Cristina Nita-Rotaru

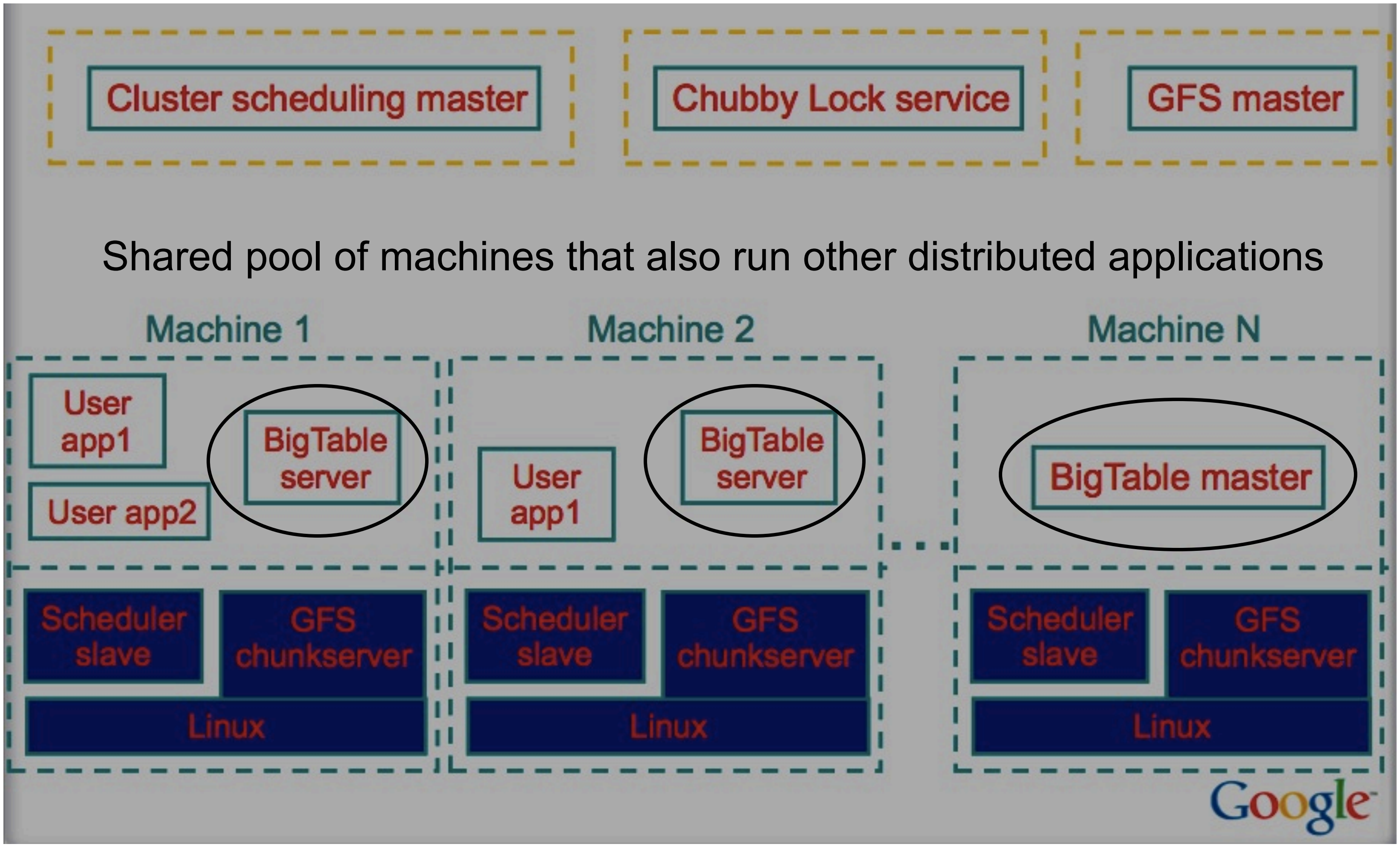
7680: Distributed Systems

MapReduce. Hadoop. Mesos. Yarn

REQUIRED READING

- MapReduce: Simplified Data Processing on Large Clusters OSDI 2004
- Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center, NSDI 2011
- Apache Hadoop YARN: Yet Another Resource Negotiator SOCC 2013 (best paper)
- (Optional) Omega: flexible, scalable schedulers for large compute clusters, EuroSys 2013 (best paper)

Typical Google Cluster



1: MapReduce

These are slides from Dan Weld’s class at U. Washington (who in turn made his slides based on those by Jeff Dean, Sanjay Ghemawat, Google, Inc.)

Motivation

- Large-Scale Data Processing
  - Want to use 1000s of CPUs
    - But don't want hassle of managing things
- MapReduce provides
  - Automatic parallelization & distribution
  - Fault tolerance
  - I/O scheduling
  - Monitoring & status updates

Map/Reduce

- Map/Reduce
  - Programming model from Lisp
  - (and other functional languages)
- Many problems can be phrased this way
- Easy to distribute across nodes
- Nice retry/failure semantics

Map in Lisp (Scheme)

- (map f list [list2 list3 …])

Unary operator

- (map square ‘(1 2 3 4))
  - (1 4 9 16)

Binary operator

- (reduce + ‘(1 4 9 16))
  - (+ 16 (+ 9 (+ 4 1)))
- (reduce + (map square (map – 11 12))))

Map/Reduce ala Google

- map(key, val) is run on each item in set
  - emits new-key / new-val pairs

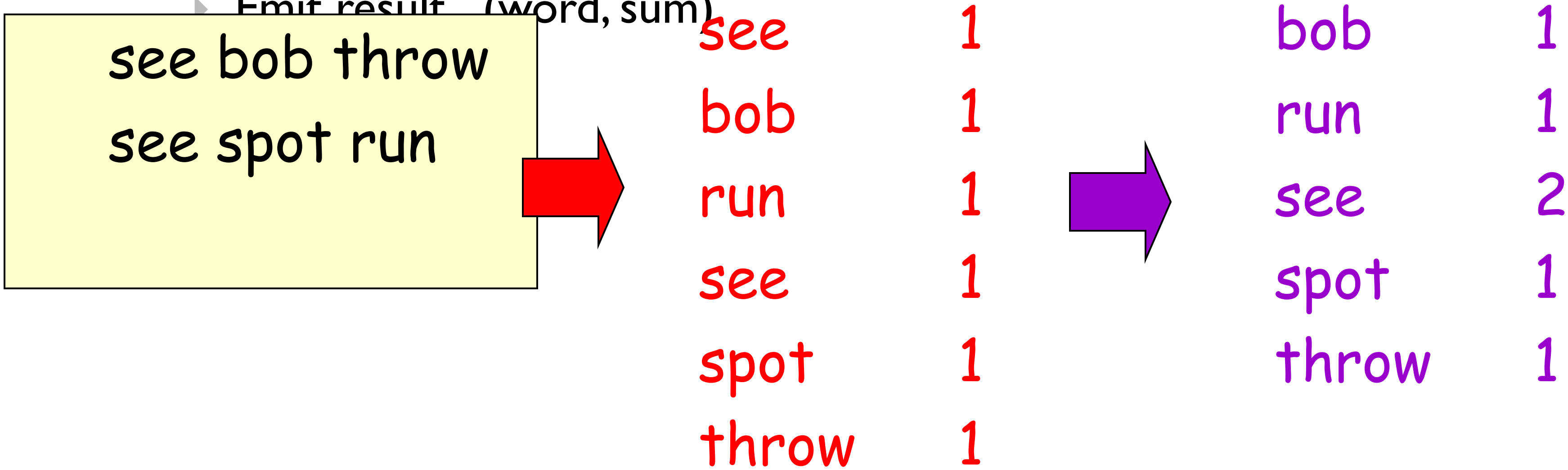
- reduce(key, vals) is run for each unique key emitted by map()
  - emits final output

count words in docs

- Input consists of (url, contents) pairs
- map(key=url, val=contents):
  - For each word w in contents, emit (w, "1")
- reduce(key=word, values=uniq_counts):
  - Sum all "1"s in values list
  - Emit result "(word, sum)"

Count, Illustrated

- map(key=url, val=contents):
  - For each word w in contents, emit (w, "1")
- reduce(key=word, values=uniq_counts):
  - Sum all "1"s in values list
  - Emit result "(word, sum)"



Grep

- Input consists of (url+offset, single line)
- map(key=url+offset, val=line):
  - If contents matches regexp, emit (line, "1")
- reduce(key=line, values=uniq_counts):
  - Don't do anything; just emit line

