IEMS5730 Big Data Systems and Information Processing

Resource Management Platforms for Big Data Processing Systems

Prof. Wing C. Lau
Department of Information Engineering
wclau@ie.cuhk.edu.hk

Acknowledgements

- The slides used in this chapter are adapted from the following sources:
  - "Data-Intensive Information Processing Applications," by Jimmy Lin, University of Maryland.

  - "Intro To Hadoop" in UCBerkeley i291 - Analyzing BigData with Twitter, by Bill Graham, Twitter.
  - Ryza of Cloudera Inc, "Can't we just get along ?", Spark Summit, 2013
  - Cloudera, "Introduction to YARN and MapReduce 2", SlideShare.net
  - Eric Brewer, Google VP of Instructure, "Google Tech Talk – Containers: What, Why, How ; Google Cloud Innovation", April 2015
  - Ajit Punj, Juan Manuel Camacho, Borg – a presentation for Stanford CS349d, Fall 2018
  - Alex Gilkson of CMU, "Cloud-Native Applications and Kubernetes (k8s), 2019
  - Cyberlearn CLOUD 2019-2020 (Master) MSE Lecture notes on Kubernetes

    https://cyberlearn.hes-so.ch/course/view.php?id=14014
  - Kubernauts – The Cloud Cosmonauts "The Kubernetes Learning Slides," v0.15.1, June 15, 2020.

    https://docs.google.com/presentation/d/13EQKZSQDounPC1I6EC4PmqaRmdCrpT3qswQJz9KRCyE/htmlpresent
  - Bob Killen, Cloud Native Computing Foundation (CNCF) Ambassador, "Kubernetes – an Introduction," July 2019.

    https://docs.google.com/presentation/d/1zrfVIE5r61ZNQrmXKx5gJmBcXnoa_WerHEnTxu5SMco/edit#slide=id.g3cfa019267_4_0
  - Alex Gilkson of CMU, "Cloud-Native Applications and Kubernetes (k8s), 2019

Hadoop 1.0 vs. Hadoop 2.0 Ecosystem

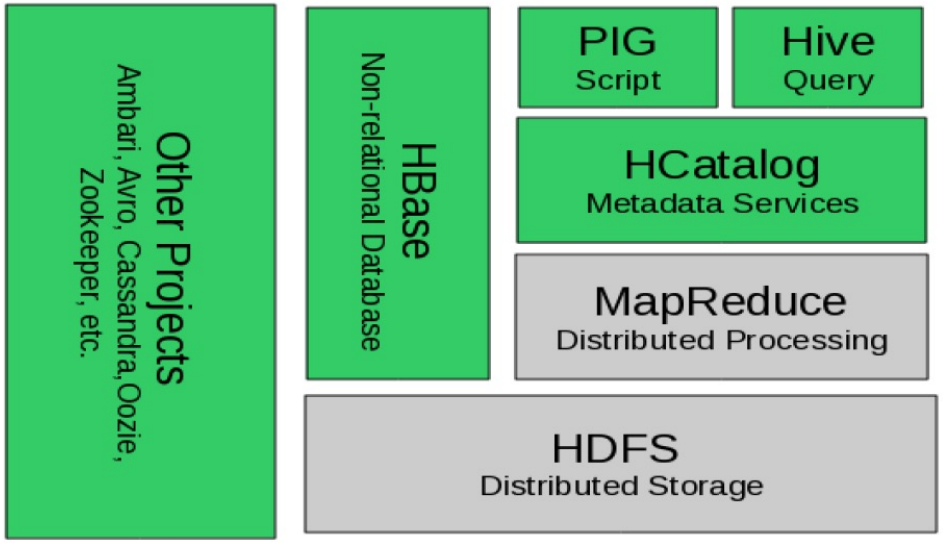


Figure 2.1 The Hadoop 1.0 ecosystem, MapReduce and HDFS are the core components, while other are built around the core.

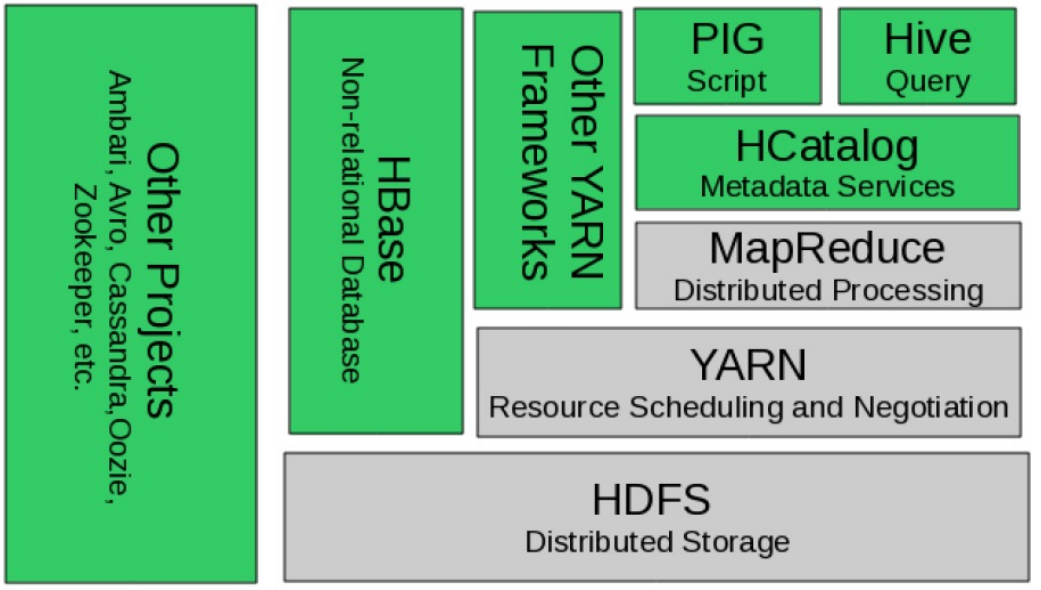


Figure 2.2 YARN adds a more general interface to run non-MapReduce jobs within the Hadoop framework

Practical Scalability Limits of Hadoop1.0

- Scalability
  - Maximum Cluster Size – 4000 Nodes
  - Maximum Concurrent Tasks – 40000
  - Coarse synchronization in Job Tracker
- Single point of failure
  - Failure kills all queued and running jobs
  - Jobs need to be resubmitted by users
- Restart is very tricky due to complex state

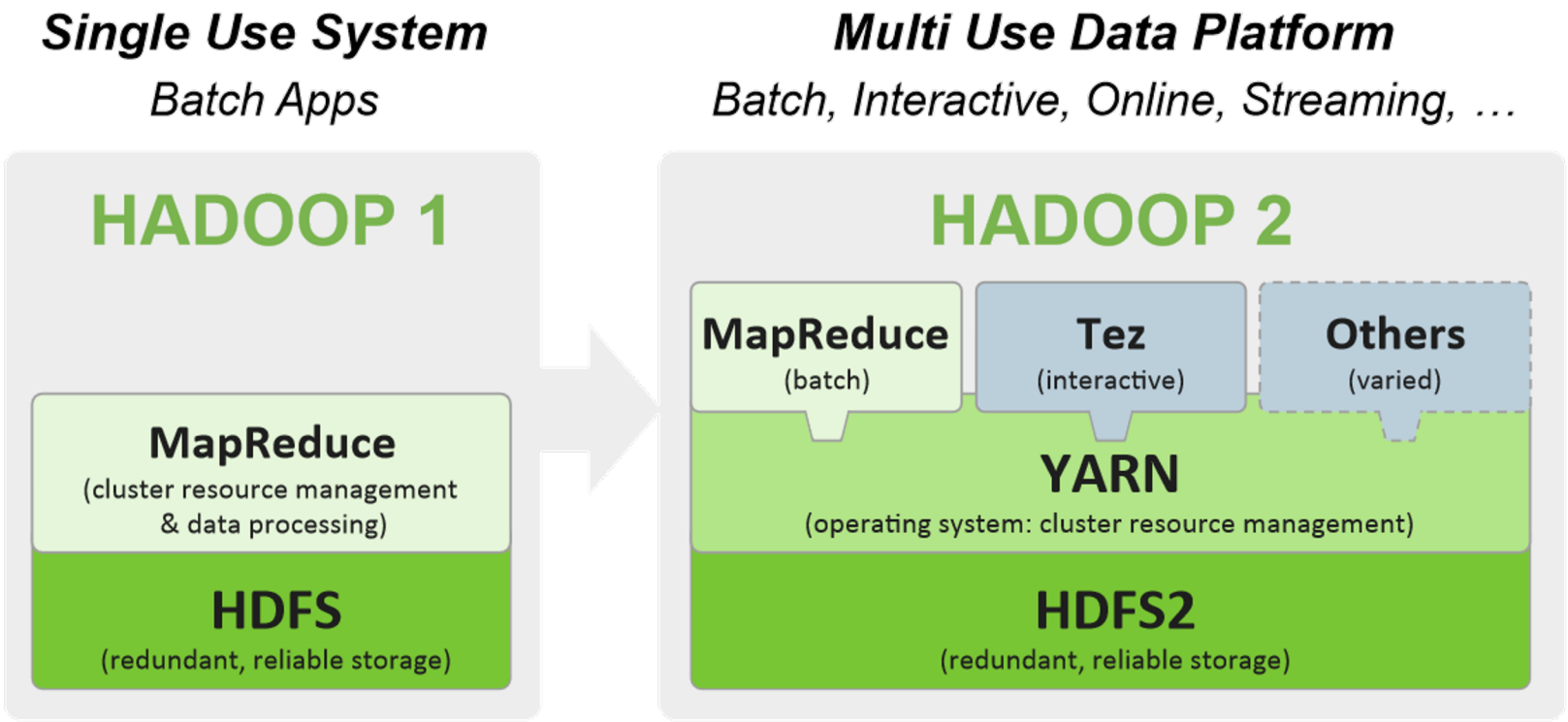
Scalability/Flexibility Issues of the MapReduce/ Hadoop 1.0 Job Scheduling/Tracking

- The MapReduce Master node (or Job-tracker in Hadoop 1.0) is responsible to monitor the progress of ALL tasks of all jobs in the system and launch backup/replacement copies in case of failures
  - For a large cluster with many machines, the number of tasks to be tracked can be huge

  => Master/Job-Tracker node can become the performance bottleneck
- Hadoop 1.0 platform focuses on supporting MapReduce as its only computational model ; may not fit all applications
- Hadoop 2.0 introduces a new resource management/ job-tracking architecture, YARN [1], to address these problems

[1] V.K. Vavilapalli, A.C.Murthy, “Apache Hadoop YARN: Yet Another Resource Negotiator,” ACM Symposium on Cloud Computing 2013.

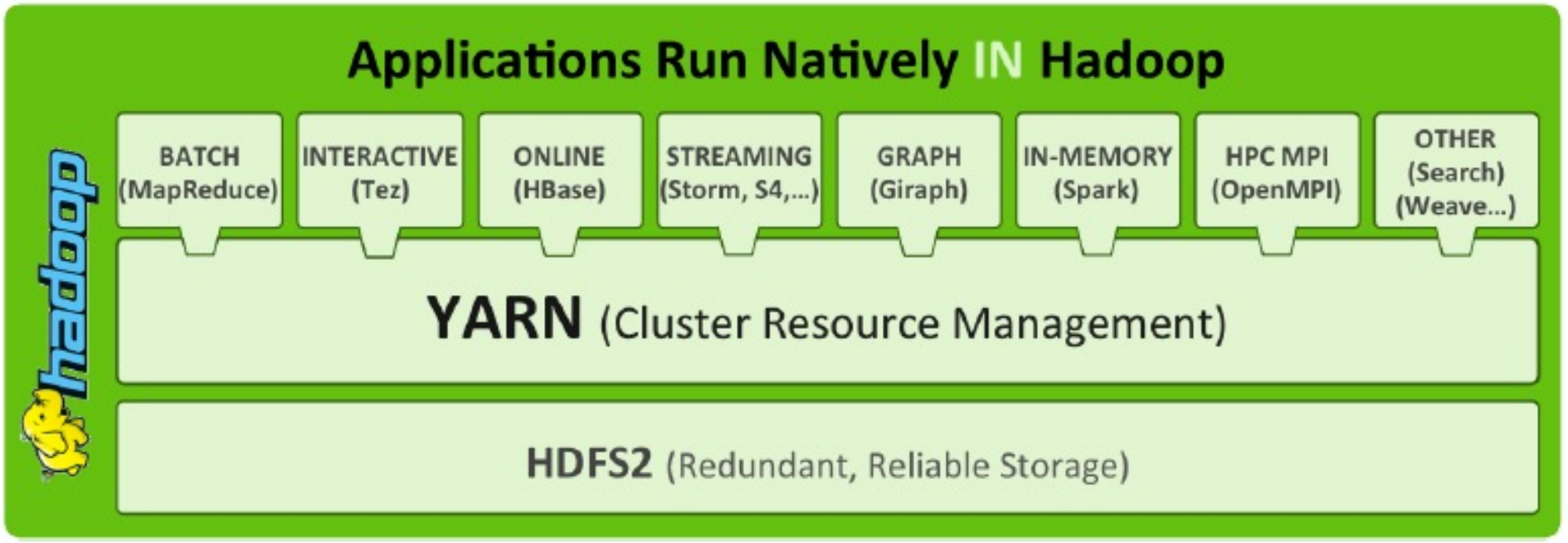
YARN for Hadoop 2.0



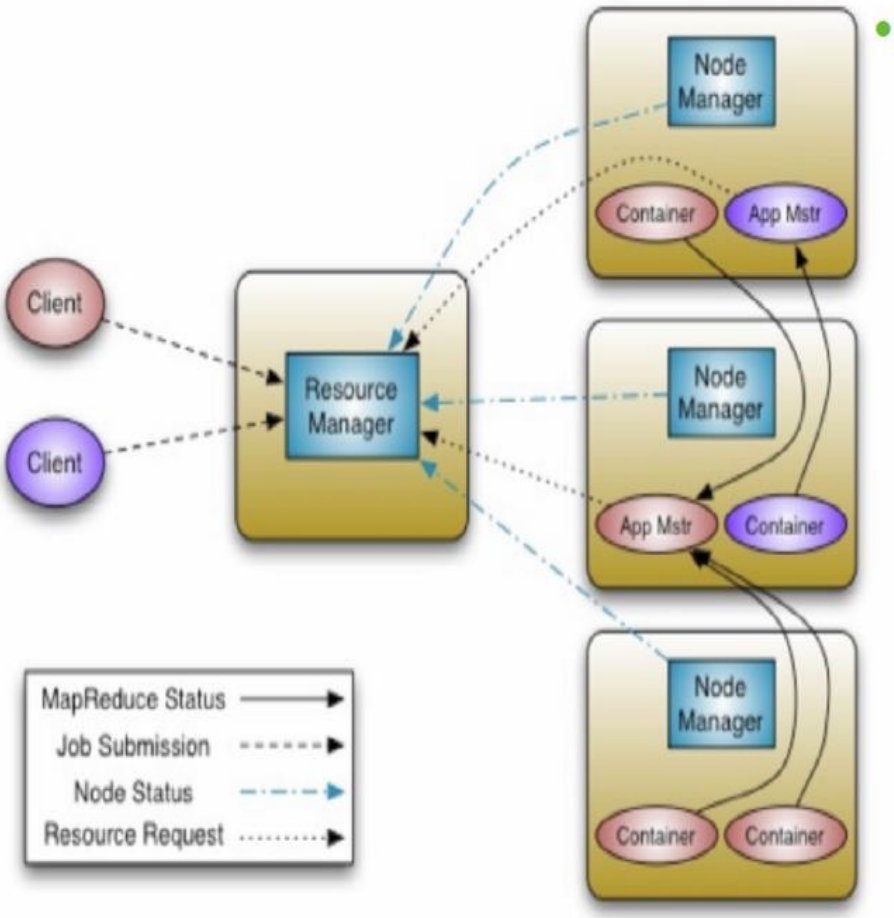
- YARN (Yet Another Resource Negotiator) provides a resource management platform for Cluster to support general Distributed/Parallel Applications/Frameworks beyond the MapReduce computational model.

