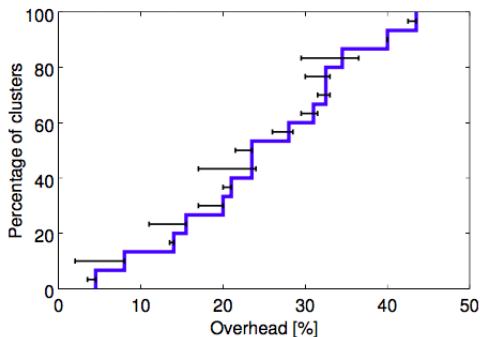
- ▶ No! It would require more machines in a cell!



## System Utilization: Resource Reclamation

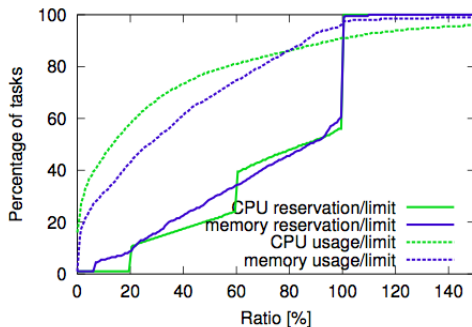
- **Borg users specify resource *limits* for their jobs**
  - ▶ Used to perform admission control
  - ▶ Used for feasibility checking (*i.e.*, find sets of suitable machines)
- **Borg users have the tendency to “over provision” for their jobs**
  - ▶ Some tasks, occasionally need to use all their resources
  - ▶ Most of the tasks never use all resources
- **Resource reclamation**
  - ▶ Borg builds estimates of resource usage: these are called **resource reservations**
  - ▶ Borgmaster receives usage updates from Borglets
  - ▶ Prod jobs are treated differently: they do not rely on reclaimed resources, Borgmaster only uses resource limits

## System Utilization: Resource Reclamation



- Resource reclamation is quite effective. A CDF of the additional machines that would be needed if it was disabled.

## System Utilization: Resource Reclamation



- Resource estimation is successful at identifying unused resources. Most tasks use much less than their limit, although a few use more CPU than requested.

# Isolation

- **Sharing a multi-tenancy are beneficial, but...**

- ▶ Tasks may interfere one with each other
- ▶ Need a good mechanism to prevent interference (both in terms of security and performance)

- **Performance isolation**

- ▶ All tasks run in Linux cgroup-based containers
- ▶ Borglets operate on the OS to control container resources
- ▶ A control-loop assigns resources based on predicted future usage or on memory pressure

- **Additional techniques**

- ▶ Application classes: latency-sensitive, vs. batch
- ▶ Resource classes: compressible and non-compressible
- ▶ Tuning of the underlying OS, especially the OS scheduler

## **Lessons learned from building and operating Borg**

And what has been included in Kubernetes, the open source version of Borg...

## Lessons Learned: the Bad

- **The “job” abstraction is too simplistic**

- ▶ Multi-job services cannot be easily managed, nor addressed
- ▶ Kubernetes uses scheduling units called **Pods** (*i.e.*, the Borg allocs) and **labels** (key/value pairs describing objects)

- **Addressing services is critical**

- ▶ One IP address implies managing ports as a resource, which complicates tasks
- ▶ Kubernetes uses Linux name-spaces, such that each pod has its own IP address

- **Power or casual users?**

- ▶ Borg is geared toward power users: e.g., BCL has about 230 parameters!
- ▶ Build automation tools, and template settings from historical executions



## Lessons Learned: the Good

- **Allocs are useful**

- ▶ Resource envelope for one or more container co-scheduled on the same machine and that can share resources

- **Cluster management is more than task management**

- ▶ Naming and load balancing are first-class citizens

- **Introspection is vital**

- ▶ Clearly true for debugging
- ▶ Key for capacity planning and monitoring

- **The master is the kernel of a distributed system**

- ▶ Monolithic designs are not working well
- ▶ Cooperation of micro-services that use a common low-level API to process requests and manipulate state