Reverse Web-Link Graph

- Map
  - For each URL linking to target, …
  - Output <target, source> pairs
- Reduce
  - Concatenate list of all source URLs
  - Outputs: <target, list (source)> pairs

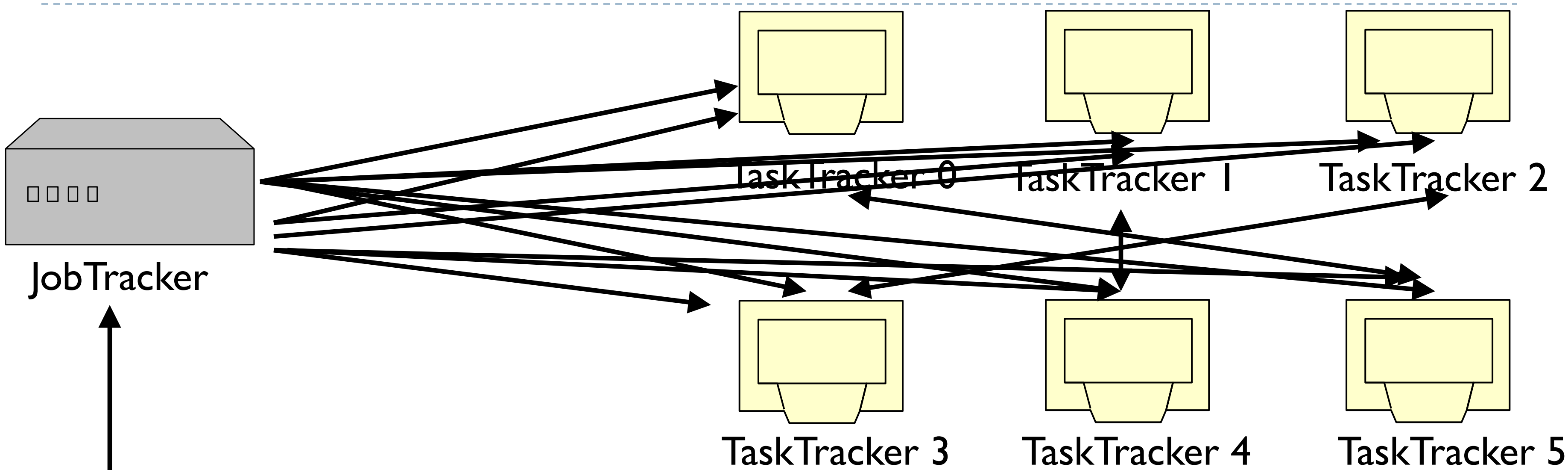
Implementation

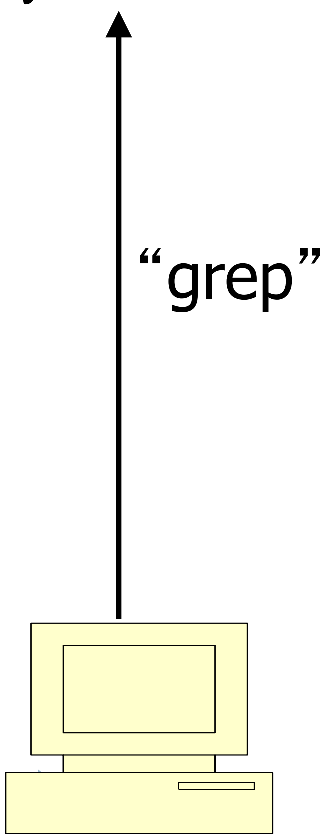
- Typical cluster:
  - 100s/1000s of 2-CPU x86 machines, 2-4 GB of memory
  - Limited bisection bandwidth
  - Storage is on local IDE disks
  - GFS: distributed file system manages data
- Job scheduling system: jobs made up of tasks, scheduler assigns tasks to machines
- Implementation is a C++ library linked into user programs

Execution

- How is this distributed?
  - Partition input key/value pairs into chunks, run map() tasks in parallel
  - After all map()s are complete, consolidate all emitted values for each unique emitted key
  - Now partition space of output map keys, and run reduce() in parallel
- If map() or reduce() fails, reexecute!