V. K. Vavilapalli, A. C. Murthy, “Apache Hadoop YARN: Yet Another Resource Negotiator”, in ACM Symposium on Cloud Computing (SoCC) 2013.

A Big Data Processing Stack with YARN



Hadoop2.0/YARN Architectural Overview



- Scalability - Clusters of 6,000-10,000 machines
  - Each machine with 16 cores, 48G/96G RAM, 24TB/36TB disks
  - 100,000+ concurrent tasks
  - 10,000 concurrent jobs

YARN Framework

RescourceManager:

Arbitrates resources among all the applications in the system

NodeManager:

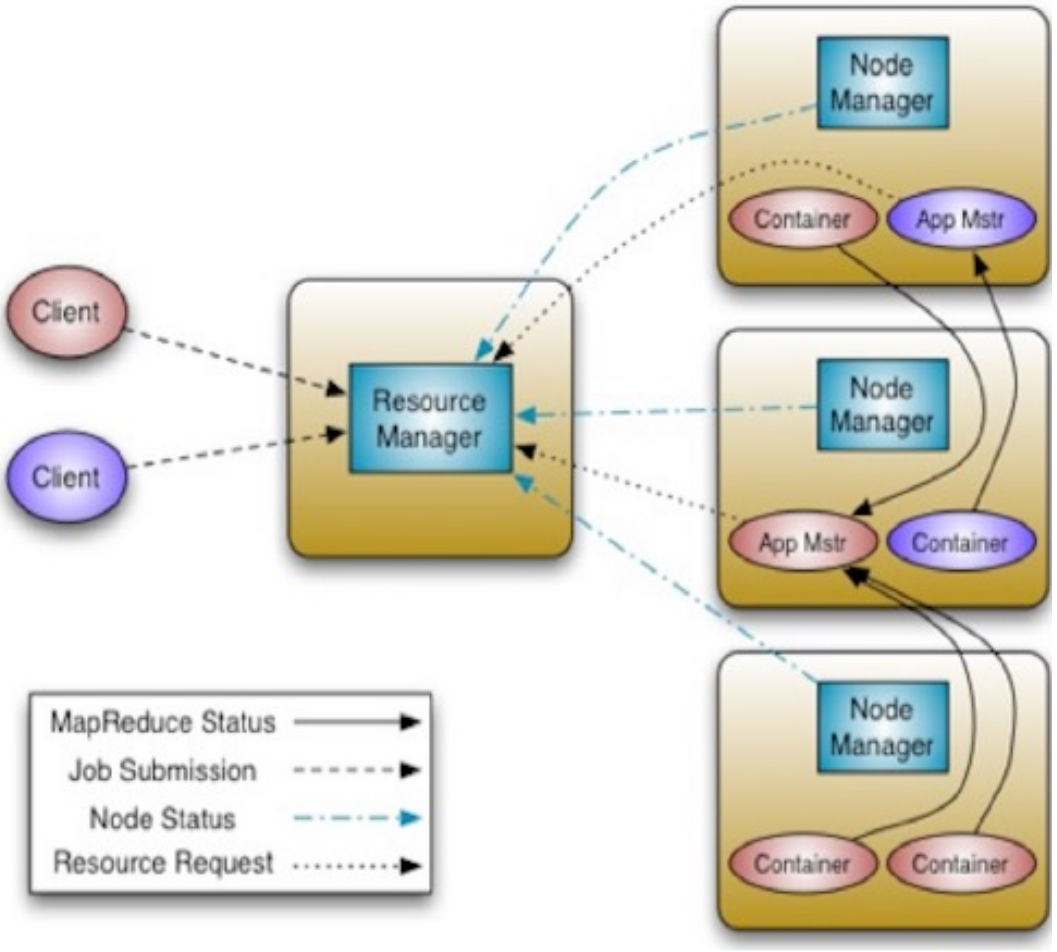
the per-machine slave, which is responsible for launching the applications' containers, monitoring their resource usage

ApplicationMaster:

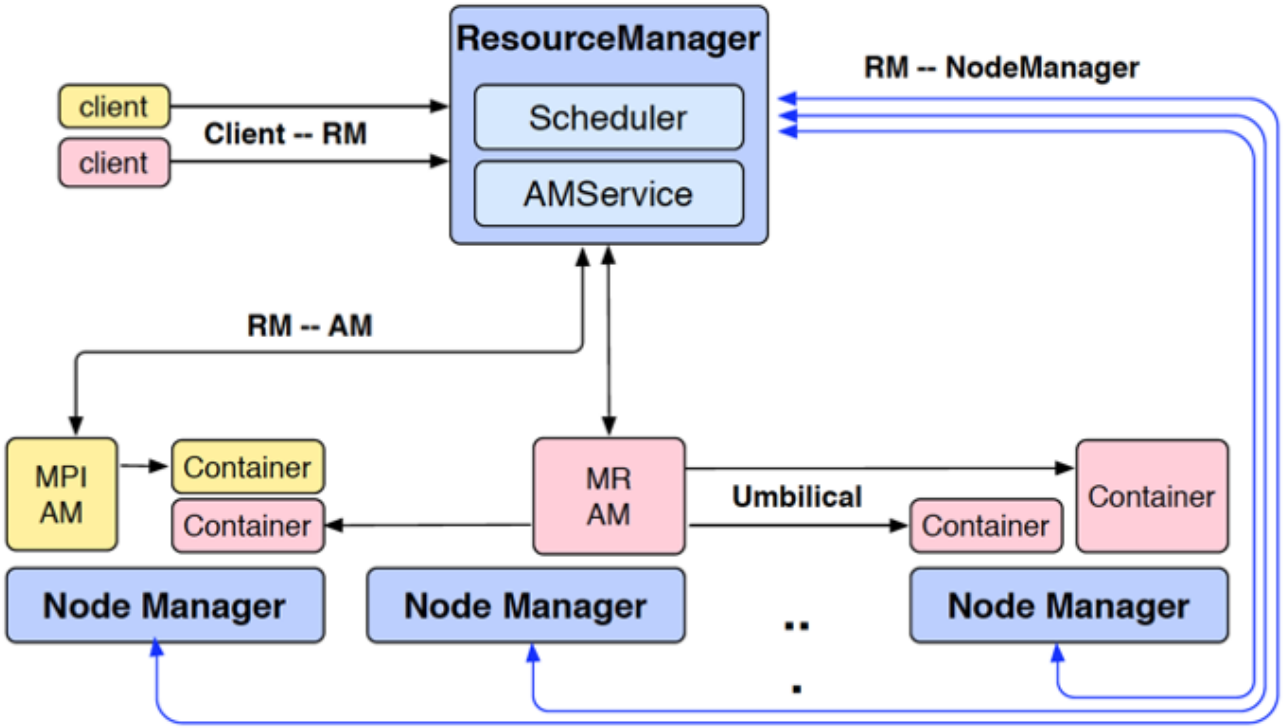
Negatiate appropriate resource containers from the Scheduler, tracking their status and monitoring for progress

Container:

Unit of allocation incorporating resource elements such as memory, cpu, disk, network etc, to execute a specific task of the application (similar to map/reduce slots in MRv1)



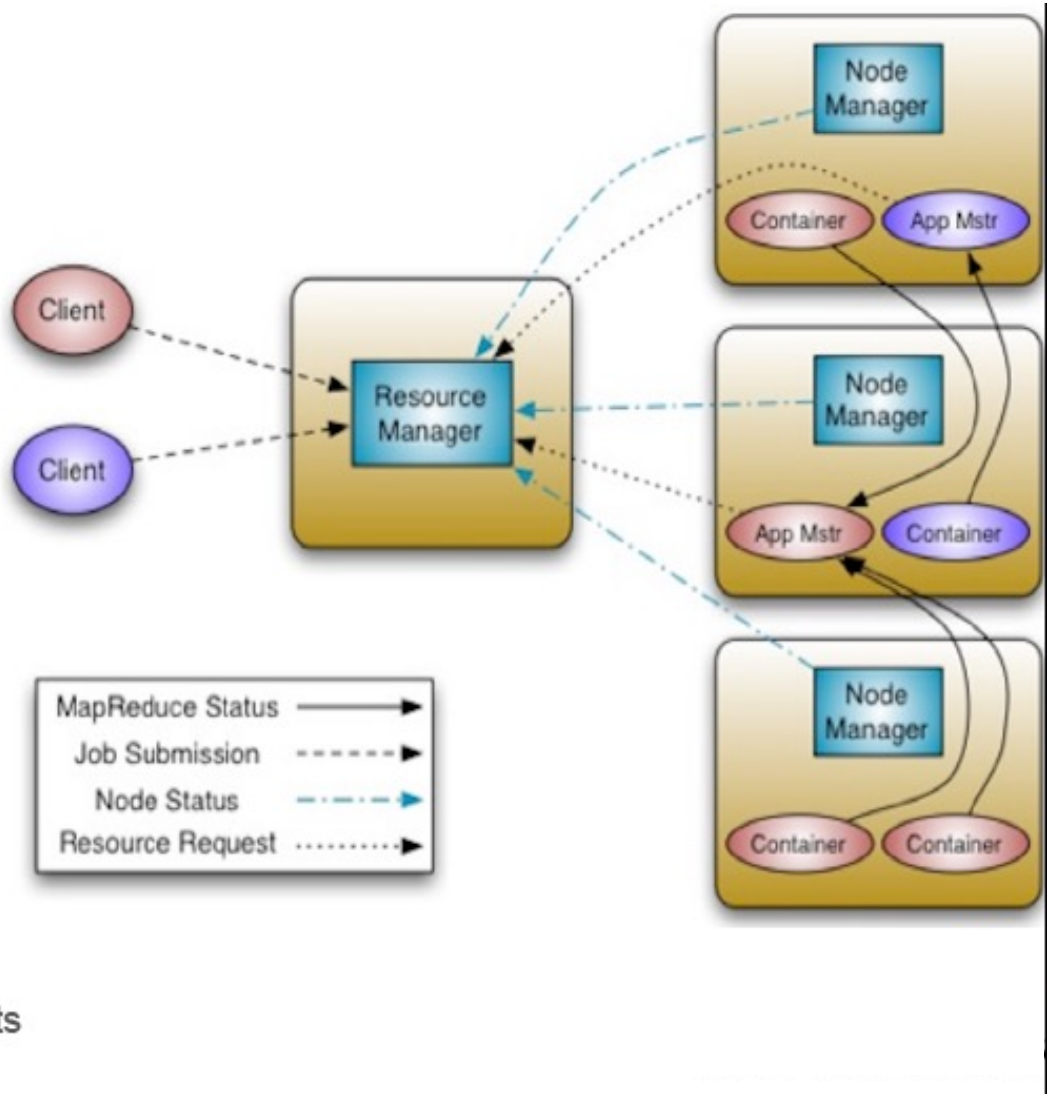
Cluster Resource Management w/ YARN in Hadoop2.0



- Multiple frameworks (Applications) can run on top of YARN to share a Cluster, e.g. MapReduce is one framework (Application), MPI, or Storm are other ones.
- YARN splits the functions of JobTracker into 2 components: resource allocation and job-management (e.g. task-tracking/ recovery):
  - Upon launching, each Application will have its own Application Master (AM), e.g. MR-AM in the figure above is the AM for MapReduce, to track its own tasks and perform failure recovery if needed
  - Each AM will request resources from the YARN Resource Manager (RM) to launch the Application's jobs/tasks (Containers in the figure above) ;
  - The YARN RM determines resource allocation across the entire cluster by communicating with/ controlling the Node Managers (NM), one NM per each machine.

YARN Execution Sequence

1. A client program submits the application
2. ResourceManager allocates a specified container to start the ApplicationMaster
3. ApplicationMaster, on boot-up, registers with ResourceManager
4. ApplicationMaster negotiates with ResourceManager for appropriate resource containers
5. On successful container allocations, ApplicationMaster contacts NodeManager to launch the container
6. Application code is executed within the container, and then ApplicationMaster is responded with the execution status
7. During execution, the client communicates directly with ApplicationMaster or ResourceManager to get status, progress updates etc.
8. Once the application is complete, ApplicationMaster unregisters with ResourceManager and shuts down, allowing its own container process



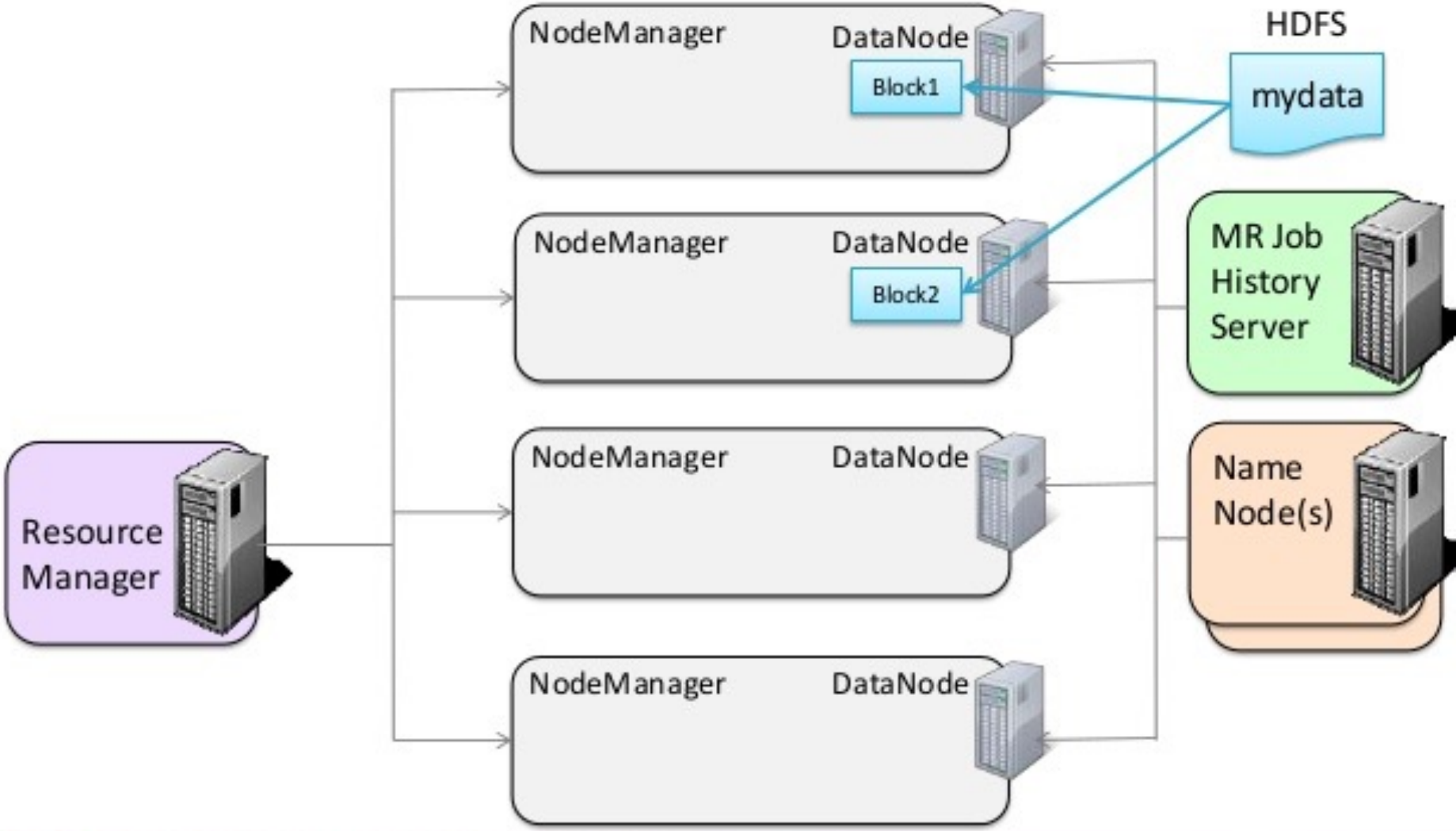
YARN Application Models

- Application Master (AM) per Job
  - Most simple for batch
  - Used by MapReduce (v2)
- Application Master per Session
  - Runs multiple jobs on behalf of the same user
  - Added in Tez ;
  - Also for Spark (one AM per SparkContext, w/ Long-lived enhancement)
- AM as permanent service, supporting Multiple Users
  - Always on, waits around for jobs to come in
  - Used for Impala (with Llama Adapter to support separate-user/queue billing of YARN)