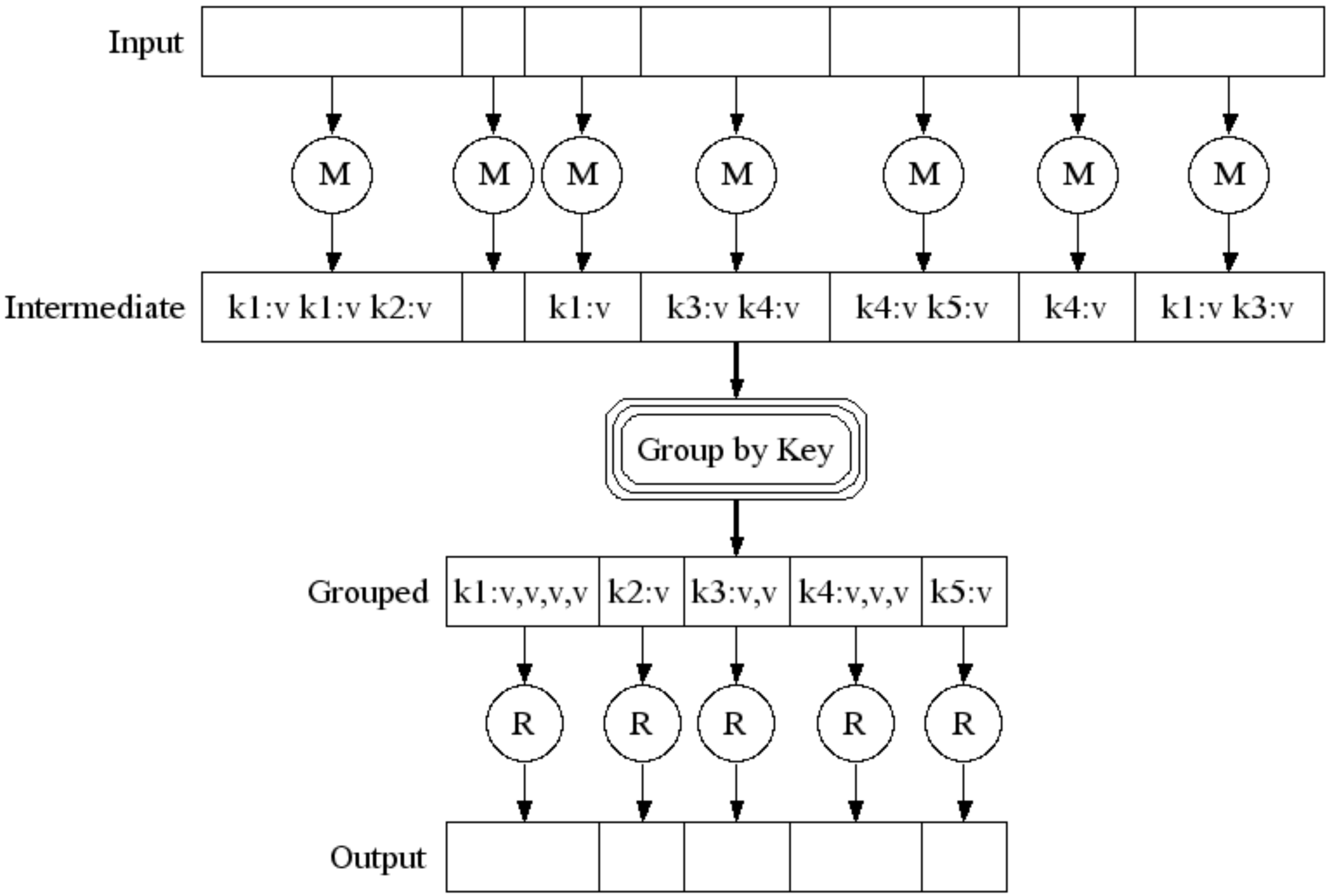
Job Processing



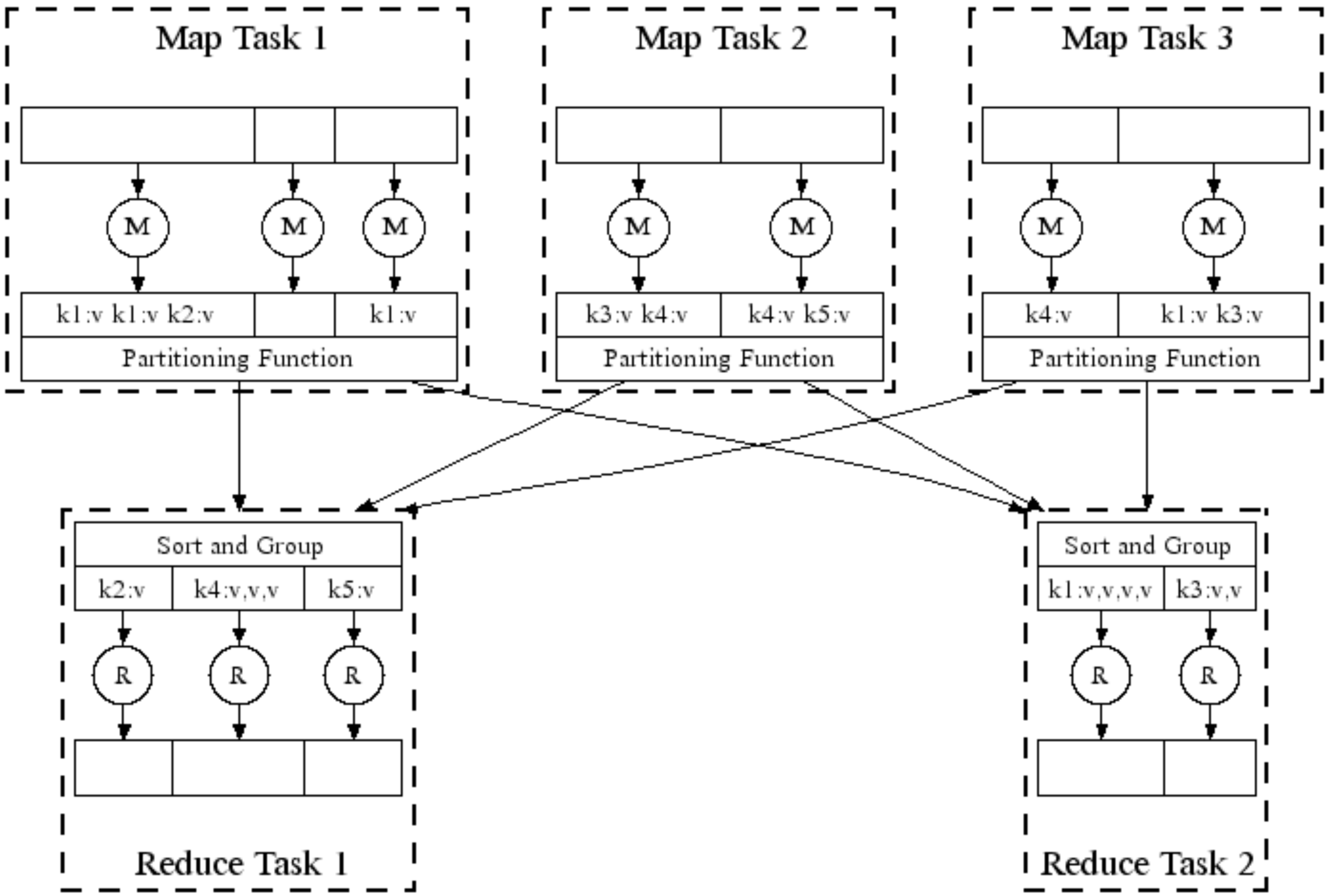


1. Client submits "grep" job, indicating code and input files
2. JobTracker breaks input file into *k* chunks, (in this case 6). Assigns work to ttrackers.
3. After map(), tasktrackers exchange map-output to build reduce() keyspace
4. JobTracker breaks reduce() keyspace into *m* chunks (in this case 6). Assigns work.
5. reduce() output may go to GFS

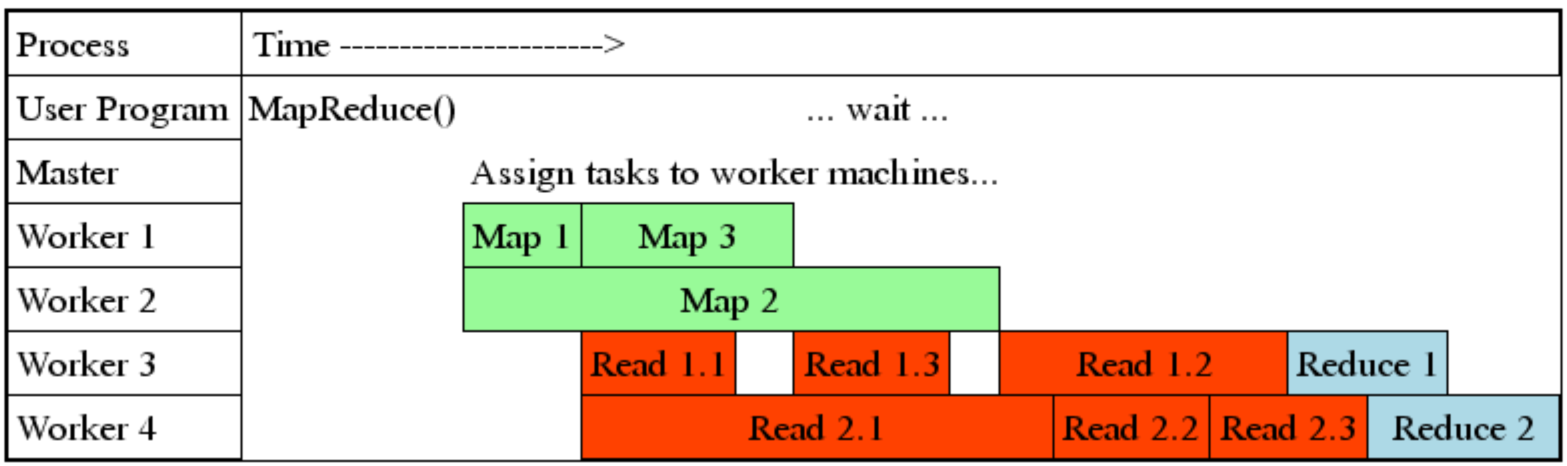
Execution


Input
M
M
M
M
M
M
M
Intermediate
k1:v k1:v k2:v
k1:v
k3:v k4:v
k4:v k5:v
k4:v
k1:v k3:v
Group by Key
Grouped
k1:v,v,v,v
k2:v
k3:v,v
k4:v,v,v
k5:v
R
R
R
R
R
Output

Parallel Execution



Task Granularity & Pipelining

- Fine granularity tasks: map tasks >> machines
  - Minimizes time for fault recovery
  - Can pipeline shuffling with map execution
  - Better dynamic load balancing
- Often use 200,000 map & 5000 reduce tasks
- 

Fault Tolerance / Workers

- Handled via re-execution
  - Detect failure via periodic heartbeats
  - Re-execute completed + in-progress map tasks
  - Re-execute in progress reduce tasks
  - Task completion committed through master
- Robust: lost 1600/1800 machines once → finished ok

Master Failure

- Could handle, … ?
- But don't yet
  - (master failure unlikely)

Refinement: Redundant Execution

Slow workers significantly delay completion time

- Other jobs consuming resources on machine
- Bad disks w/ soft errors transfer data slowly
- Weird things: processor caches disabled (!!)