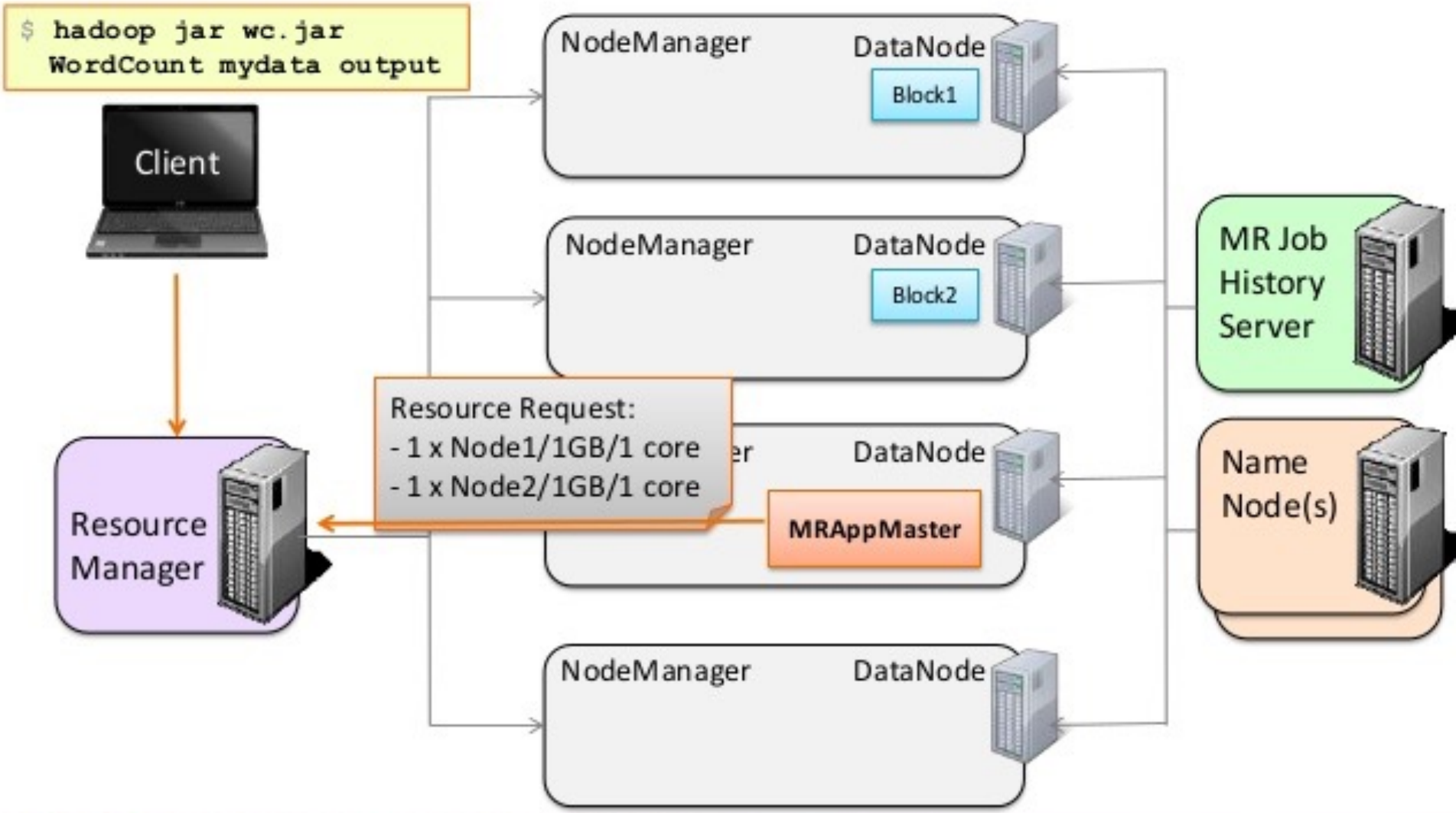
Example: Running MapReduce (v2) on YARN

- Each MapReduce Job has a separate instance of AM
- A Separate MapReduce Job History Server to track MR job history
- YARN runs Shuffle as a persistent, auxiliary service

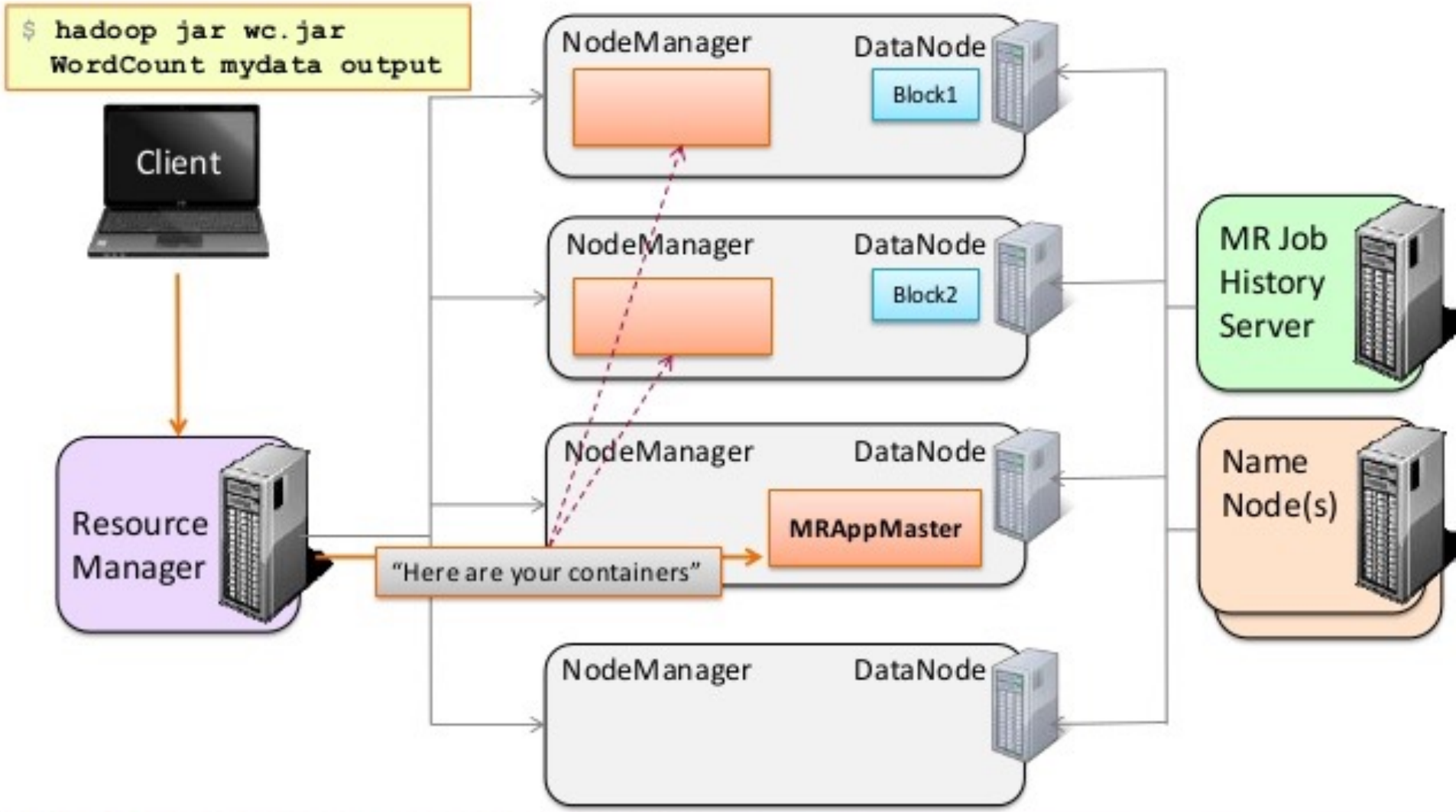
Running a MapReduce Application in MRv2



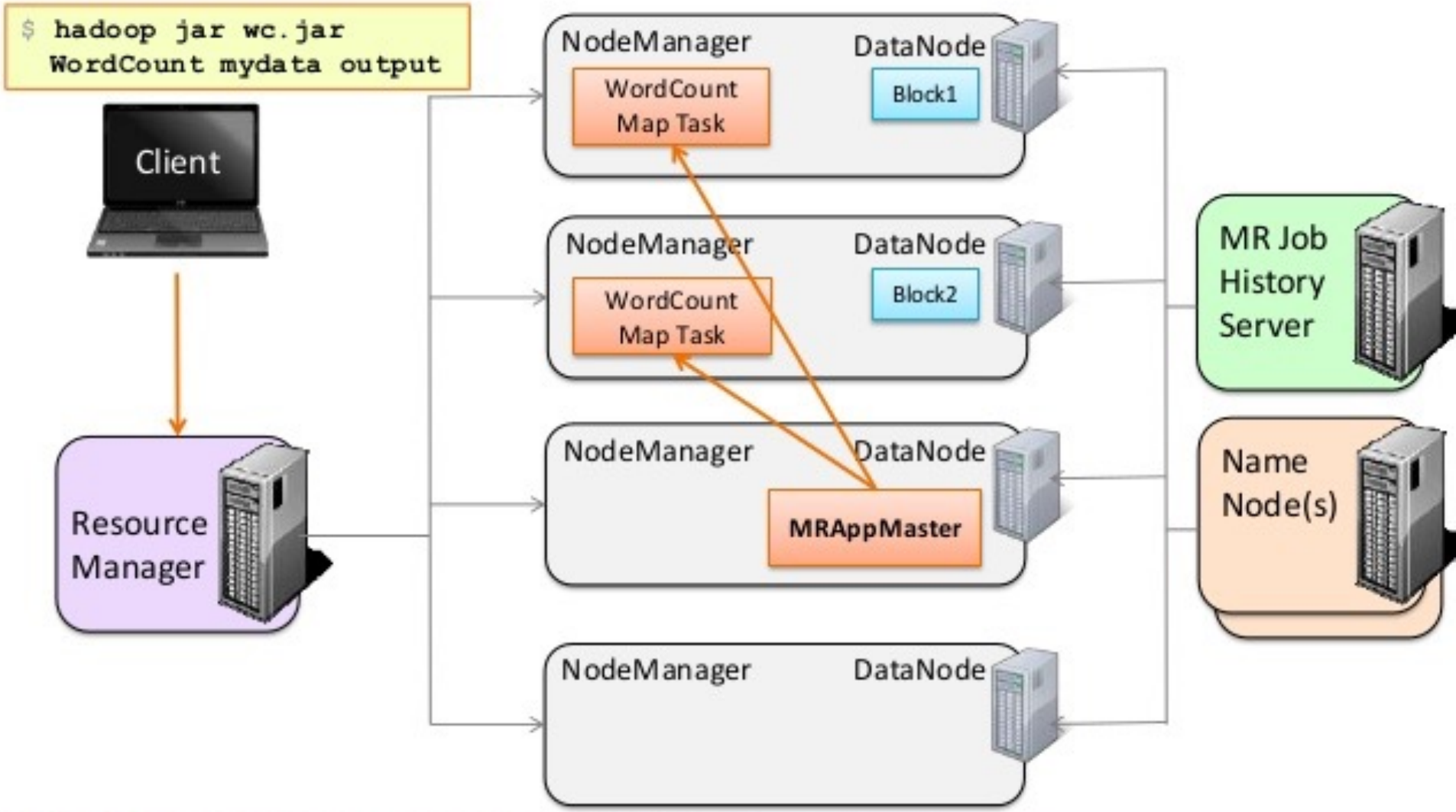
Running a MapReduce Application in MRv2



Running a MapReduce Application in MRv2

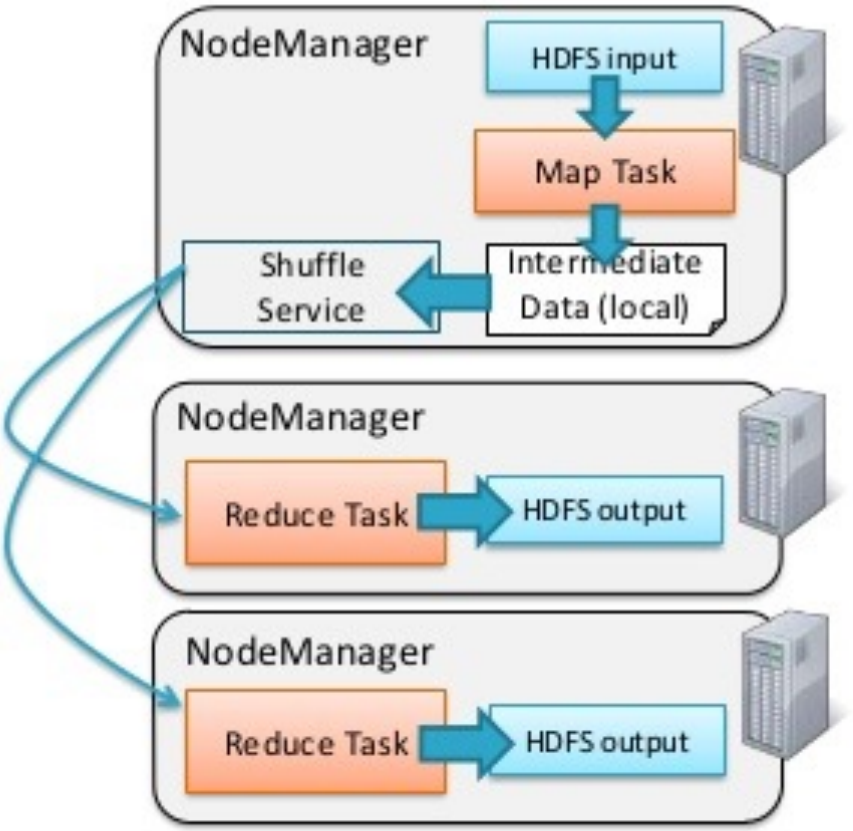


Running a MapReduce Application in MRv2



The MapReduce Framework on YARN

- **In YARN, Shuffle is run as an auxiliary service**
  - Runs in the NodeManager JVM as a persistent service

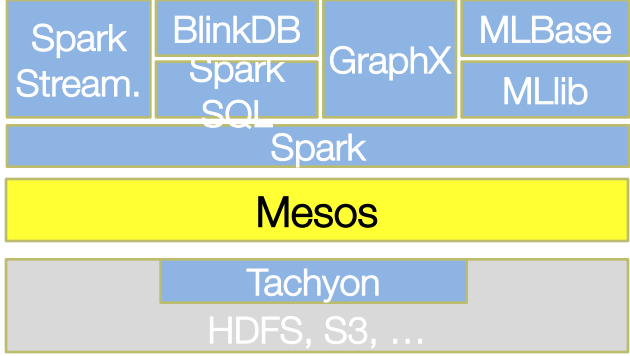


Hadoop 2.0 vs. Hadoop1.0

- Hadoop 2.0 includes YARN's Multi-tenant Support for different Big Data Processing Frameworks
- YARN Fault Tolerance and Availability
  - Resource Manager
    - No single point of failure – state saved in ZooKeeper
    - Application Masters are restarted automatically on RM restart
  - Application Master
    - Optional failover via application-specific checkpoint
    - MapReduce applications pick up where they left off via state saved in HDFS
- Wire Compatibility
  - Protocols are wire-compatible
  - Old clients can talk to new servers
  - Rolling upgrades
- Besides YARN, Hadoop 2.0 also supports High Availability and Federation
  - High Availability takes away the Single Point of failure from HDFS Namenode and introduces the concept of the QuorumJournalNodes to sync edit logs between active and standby Namenodes
  - Federation allows multiple independent namespaces (private namespaces, or Hadoop as a service)

Apache Mesos

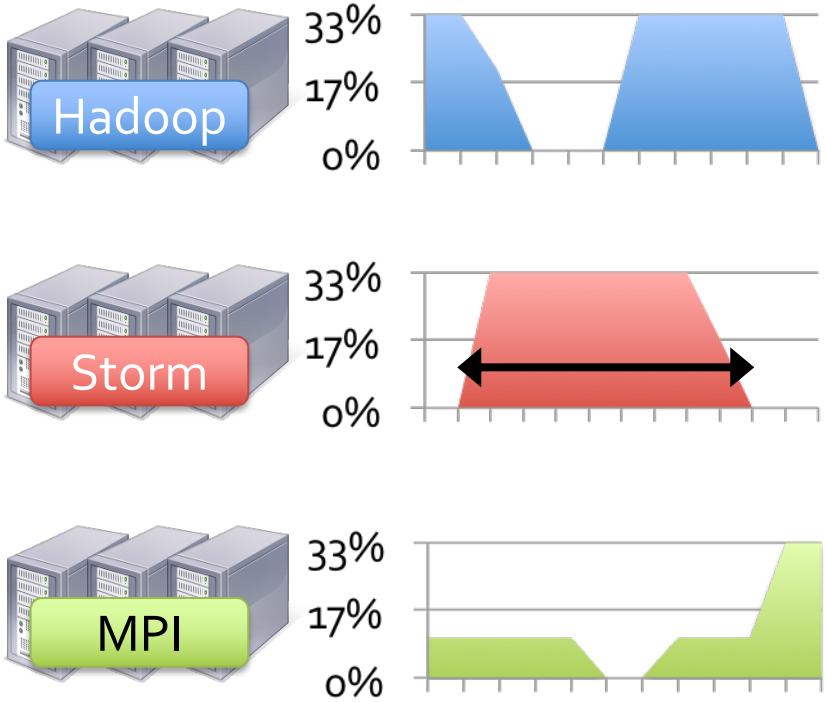
(http://mesos.apache.org)



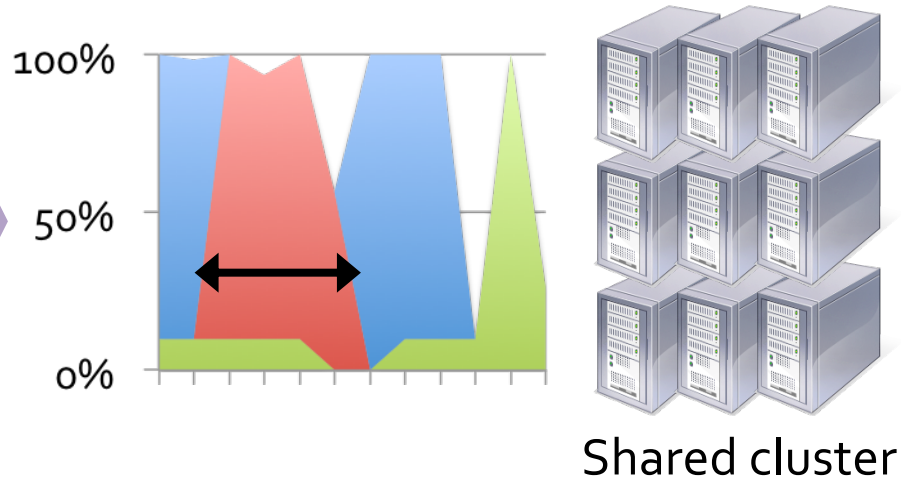
- Another competing Cluster Resource Management platform
- Enable multiple frameworks to share same cluster resources (e.g., MapReduce, Storm, Spark, HBase, etc)
- Originated from UCBerkeley's BDAS project ;
  - B. Hindman et al, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center", Usenix NSDI 2011.
- Hardened via Twitter's large scale in-house deployment
  - 6,000+ servers,
  - 500+ engineers running jobs on Mesos
- Third party Mesos schedulers
  - AirBnB's Chronos ; Twitter's Aurora
- Mesospehere: startup to commercialize Mesos

Motivation of Mesos

Previously: Static partitioning of a cluster among different big data processing frameworks



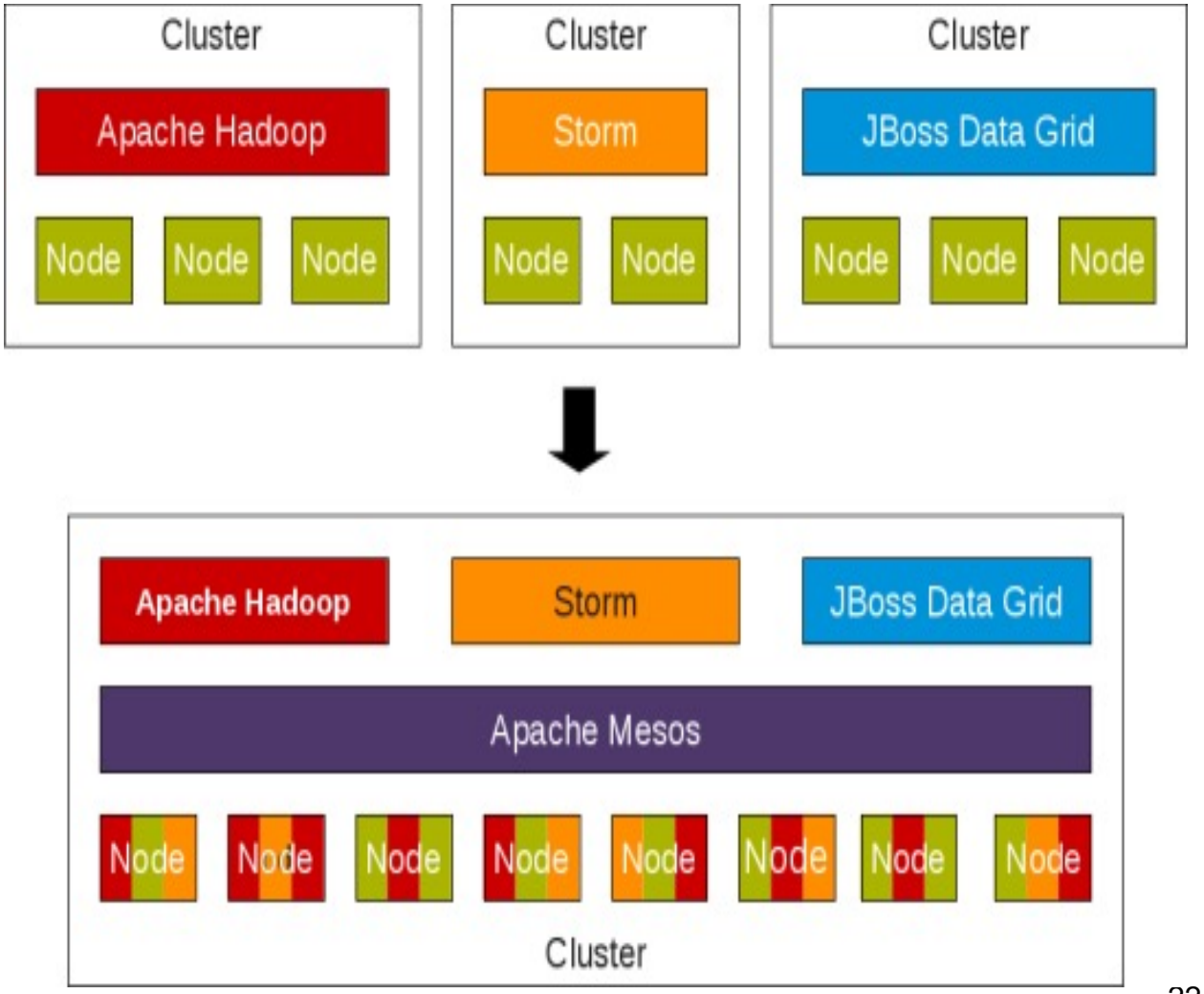
Mesos aims to achieve dynamic sharing of cluster across different frameworks



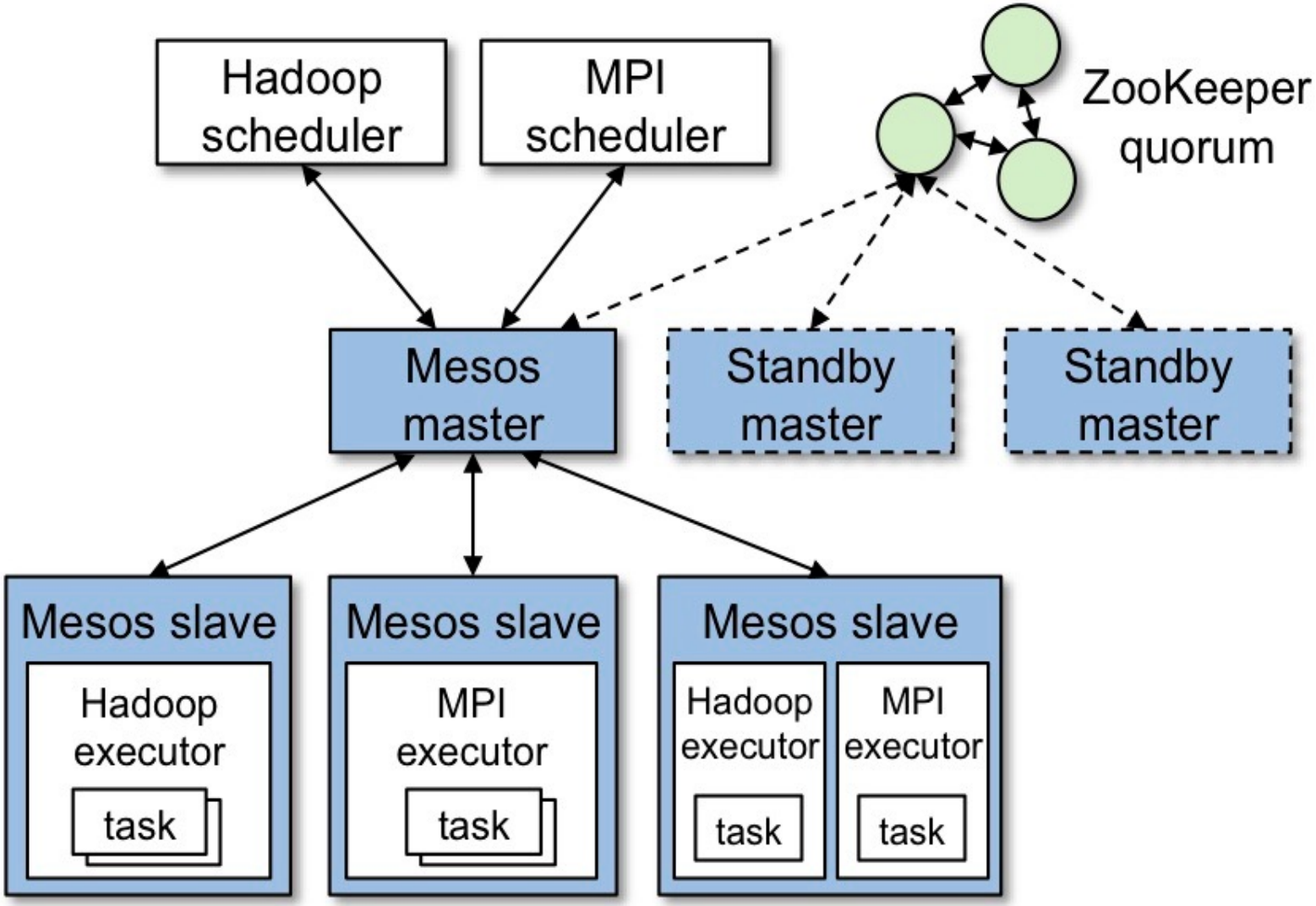
- Hard to fully utilize machines (e.g., X GB RAM & Y CPUs)
- Hard to scale elastically (to take advantage of statistical multiplexing)
- Hard to deal with failures

Mesos as a Data-Center “Kernel”

- Like YARN, Mesos provides a Node Abstraction of the entire Cluster
- Like YARN, Mesos is a common resource sharing layer over which diverse frameworks can run