Solution: Near end of phase, spawn backup tasks

- Whichever one finishes first "wins"

Dramatically shortens job completion time

Refinement: Locality Optimization

- Master scheduling policy:
  - Asks GFS for locations of replicas of input file blocks
  - Map tasks typically split into 64MB (GFS block size)
  - Map tasks scheduled so GFS input block replica are on same machine or same rack
- Effect
  - Thousands of machines read input at local disk speed
    - Without this, rack switches limit read rate

Refinement: Skipping Bad Records

- Map/Reduce functions sometimes fail for particular inputs
  - Best solution is to debug & fix
    - Not always possible ~ third-party source libraries
  - On segmentation fault:
    - Send UDP packet to master from signal handler
    - Include sequence number of record being processed
  - If master sees two failures for same record:
    - Next worker is told to skip the record

Other Refinements

- Sorting guarantees
  - within each reduce partition
- Compression of intermediate data
- Combiner
  - Useful for saving network bandwidth
- Local execution for debugging/testing
- User-defined counters

Performance

Tests run on cluster of 1800 machines:

- 4 GB of memory
- Dual-processor 2 GHz Xeons with Hyperthreading
- Dual 160 GB IDE disks
- Gigabit Ethernet per machine
- Bisection bandwidth approximately 100 Gbps

Two benchmarks:

MR_GrepScan 1010 100-byte records to extract records matching a rare pattern (92K matching records)

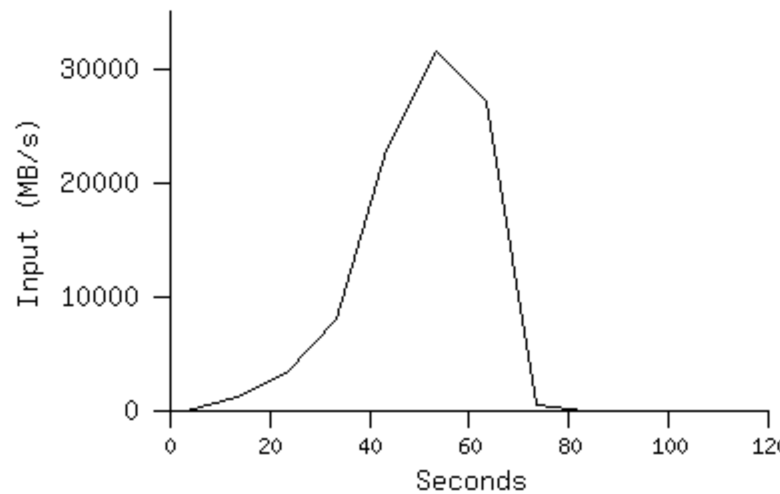
MR_SortSort 1010 100-byte records (modeled after TeraSort benchmark)

MR_Grep

Locality optimization helps:

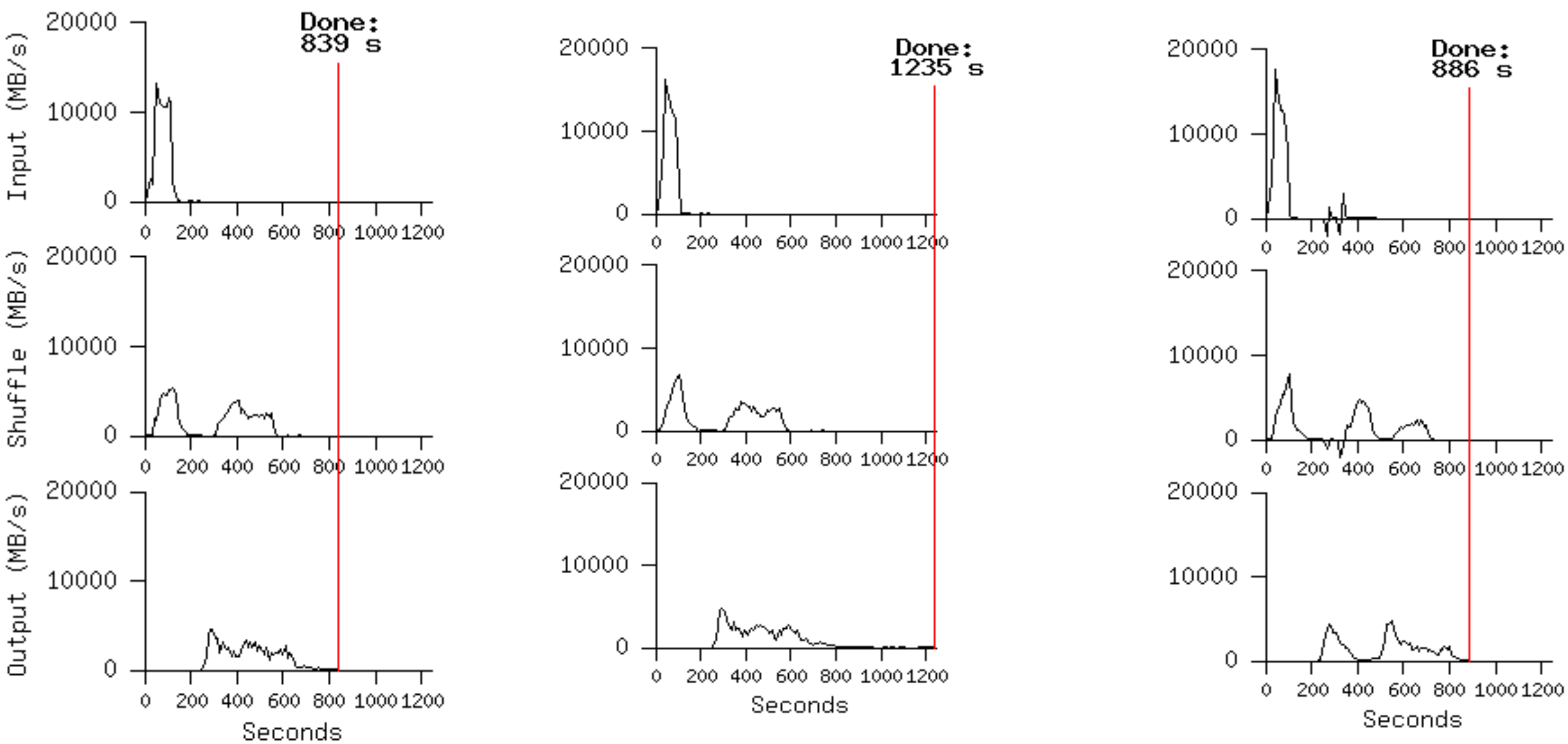
- 1800 machines read 1 TB at peak ~31 GB/s
- W/out this, rack switches would limit to 10 GB/s

Startup overhead is significant for short jobs



MR_Sort

Normal **No backup tasks** **200 processes killed**



- Backup tasks reduce job completion time a lot!
- System deals well with failures

2: Hadoop

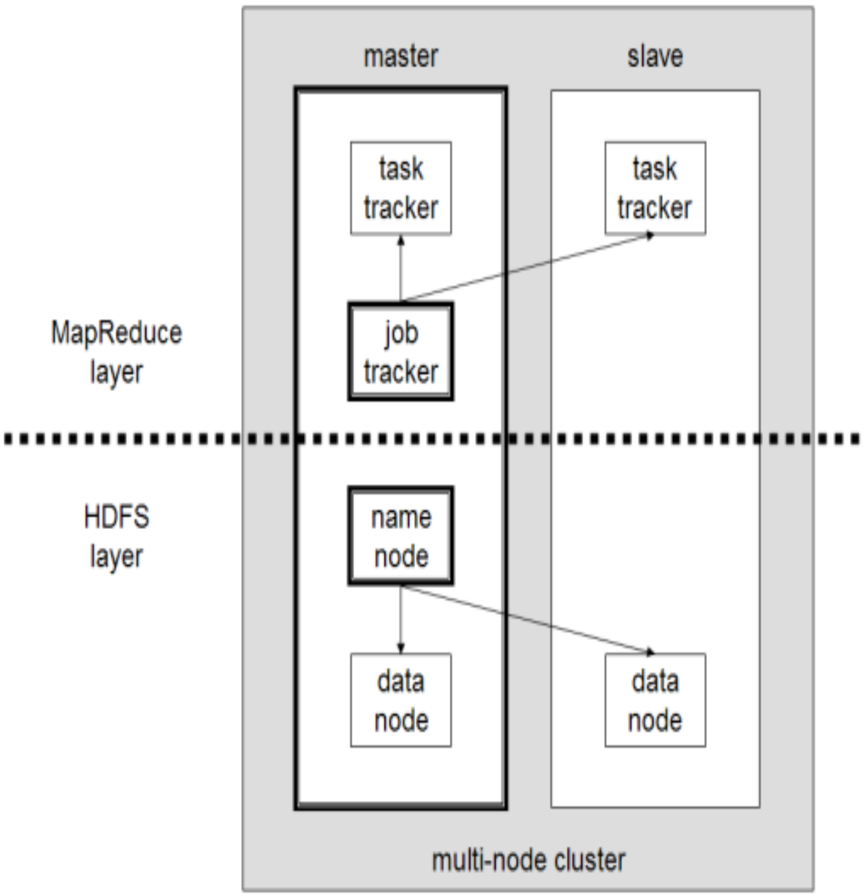
Apache Hadoop

- Apache Hadoop's MapReduce and HDFS components originally derived from
  - Google File System (GFS)[1] – 2003
  - Google's MapReduce[2] - 2004
- Data is broken in splits that are processed in different machines.
- Industry wide standard for processing Big Data.