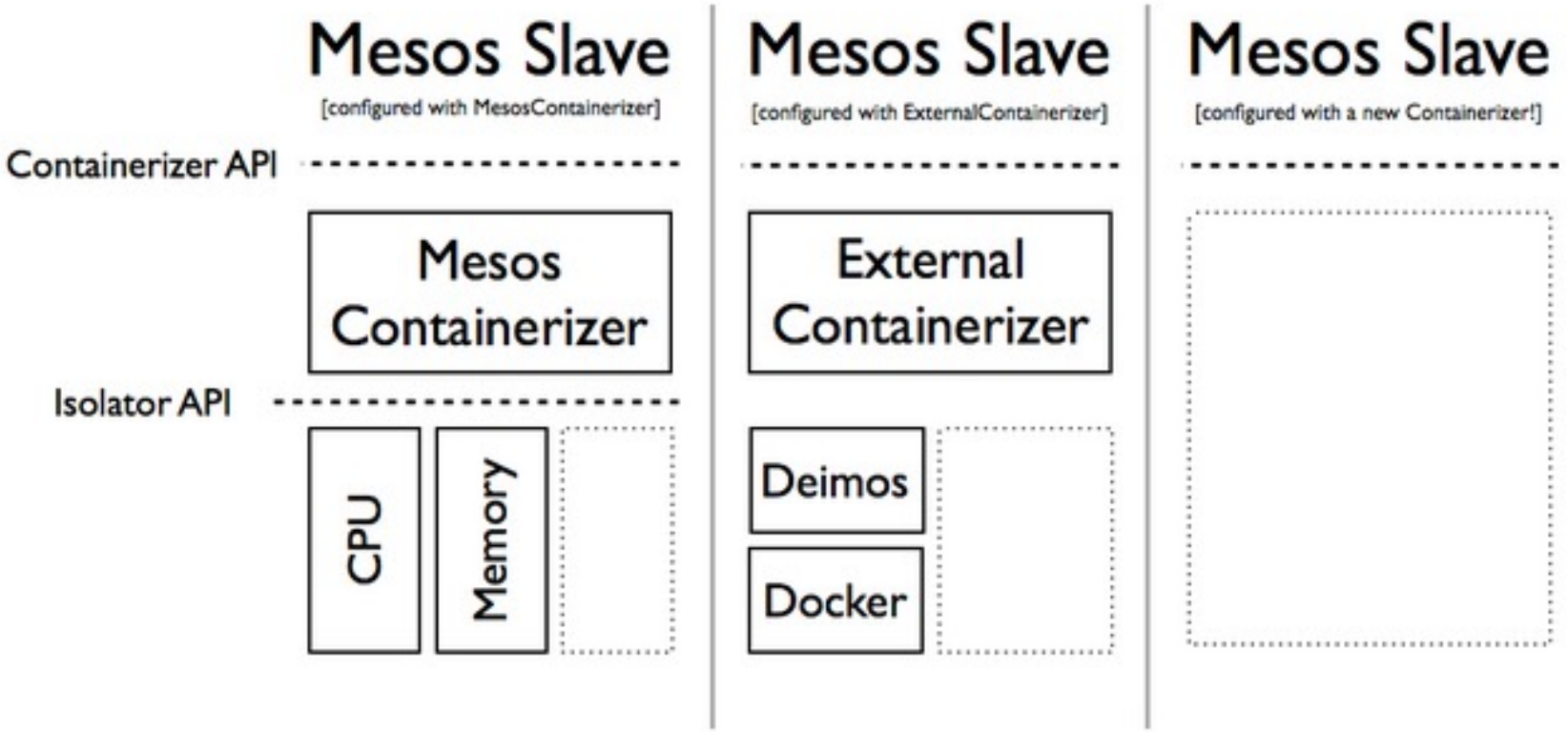
System Architecture of Mesos



Framework Isolation

- Mesos uses OS isolation mechanisms, such as Linux containers and Solaris projects
- Containers currently support CPU, memory, IO and network bandwidth isolation
- Not perfect, but much better than no isolation

Mesos' use of Container Technology

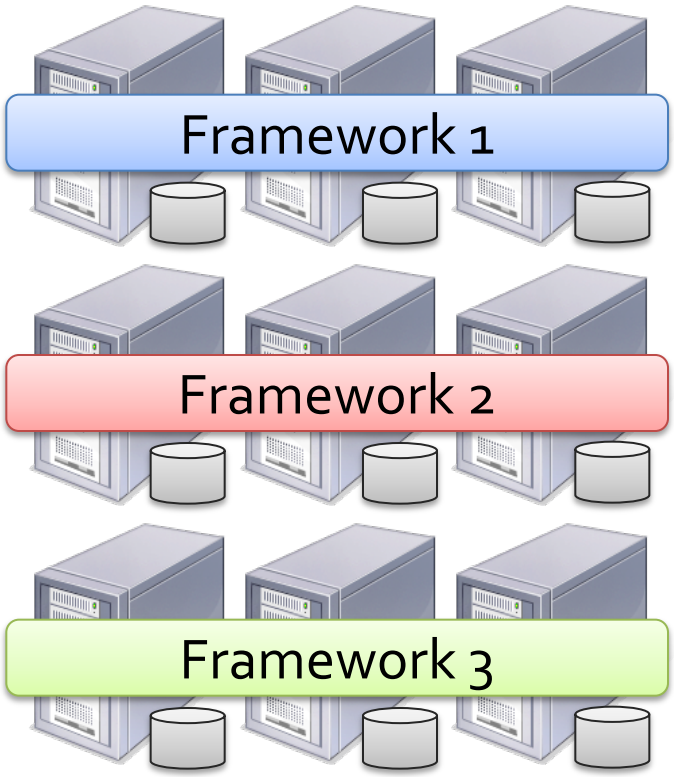


Design Elements

- Fine-grained sharing:
  - Allocation at the level of *tasks* within a job
  - Improves utilization, latency, and data locality
- Resource offers:
  - Simple, scalable application-controlled scheduling mechanism

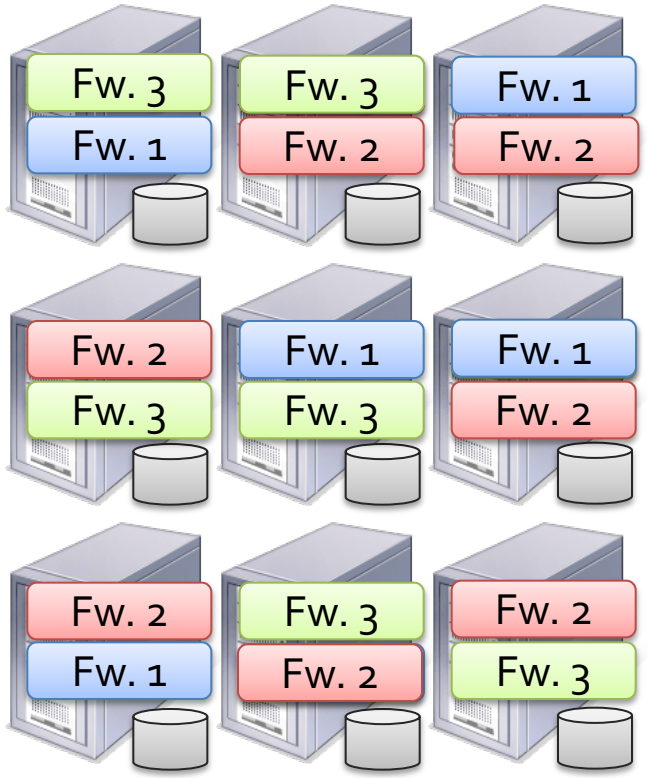
Element 1: Fine-Grained Sharing

Coarse-Grained Sharing (HPC):



Storage System (e.g. HDFS)

Fine-Grained Sharing (Mesos):



Storage System (e.g. HDFS)

+ Improved utilization, responsiveness, data locality

Element 2: Resource Offers

- Option: Global scheduler
  - Frameworks express needs in a specification language, global scheduler matches them to resources

+ Can make optimal decisions

– Complex: language must support all framework needs

– Difficult to scale and to make robust

– Future frameworks may have unanticipated needs

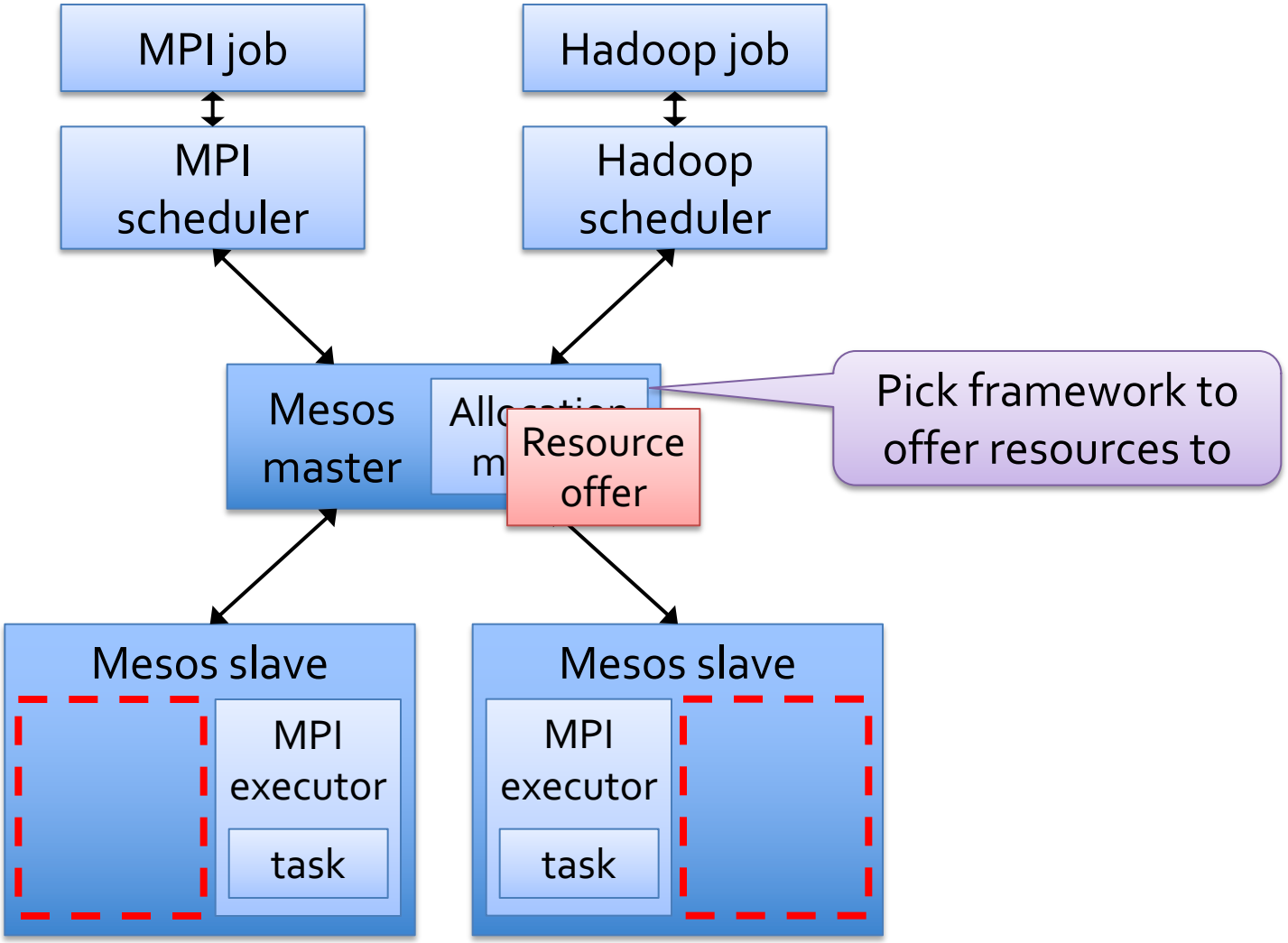
Element 2: Resource Offers

- Mesos: Resource offers
  - Offer available resources to frameworks, let them pick which resources to use and which tasks to launch

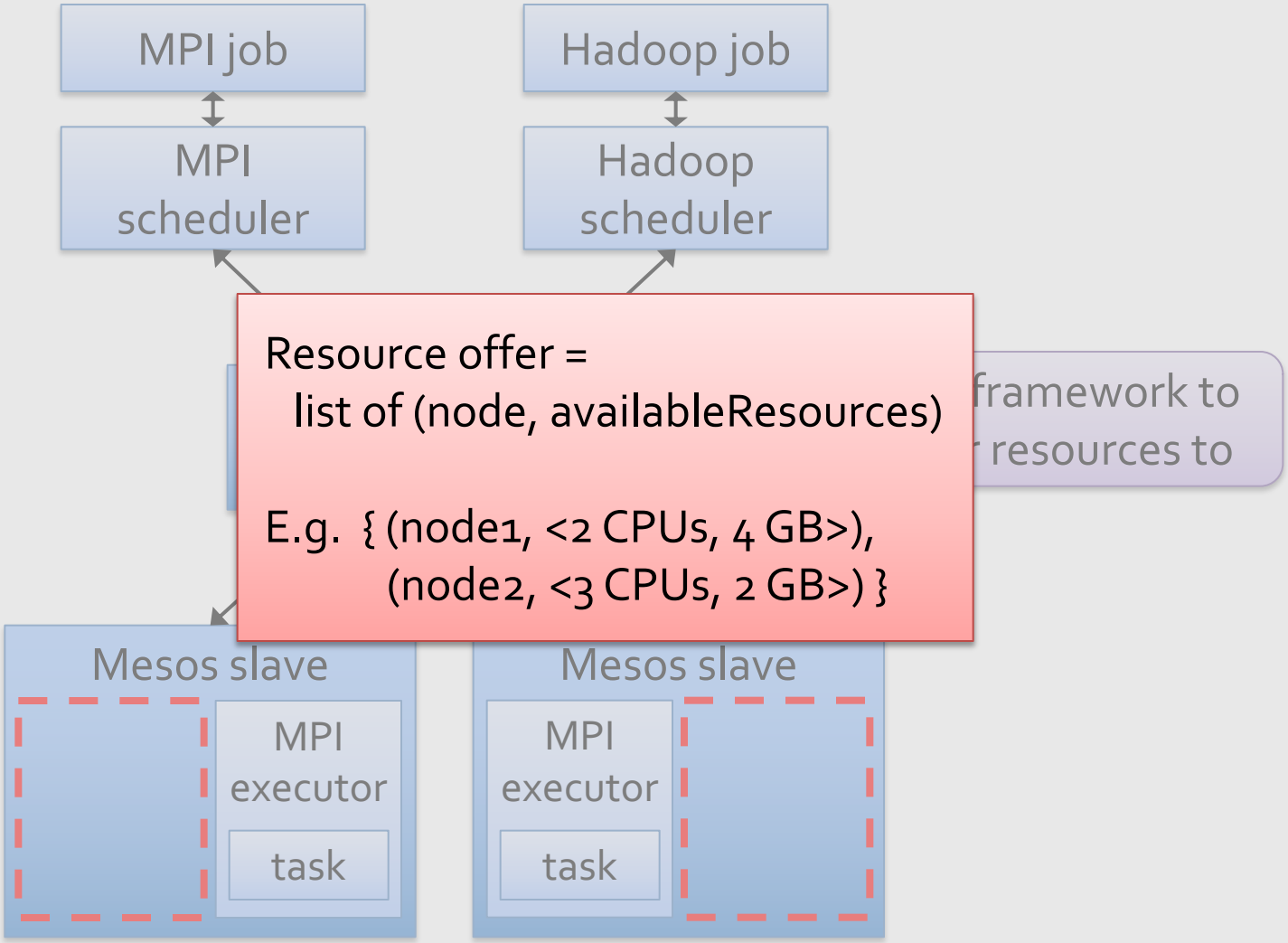
  \+ Keep Mesos simple, let it support future frameworks