Overview of Hadoop

- Basic components of Hadoop are:
  - **Map Reduce Layer**
    - **Job tracker** (master) -which coordinates the execution of jobs;
    - **Task trackers** (slaves)- which control the execution of map and reduce tasks in the machines that do the processing;
  - **HDFS Layer**- which stores files.
    - **Name Node** (master)- manages the file system, keeps metadata for all the files and directories in the tree
    - **Data Nodes** (slaves)- work horses of the file system. Store and retrieve blocks when they are told to (by clients or name node) and report back to name node periodically

Overview of Hadoop contd.



Job Tracker - coordinates the execution of jobs

Task Tracker – control the execution of map and reduce tasks in slave machines

Name Node – Manages the file system, keeps metadata

Data Node – Follow the instructions from name node
- stores, retrieves data

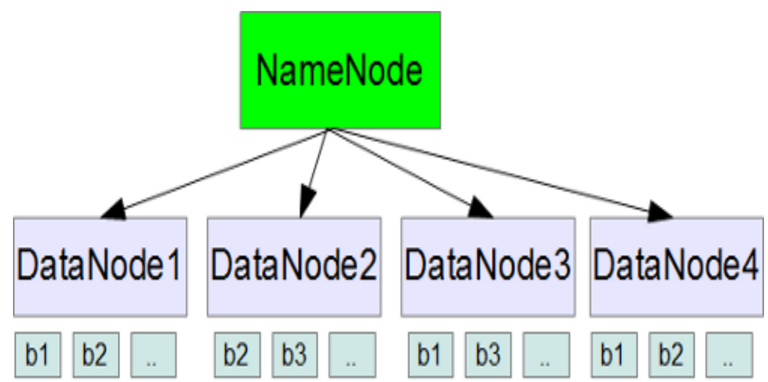
Hadoop Versions

Feature	1.x	0.22	2.x
Secure authentication	Yes	No	Yes
Old configuration names	Yes	Deprecated	Deprecated
New configuration names	No	Yes	Yes
Old MapReduce API	Yes	Yes	Yes
New MapReduce API	Yes (with somemissing libraries)	Yes	Yes
MapReduce 1 runtime (Classic)	Yes	Yes	No
MapReduce 2 runtime (YARN)	No	No	Yes
HDFS federation	No	No	Yes
HDFS high-availability	No	No	Yes

- MapReduce 2 runtime and HDFS HA was introduced in Hadoop 2.x

Fault Tolerance in HDFS layer

- Hardware failure is the norm rather than the exception
- **Detection of faults and quick, automatic recovery from them** is a core architectural goal of HDFS.
- Master Slave Architecture with NameNode (master) and DataNode (slave)
- Common types of failures
  - NameNode failures
  - DataNode failures



Handling Data Node Failure

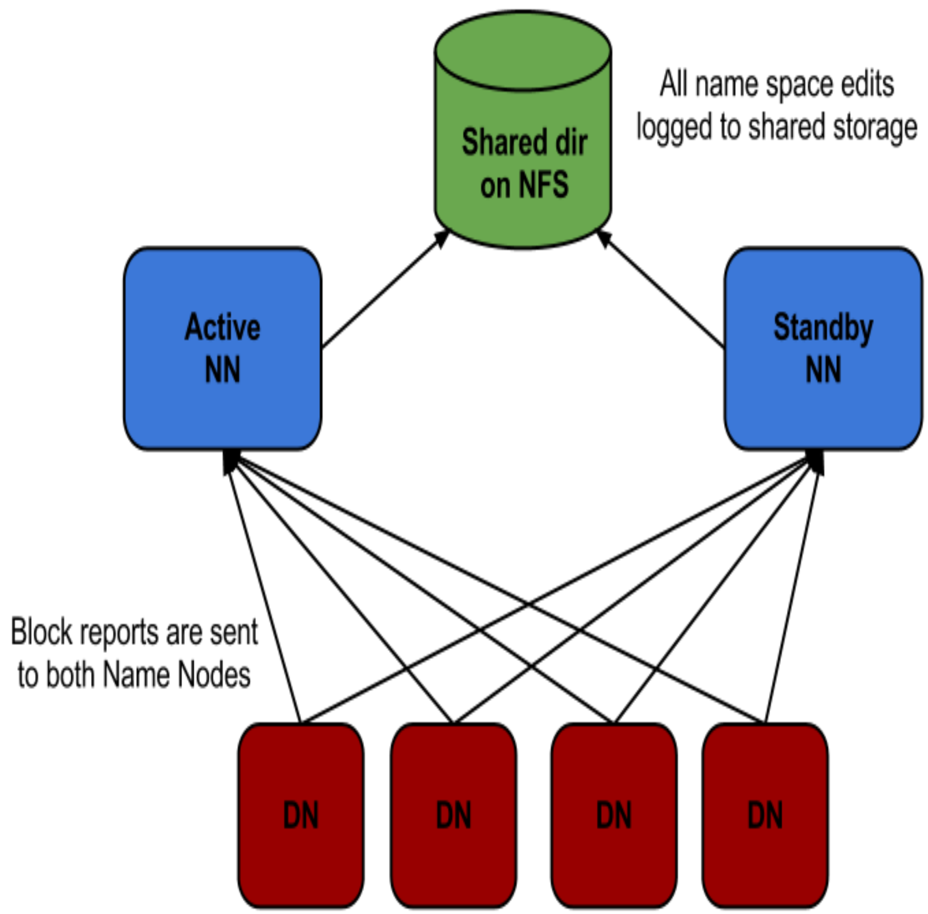
- Each DataNode sends a Heartbeat message to the NameNode periodically
- If the namenode does not receive a heartbeat from a particular data node for 10 minutes, then it considers that data node to be dead/out of service.
- Name Node initiates replication of blocks which were hosted on that data node to be hosted on some other data node.

Handling Name Node Failure

- Single Name Node per cluster.
- Prior to Hadoop 2.0.0, the NameNode was a single point of failure (SPOF) in an HDFS cluster.
- If NameNode becomes unavailable, the cluster as a whole would be unavailable
  - NameNode has to be restarted
  - Brought up on a separate machine.

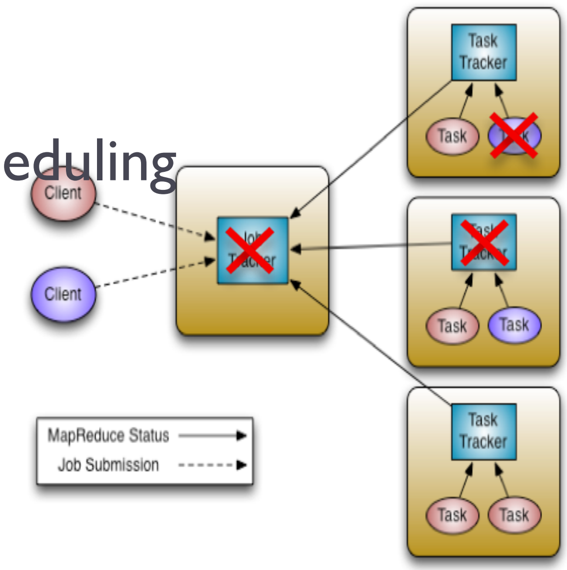
HDFS High Availability

- Provides an option of running two redundant NameNodes in the same cluster
- Active/Passive configuration with a hot standby.
- Fast failover to a new NameNode in the case that a machine crashes
- Graceful administrator-initiated failover for the purpose of planned maintenance.



Classic MapReduce (v1)

- Job Tracker
  - Manage Cluster Resources and Job Scheduling
- Task Tracker
  - Per-node agent
  - Manage Tasks
- Jobs can fail
  - While running the task (Task Failure)
  - Task Tracker failure
  - Job Tracker failure



Handling Task Failure

- User code bug in map/reduce
  - Throws a RunTimeException
  - Child JVM reports a failure back to the parent task tracker before it exits.
- Sudden exit of the child JVM
  - Bug that causes the JVM to exit for the conditions exposed by map/reduce code.
- Task tracker marks the task attempt as failed, makes room available to another task.

Task Tracker Failure

- Task tracker will stop sending the heartbeat to the Job Tracker
- Job Tracker notices this failure
  - Hasn't received a heart beat from 10 mins
  - Can be configured via mapred.tasktracker.expiry.interval property
- Job Tracker removes this task from the task pool
- Rerun the Job even if map task has ran completely
  - Intermediate output resides in the failed task trackers local file system which is not accessible by the reduce tasks.

Job Tracker Failure

- This is serious than the other two modes of failure.
  - Single point of failure.
  - In this case all jobs will fail.
- After restarting Job Tracker all the jobs running at the time of the failure needs to be resubmitted.