  – Decentralized decisions might not be optimal

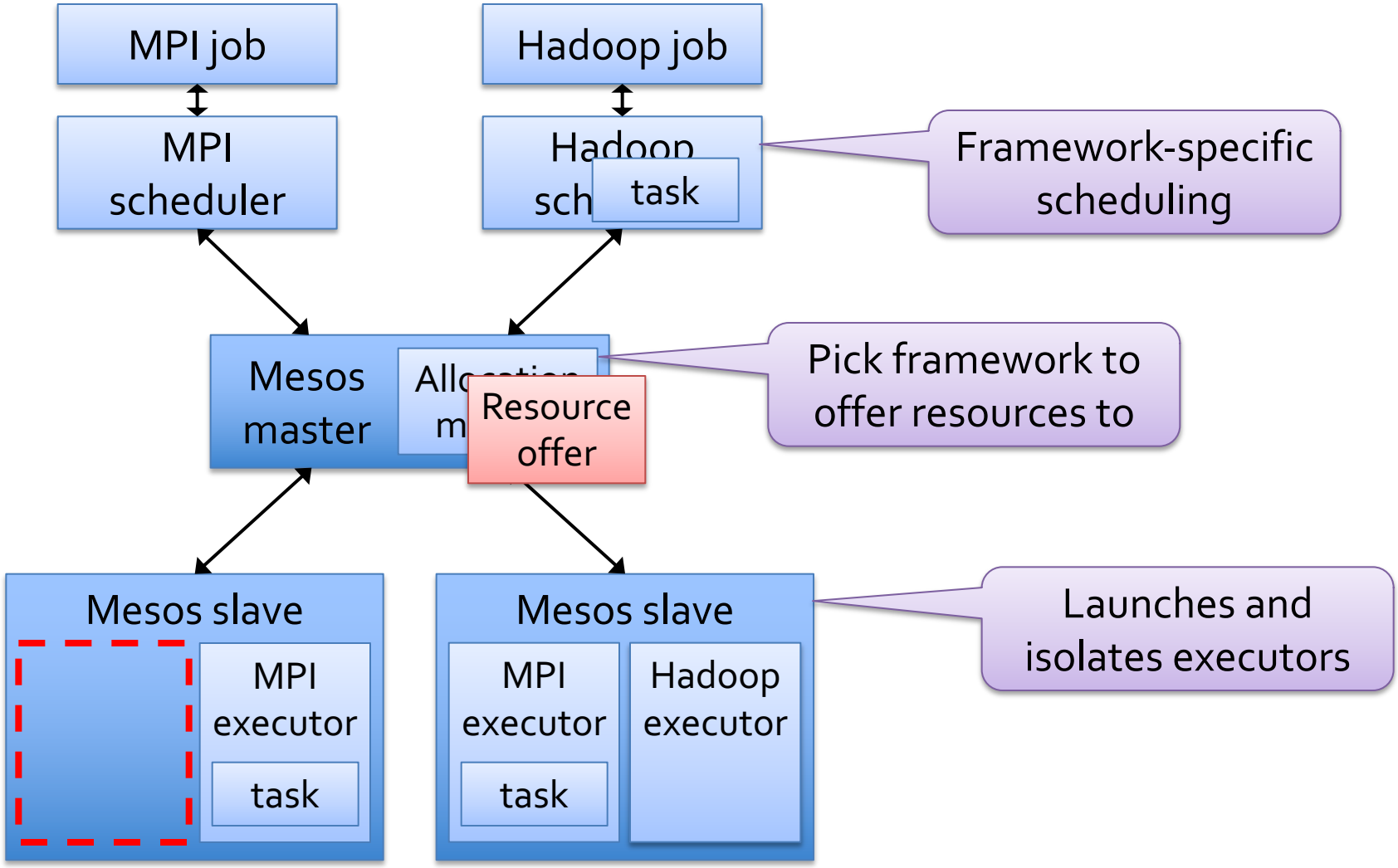
Mesos Architecture



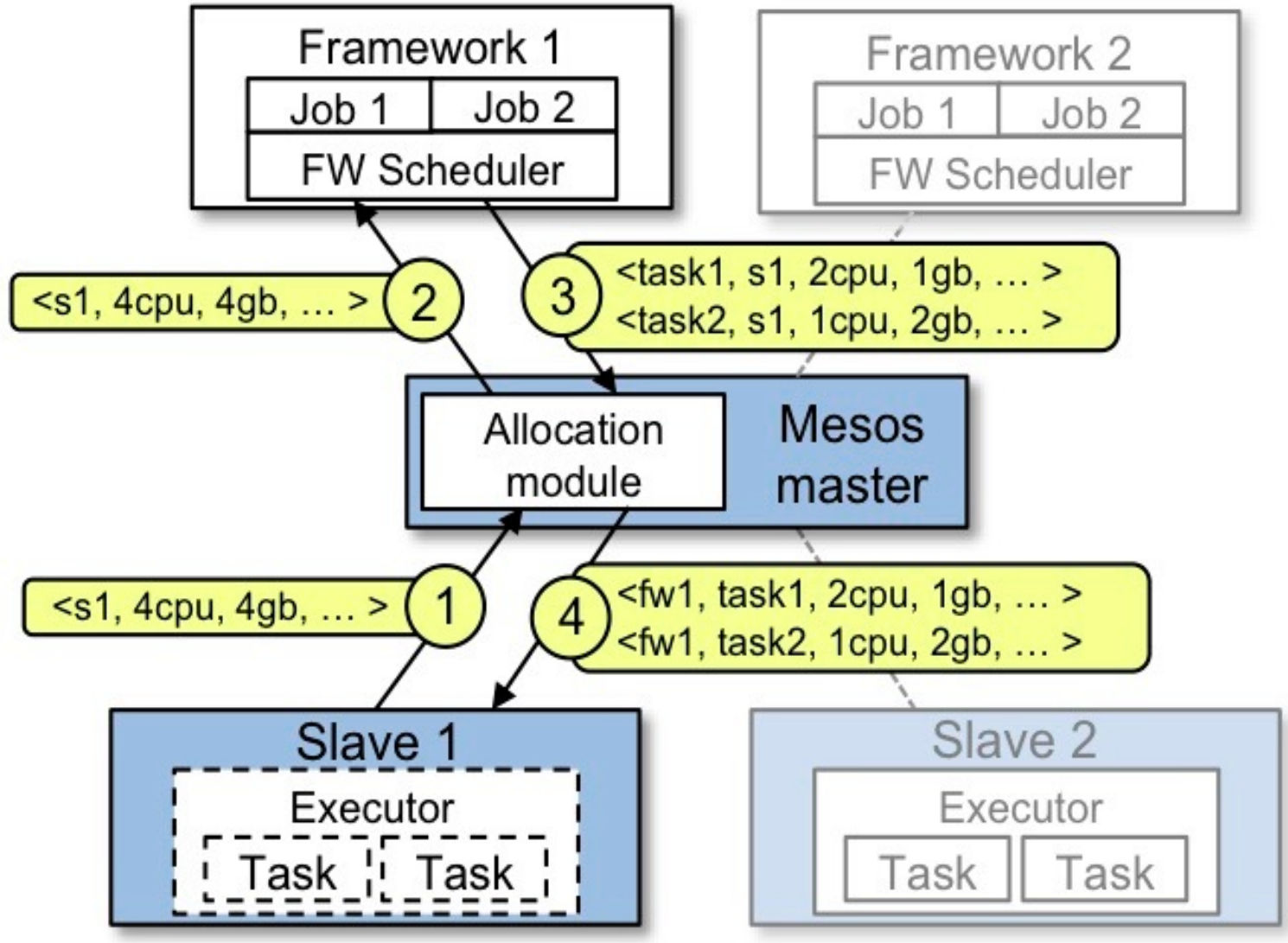
Mesos Architecture



Mesos Architecture



Another Resource Offering Example



Optimization: Filters

- Let frameworks short-circuit rejection by providing a predicate on resources to be offered
  - » E.g. "nodes from list L" or "nodes with > 8 GB RAM"
  - » Could generalize to other hints as well
- Ability to reject still ensures *correctness* when needs cannot be expressed using filters

Revocation

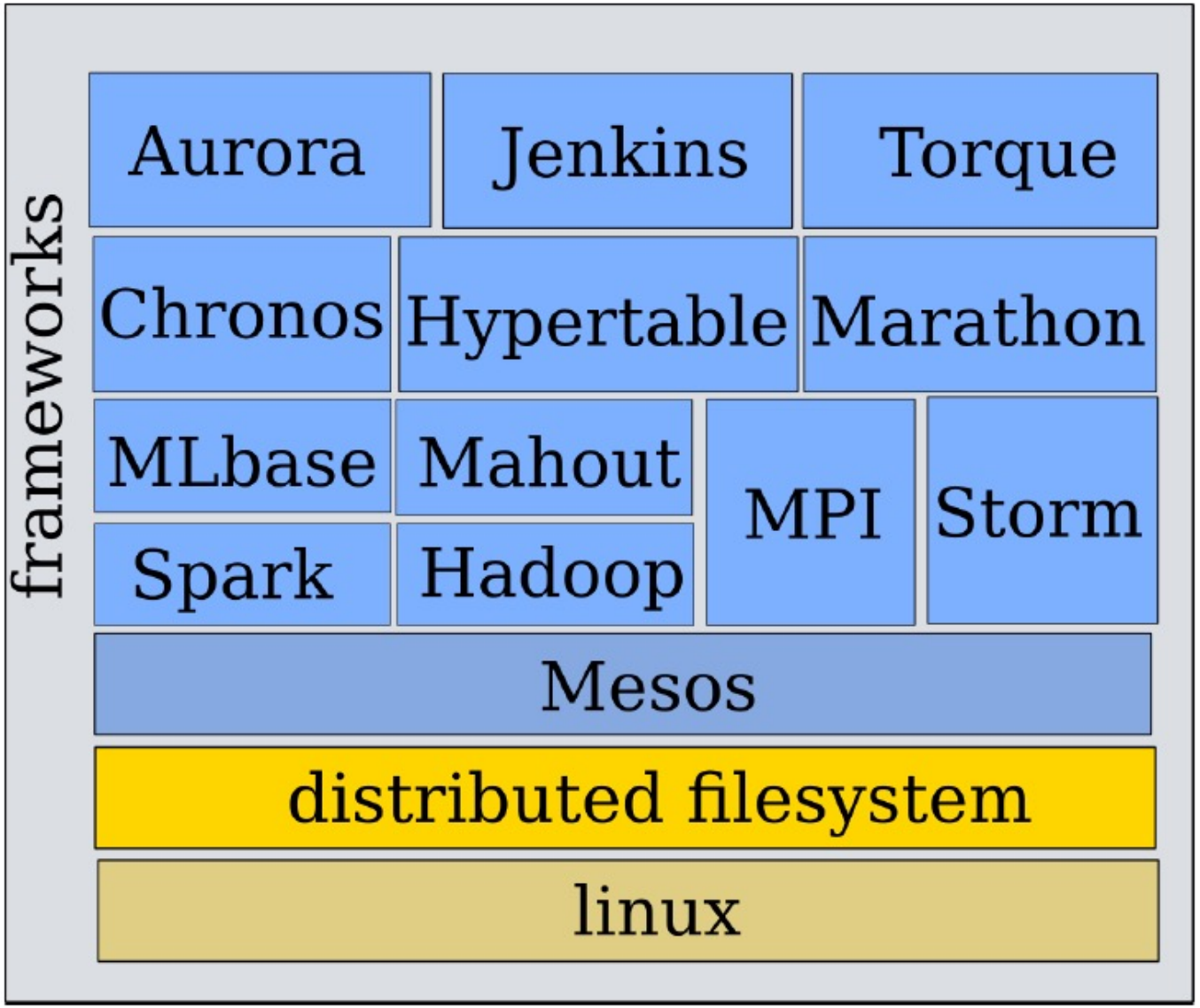
- Mesos allocation modules can revoke (kill) tasks to meet organizational SLOs
- Framework given a grace period to clean up
- "Guaranteed share" API lets frameworks avoid revocation by staying below a certain share

Mesos API

Scheduler Callbacks	Scheduler Actions
resourceOffer(offerId, offers) offerRescinded(offerId) statusUpdate(taskId, status) slaveLost(slaveId)	replyToOffer(offerId, tasks) setNeedsOffers(bool) setFilters(filters) getGuaranteedShare() killTask(taskId)

Executor Callbacks	Executor Actions
launchTask(taskDescriptor) killTask(taskId)	sendStatus(taskId, status)

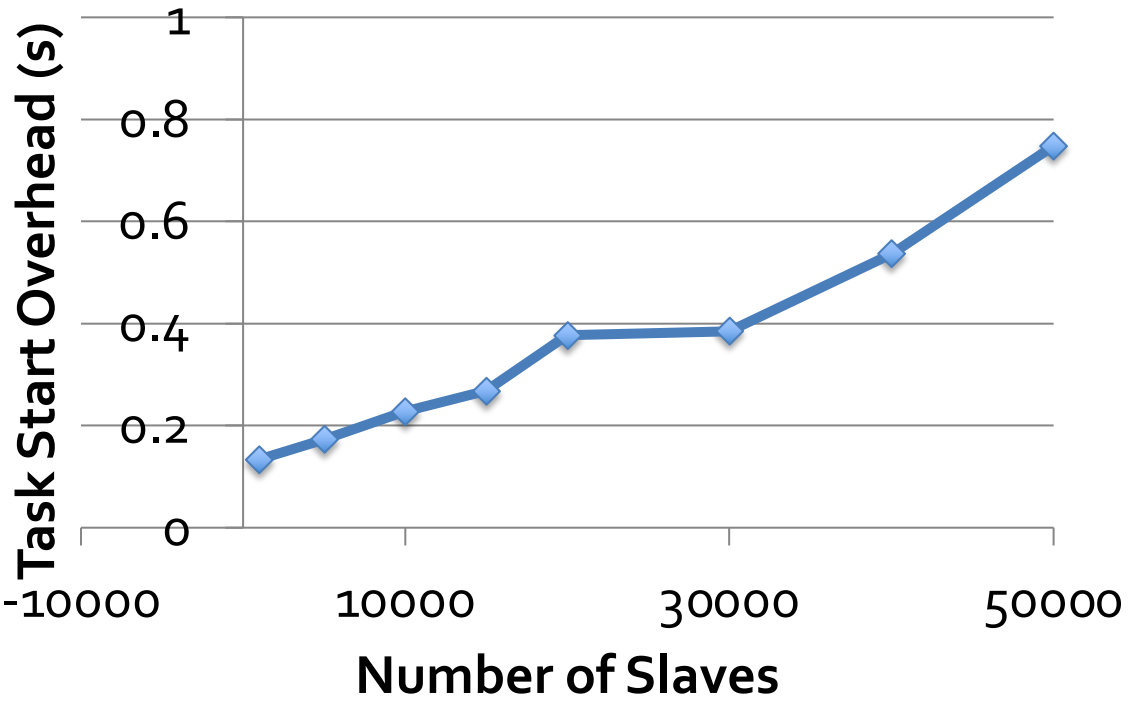
A Big Data Processing Stack w/ Mesos



Scalability

Mesos only performs *inter-framework* scheduling (e.g. fair sharing), which is easier than intra-framework scheduling

Result:
Scaled to 50,000 emulated slaves, 200 frameworks, 100K tasks (30s len)



Fault Tolerance

- Mesos master has only *soft state*: list of currently running frameworks and tasks
- Rebuild when frameworks and slaves re-register with new master after a failure

Result: fault detection and recovery in ~10 sec

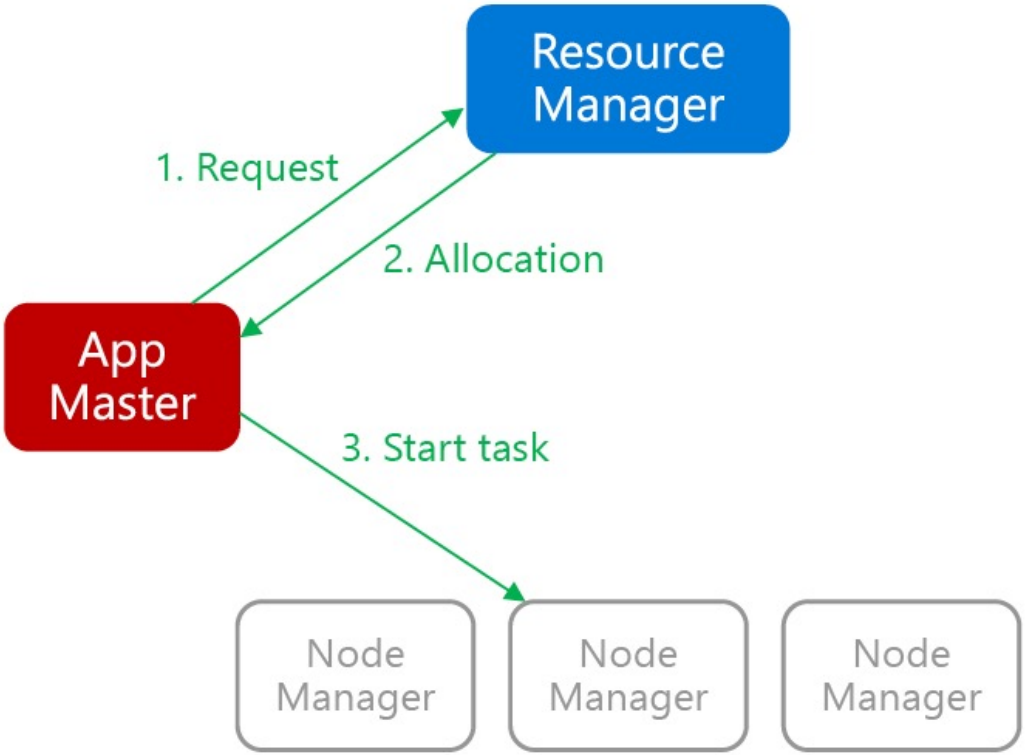
Mesos Implementation Statistics

- 20,000 lines of C++
- Master failover using ZooKeeper
- Frameworks ported: Hadoop1.0, MPI, Storm, etc
- Specialized framework: Spark, for iterative jobs (up to 20 × faster than Hadoop)
- Open source under Apache license

Other Schedulers/ Resource Management Platforms for Big Data Processing Clusters

Approach 1: Centralized Resource Management

[YARN, Mesos, Omega, Borg]



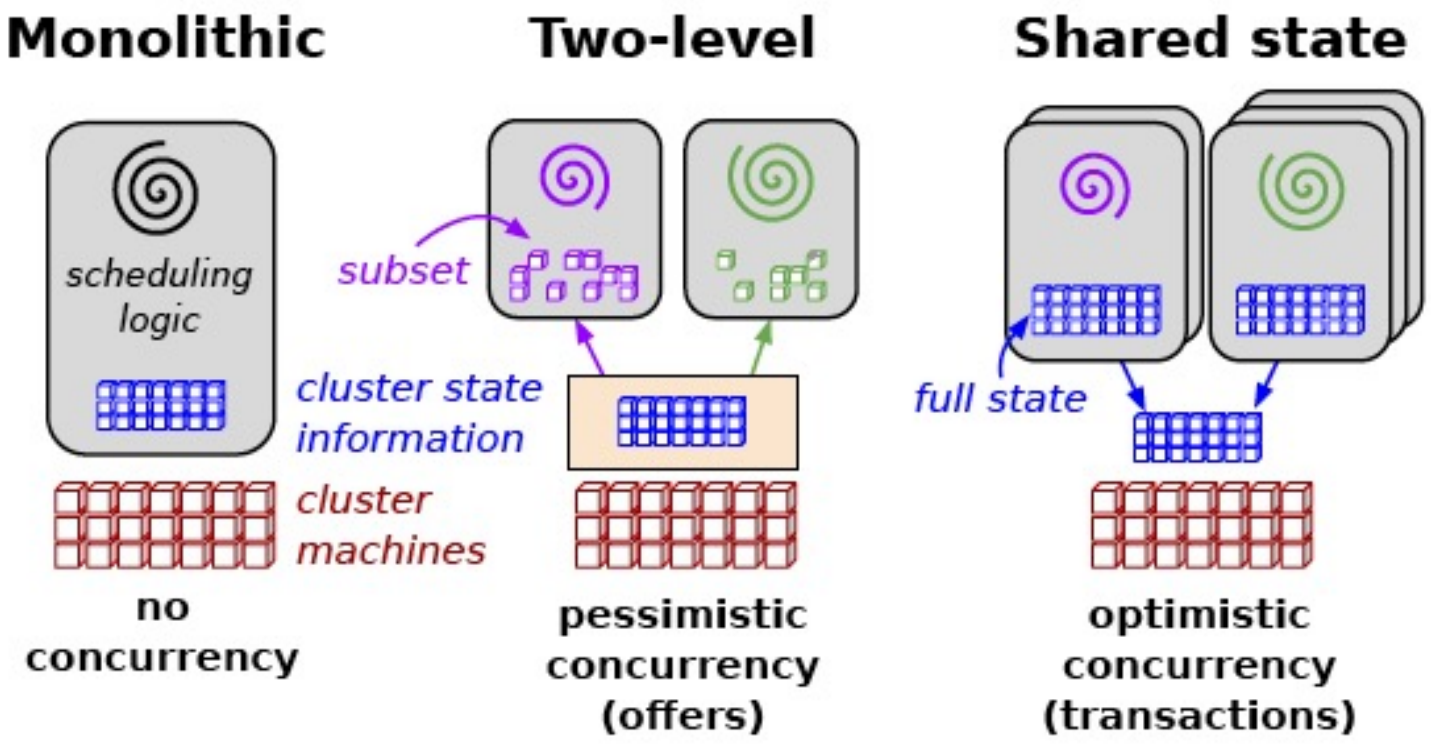
- All scheduling decisions go through the central RM
- The RM resolves all conflicts and guarantees resources to applications

M. Schwarzkopf, A. Konwinski, M.Abd-El-Malek, J. Wilkes, “Omega: flexible, scalable schedulers for large compute clusters,” Eurosys 2013

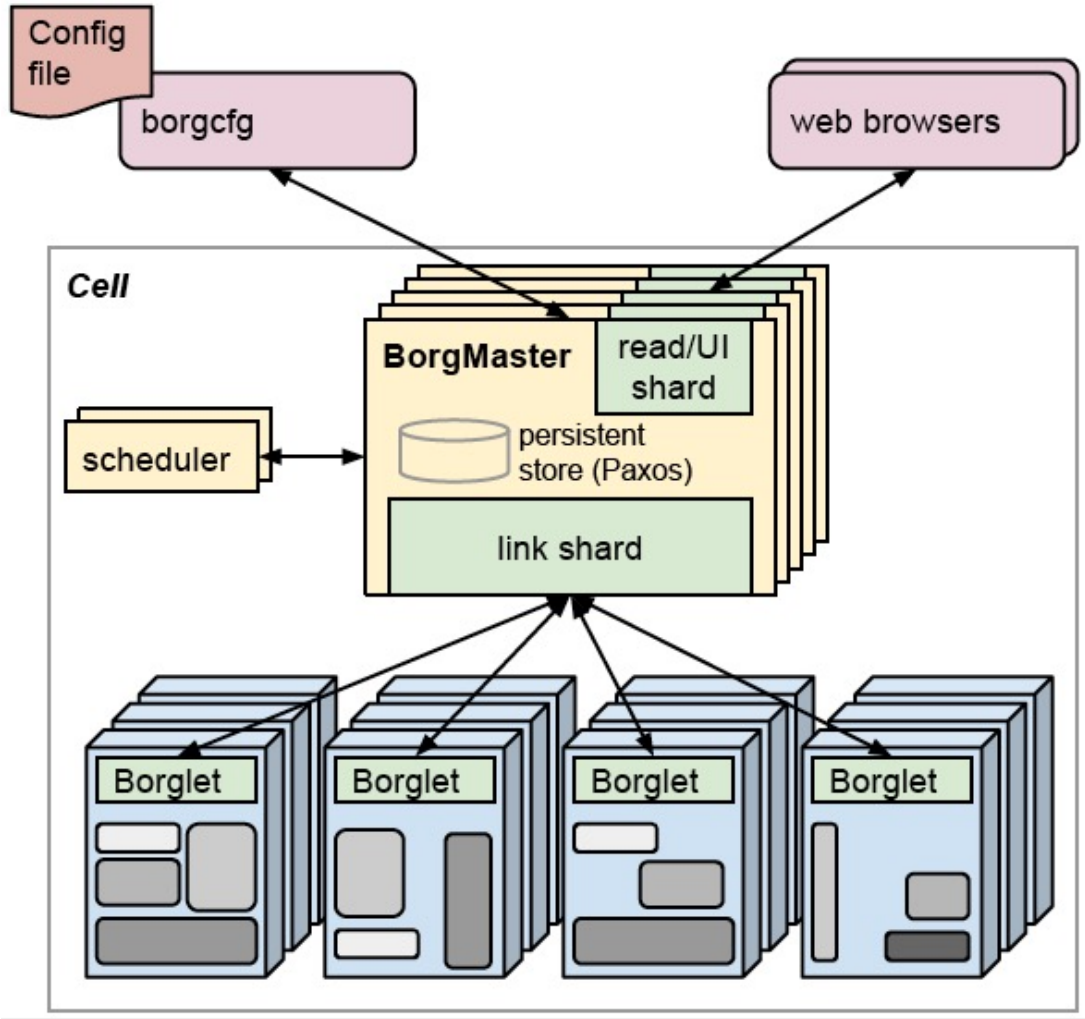
A. Verma, L. Pedrosa, “Large-scale cluster management at Google with Borg”, Eurosys 2015

Design Options for Centralized Resource Management:

Monolithic[Hadoop1.0, YARN] vs.Two-level[Mesos] vs.Shared-state[Omega, Borg



High-level Architecture of Google's Borg



A. Verma, L. Pedrosa, "Large-scale cluster management at Google with Borg", Eurosys 2015

Borg Architecture - Borgmaster

- Each cell contains a Borgmaster
- Each Borgmaster consists of 2 processes:
  - Main Borgmaster process
  - Scheduler
- Multiple replicas of each Borgmaster
- Role of (elected leader) Borgmaster:
  - submission of job, termination of any of job's task

Borg Architecture - Borglet

- Local Borg agent on every cell
  - starts/stops/restarts tasks
  - Manages local resources
  - Rolls over debug logs
- Polled by Borgmaster to get machine's current state
- If a Borglet does not respond to several poll messages, it is marked as down
  - Tasks re-distributed
  - If communication is restored, Borgmaster tells Borglet to kill rescheduled tasks

How does Borg work?

- Users submit “jobs”
  - Each “job” contains 1+ “task” that all run the same program/binary
  - Runs inside containers (not VMs as it would cost higher latency)
- Each “job” runs on one “cell”
  - A “cell” is a set of machines that run as one unit

- Two main types of jobs:
  - “Prod” job : long-running server jobs, higher priority
  - “Non-prod” job : quick batch jobs, lower priority

How does Borg work?

- **Allocs:**
  - Reserved set of resources in one machine
  - Can run multiple instances of a task, different tasks from many jobs, or future tasks
- **Priority and quota:**
  - Each job has a priority
  - Preemption disallowed between “prod” jobs.
  - Quota refers to vector of resource quantities for period of time
- **Support for naming and monitoring**

Borg Architecture - Scheduling

- Borgmaster adds new jobs to a pending queue after recording it in the Paxos store
- A scheduler (primarily operates on tasks) scans and assigns tasks to machines
  - Feasibility checking
  - Scoring
- E-PVM vs “best-fit”
  - E-PVM leaves headroom for load-spikes but has increased fragmentation
  - Best-fit fills machines as tightly as possible, but hurts “bursty loads”
- Current model is a hybrid of both
  - Borg will kill lower priority tasks until it finds room for an assigned task

Techniques Borg uses for managing utilization

- Cell-sharing: sharing prod and non-prod tasks
  - Resource reclaiming
  - Not sharing prod and non-prod work would increase machine needs by 20-30%
- Large cells: to allow large computations and decrease fragmentation
  - splitting up jobs and distributing them requires significantly more machines
- Fine-grained resource requests
  - fixed size containers/VMs not ideal
  - instead there are “buckets” of CPU/memory requirements
- Resource reclamation: jobs specify limits
  - Borg can kill tasks that use more RAM or disk space than requested
  - Throttle CPU usage
  - Prioritize prod tasks over non-prod

Borg Architecture - Scalability

- Ultimate scalability limit is unknown
  - Single Borgmaster can manage thousands of borglets
  - Rates above 10,000 tasks per minute
  - Busy Borgmaster uses 10-14 CPU cores and 50GiB RAM

Borg - Achieving Availability

- To mitigate inevitable failures, Borg will:
  - Automatically reschedule evicted tasks
  - Reduce correlated failures by distributing across failure domains
  - Limits downtime due to maintenance
  - Use “declarative desired-state representations and idem-potent mutating operations” to ease resubmission of forgotten requests
  - Avoid task to machine pairings that cause crashes
  - Use a logsaver to recover critical data written to a local disk
- Achieve 99.99% availability in practice

Isolation

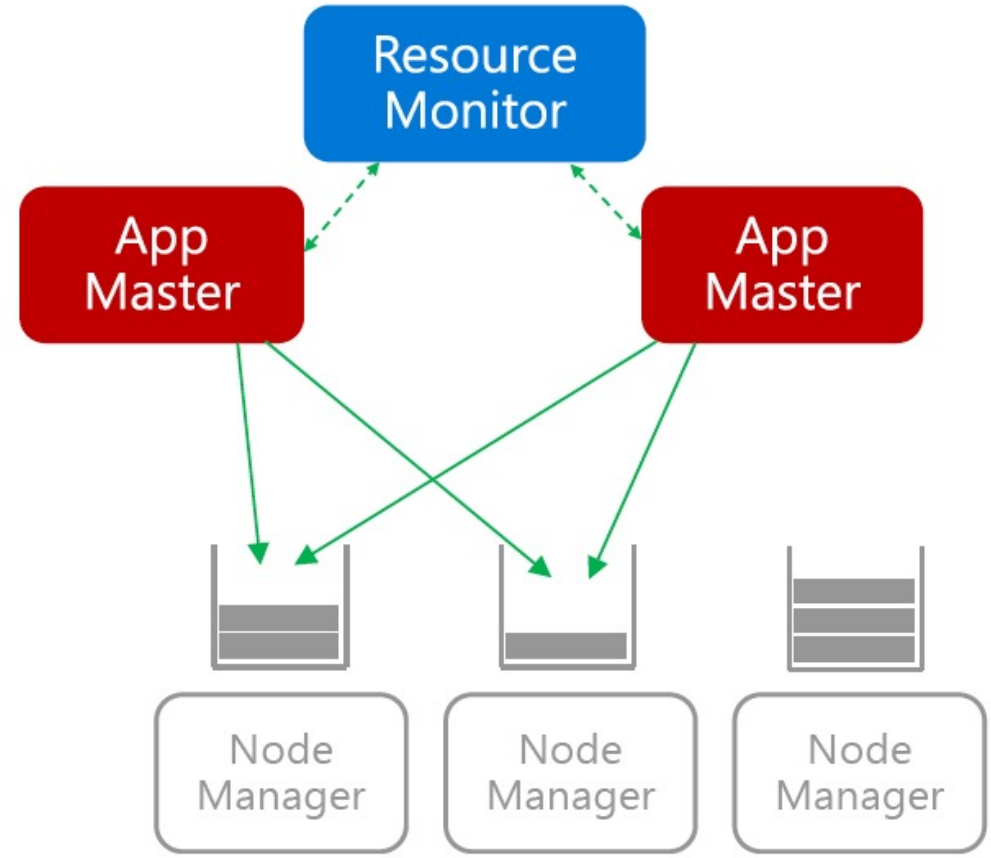
- Security:
  - Linux *chroot* command used for process isolation
  - Standard sandboxing techniques used for running external software
- Performance:
  - Borg makes explicit distinction between LS (latency-intensive) tasks and batch tasks. Helps for priority-based preemption
  - Borg uses notion of compressible resources (CPU cycles, disk I/O bandwidth) and non-compressible resources (RAM, disk space)

Why is it important to have isolation, and how does Borg implement it?