3: Mesos

Slides by Matei Zaharia

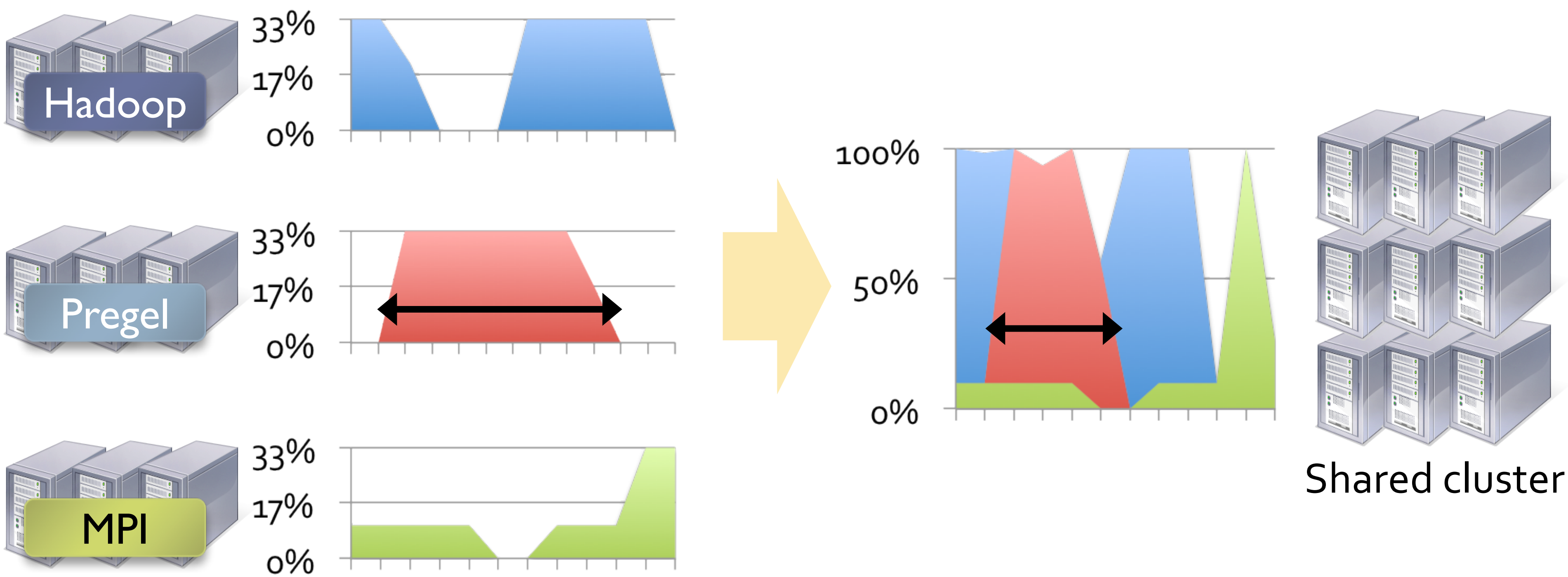
Problem

- Rapid innovation in cluster computing frameworks
- No single framework optimal for all applications
- Want to run multiple frameworks in a single cluster
  - …to maximize utilization
  - …to share data between frameworks

Where We Want to Go

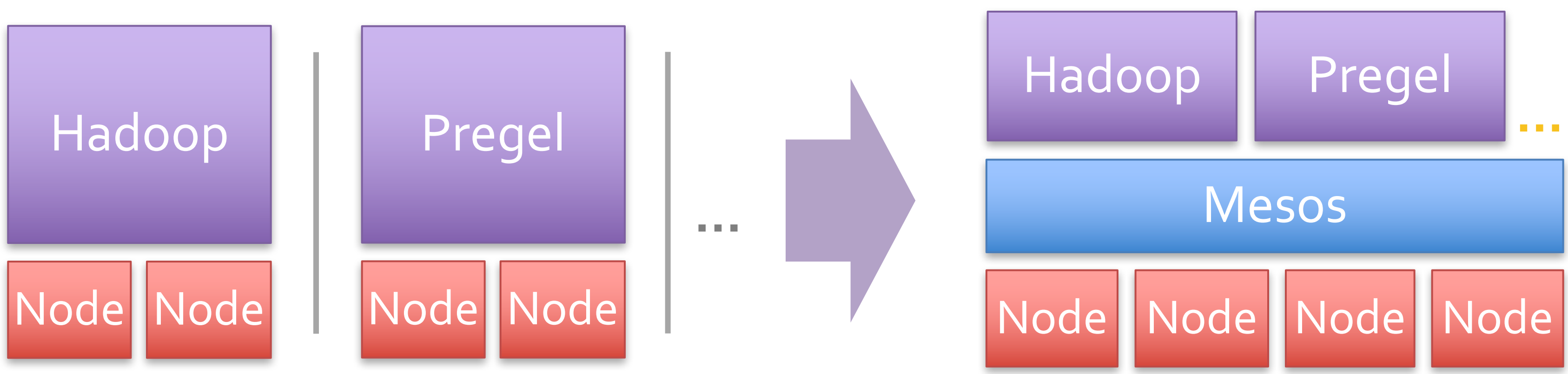
Today: static partitioning

Mesos: dynamic sharing



Solution

- Mesos is a common resource sharing layer over which diverse frameworks can run



Other Benefits of Mesos

- Run multiple instances of the same framework
  - Isolate production and experimental jobs
  - Run multiple versions of the framework concurrently
- Build specialized frameworks targeting particular problem domains
  - Better performance than general-purpose abstractions

Mesos Goals

- High utilization of resources
- Support diverse frameworks (current & future)
- Scalability to 10,000's of nodes
- Reliability in face of failures

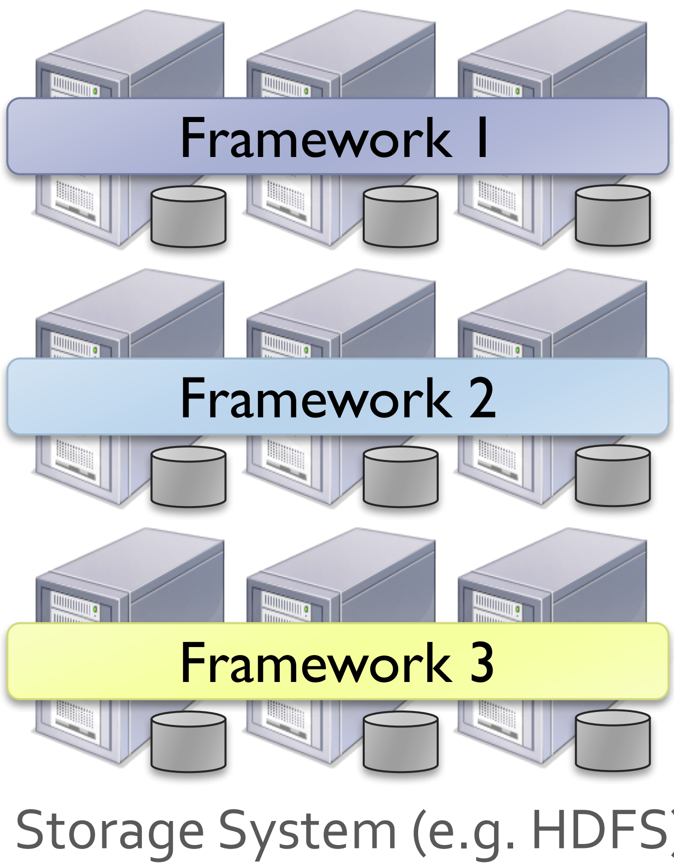
Resulting design: Small microkernel-like core that pushes scheduling logic to frameworks