- To protect an app from Noisy, Nosy and Messy neighbors
- Sharing machines between applications increases utilization, but isolation is needed to prevent tasks from interfering
  - Security: rogue tasks can affect other tasks, and information should not be visible between tasks
  - Performance:
    - Utilization can be decreased by users inflating resource requests to prevent interference
    - Again, rogue tasks can affect your task
- Security: Linux chroot jail is the primary security isolation mechanism
- Performance: Linux cgroup-based container
  - Also appclass is used to help with overload and overcommitment
  - High priority LS (latency-sensitive) tasks

Approach 2: Distributed Resource Management

[Apollo, Sparrow]

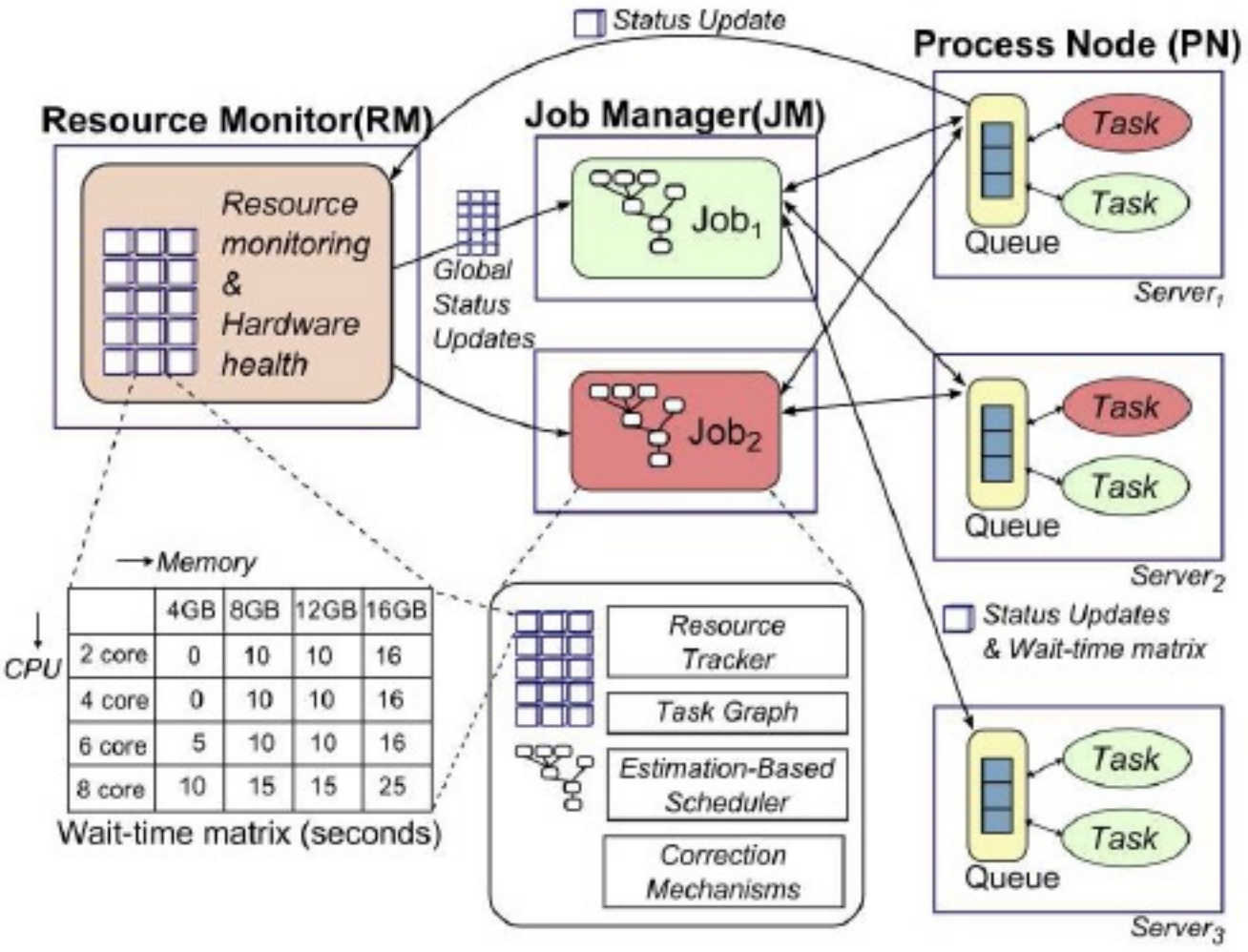


- AMs queue tasks directly to NMs
- Loose coordination through the Resource Monitor

K. Ousterhout et al, “Sparrow: Distributed, Low Latency Scheduling”, ACM SOSP 2013

E. Boutin et al, “Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing”, Usenix OSDI 2014

High-level Distributed Resource Management Architecture of Microsoft's Apollo



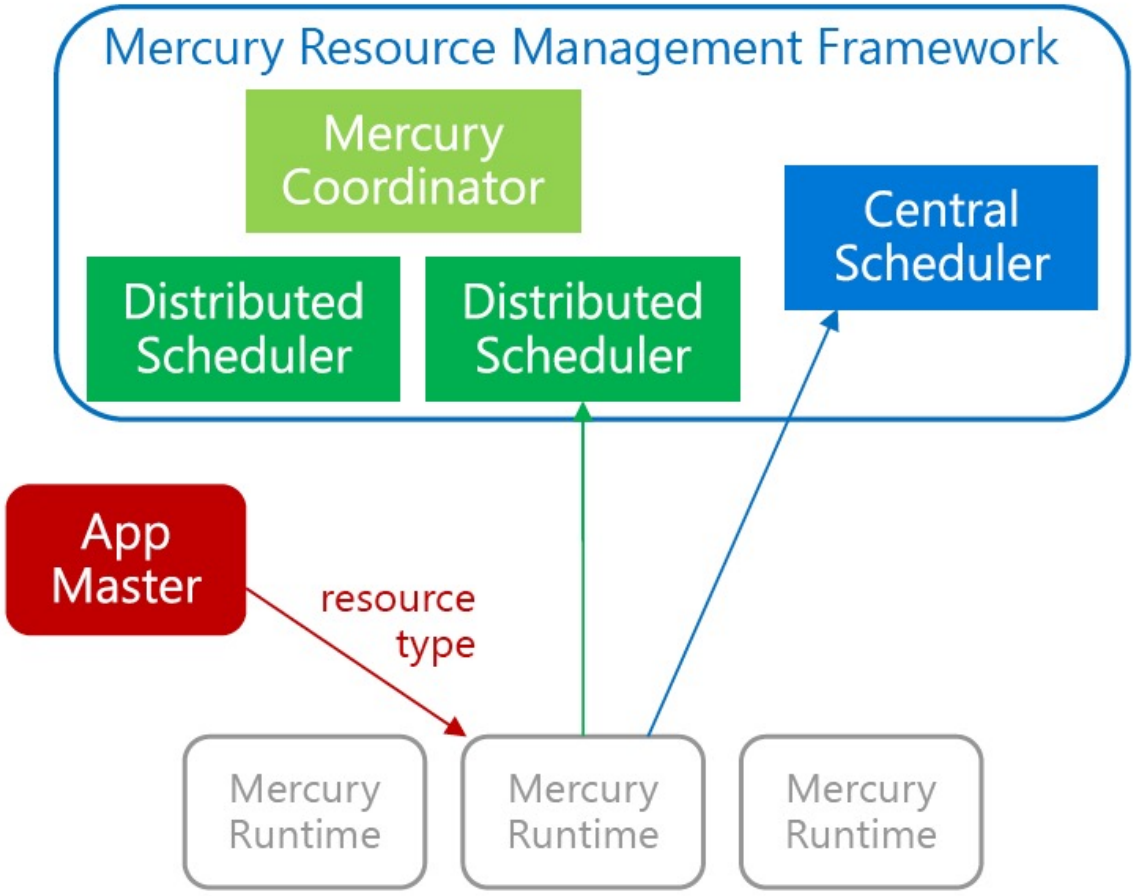
	4GB	8GB	12GB	16GB
2 core	0	10	10	16
4 core	0	10	10	16
6 core	5	10	10	16
8 core	10	15	15	25

E. Boutin et al, "Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing", OSDI 2014

Centralized vs. Distributed Resource Management

	Centralized	Distributed
Workload heterogeneity	✓	
Task placement	✓	
Enforcing scheduling invariants	✓	
Allocation latency		✓
Slot utilization		✓
Scalability		✓

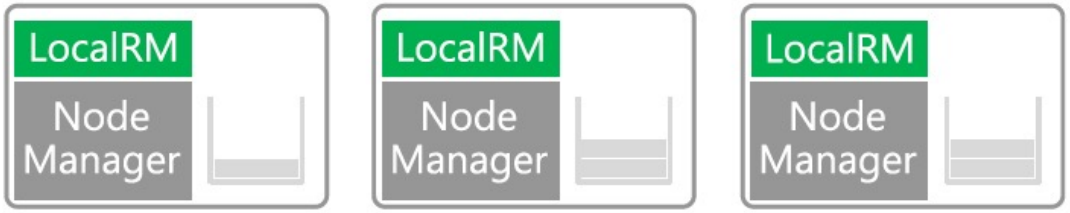
Approach 3: Hybrid (Distributed and Centralized) Resource Management in Microsoft's Mercury



- Two types of schedulers
- Central scheduler
  Scheduling policies/guarantees
  Slow(er) decisions
- Distributed schedulers
  Fast/low-latency decisions
- AM specifies resource *type*

K. Karanasos et al, "Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters", Usenix ATC 2015

Mercury Architecture over YARN



Overview of YARN extensions

- LocalRM (distributed scheduling)
- Queuing of (QUEUEABLE) containers at the NMs
- Framework policies
- Application policies for determining container type per task

Operations and Implementation of Mercury

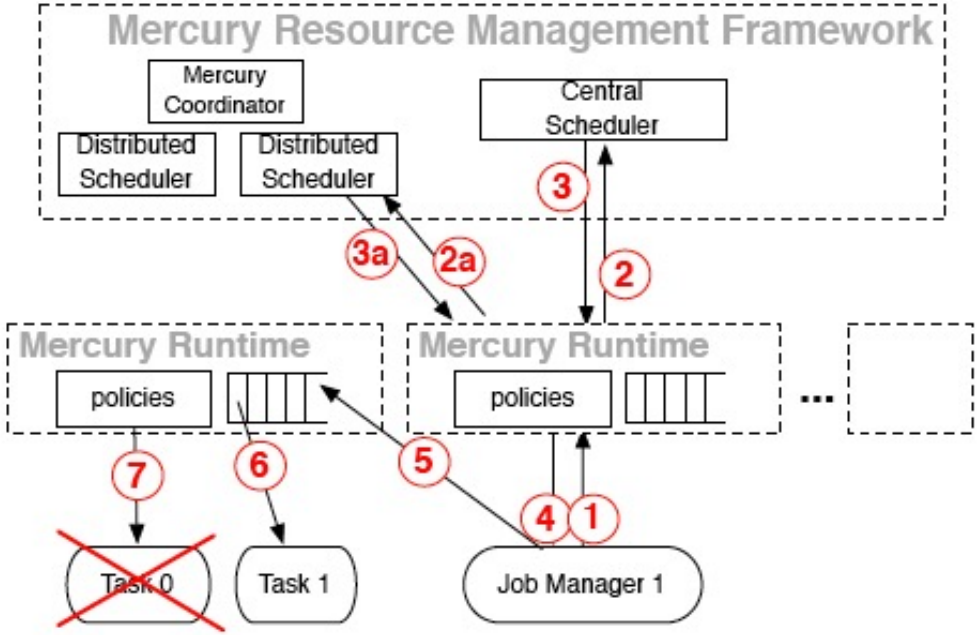


Figure 3: Mercury resource management lifecycle.

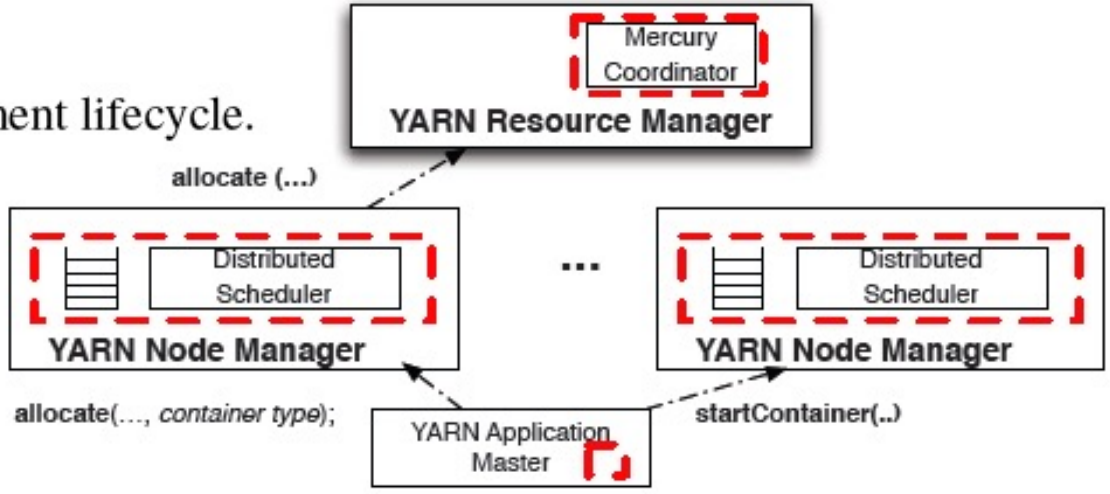


Figure 4: *Mercury implementation*: dashed boxes show Mercury modules and APIs as YARN extensions.

Comparisons of Recent Resource Management Platforms for Clusters

Resource Management Platform for Clusters	Scheduling/Resource Sharing paradigm	Scalability	Multiple Programming Frameworks/ Multi-tenant Support
Hadoop 1.0	Centralized	Limited but OK	No
YARN in Hadoop 2.0	Centralized	Good	Yes
Mesos	Centralized (Two-level) via Resource Offers to Individual Frameworks	Better	Yes
Apollo	Distributed and Loosely Coordinated (via Expected Resource Wait-Time matrix)	Very Good	Yes
Borg, Omega	Centralized per-cell BorgMaster which allows multiple // schedulers to performs optimistic-concurrent allocation followed by checking	Very Good	Yes
Mercury	Hybrid (Centralized and Distributed scheduling for Big and Small jobs respectively)	Very Good	Yes