Design Elements

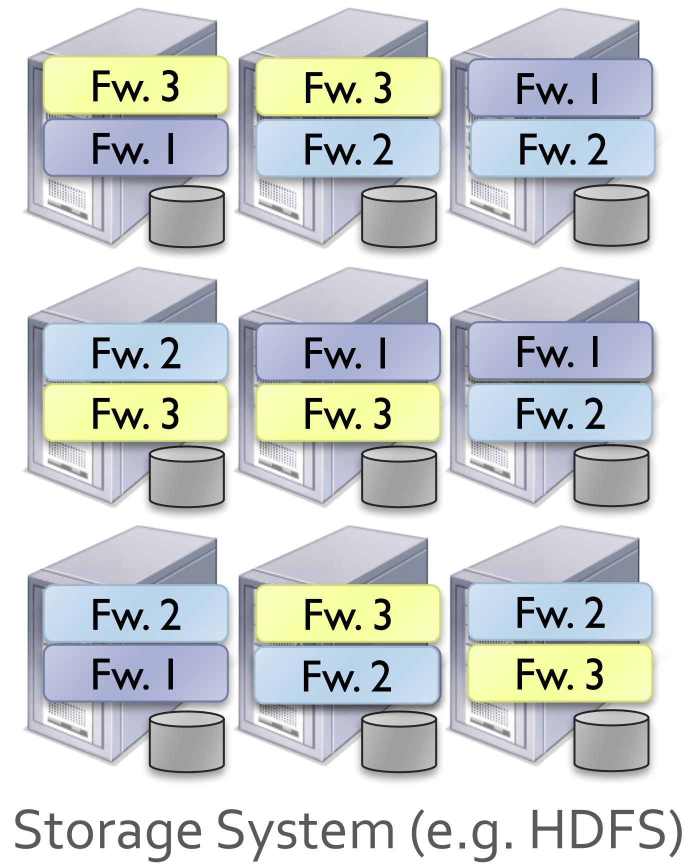
- Fine-grained sharing:
  - Allocation at the level of tasks within a job
  - Improves utilization, latency, and data locality
- Resource offers:
  - Simple, scalable application-controlled scheduling mechanism

Element 1: Fine-Grained Sharing

Coarse-Grained Sharing (HPC):



Fine-Grained Sharing (Mesos):



+ Improved utilization, responsiveness, data locality

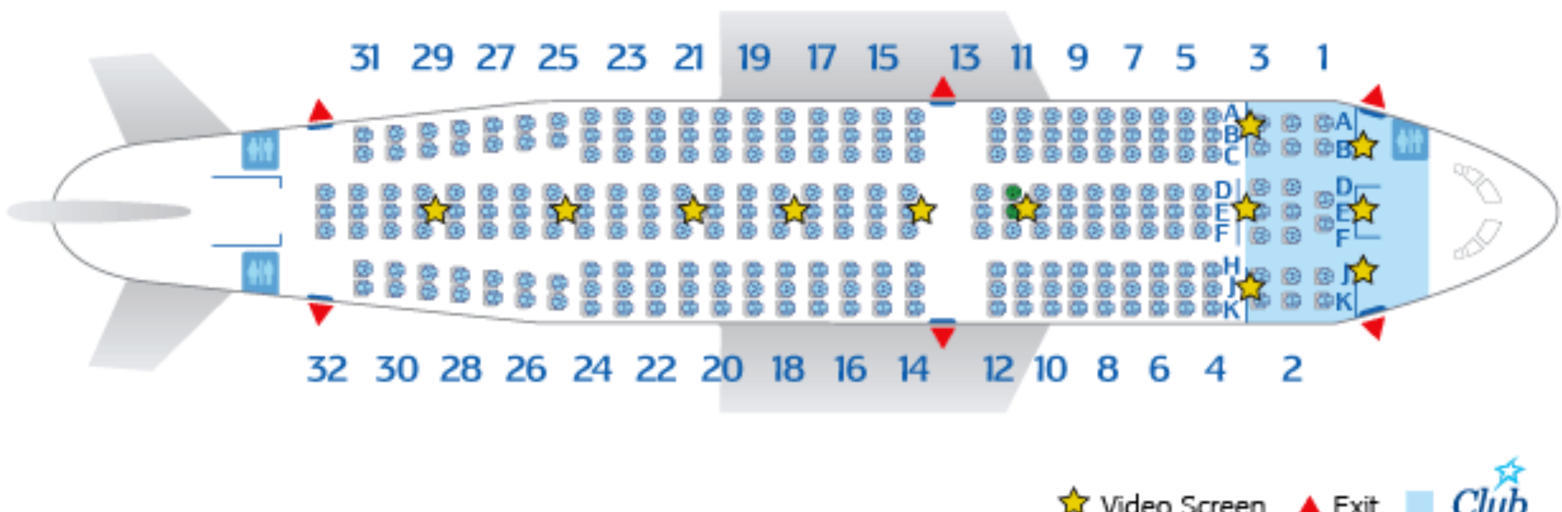
Element 2: Resource Offers

- 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

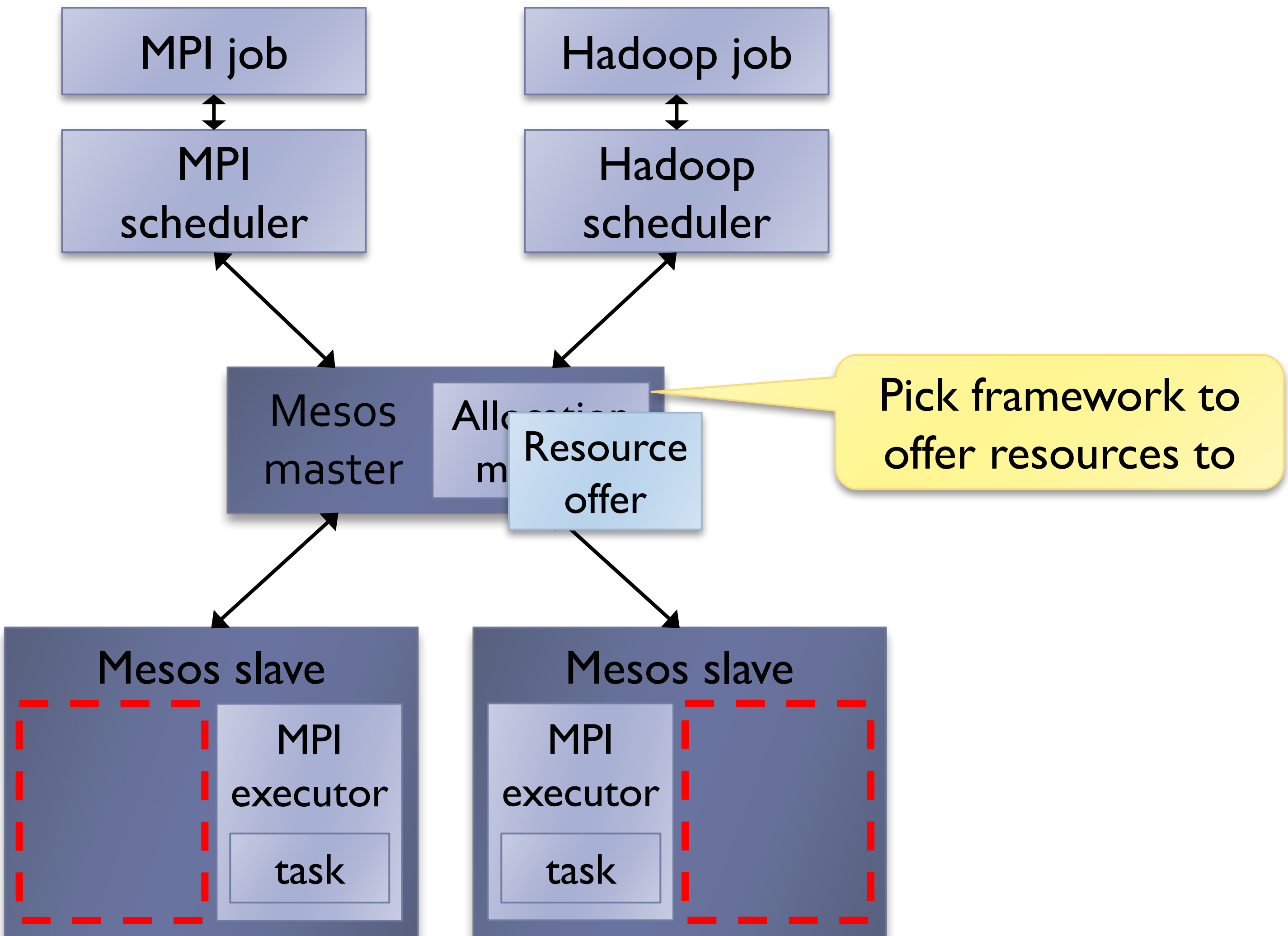
Element 2: Resource Offers

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

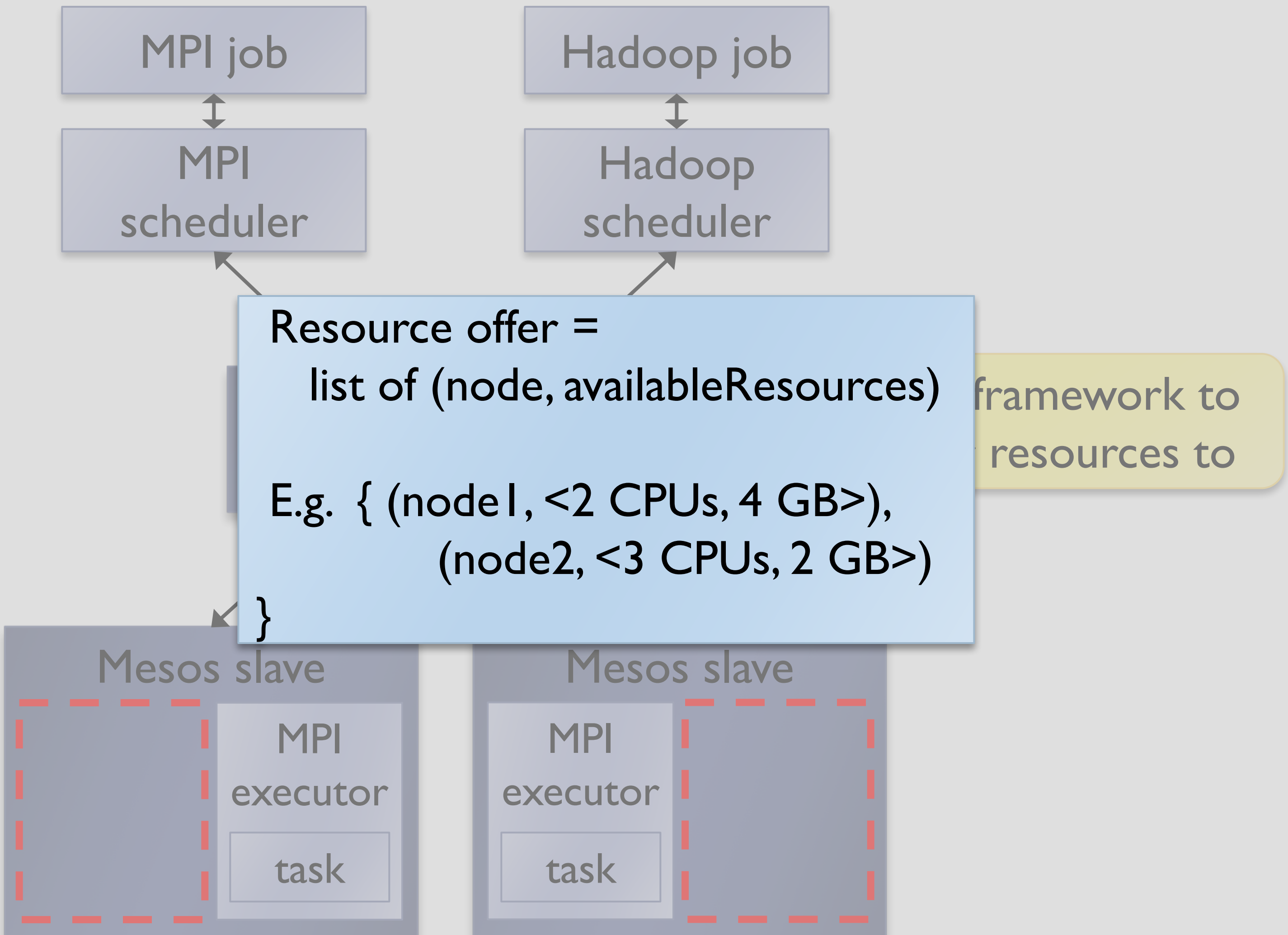
  - Keeps Mesos simple, lets it support future frameworks
  - Decentralized decisions might not be optimal



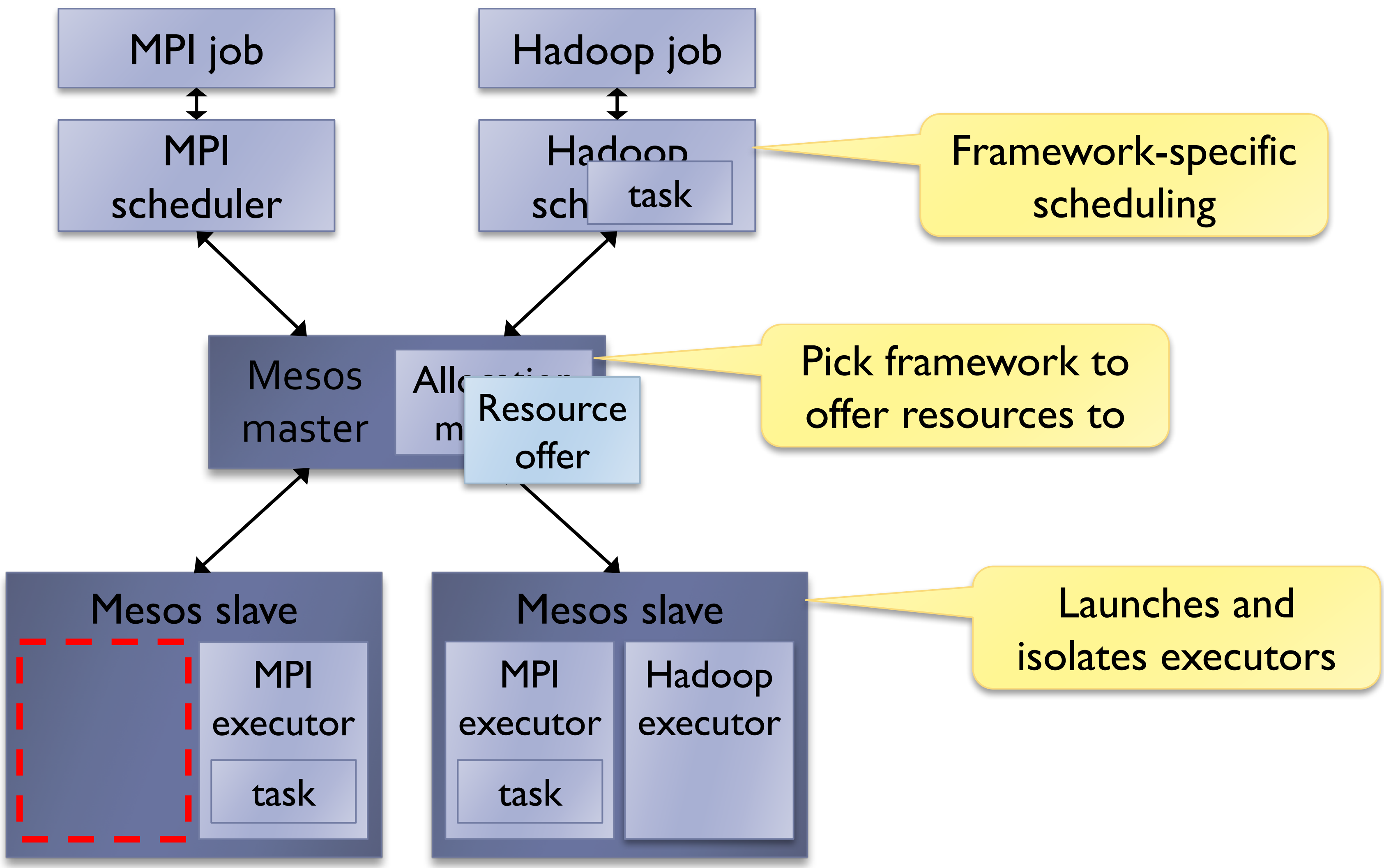
Mesos Architecture



Mesos Architecture



Mesos Architecture



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

Implementation Stats

- 20,000 lines of C++
- Master failover using ZooKeeper
- Frameworks ported: Hadoop, MPI, Torque
- New specialized framework: Spark, for iterative jobs (up to 20 × faster than Hadoop)
- Open source in Apache Incubator

Users

- Twitter uses Mesos on > 100 nodes to run ~12 production services (mostly stream processing)
- Berkeley machine learning researchers are running several algorithms at scale on Spark
- Conviva is using Spark for data analytics
- UCSF medical researchers are using Mesos to run Hadoop and eventually non-Hadoop apps

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

Analysis

- Resource offers work well when:
  - Frameworks can scale up and down elastically
  - Task durations are homogeneous
  - Frameworks have many preferred nodes
- These conditions hold in current data analytics frameworks (MapReduce, Dryad, …)
  - Work divided into short tasks to facilitate load balancing and fault recovery
  - Data replicated across multiple nodes

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

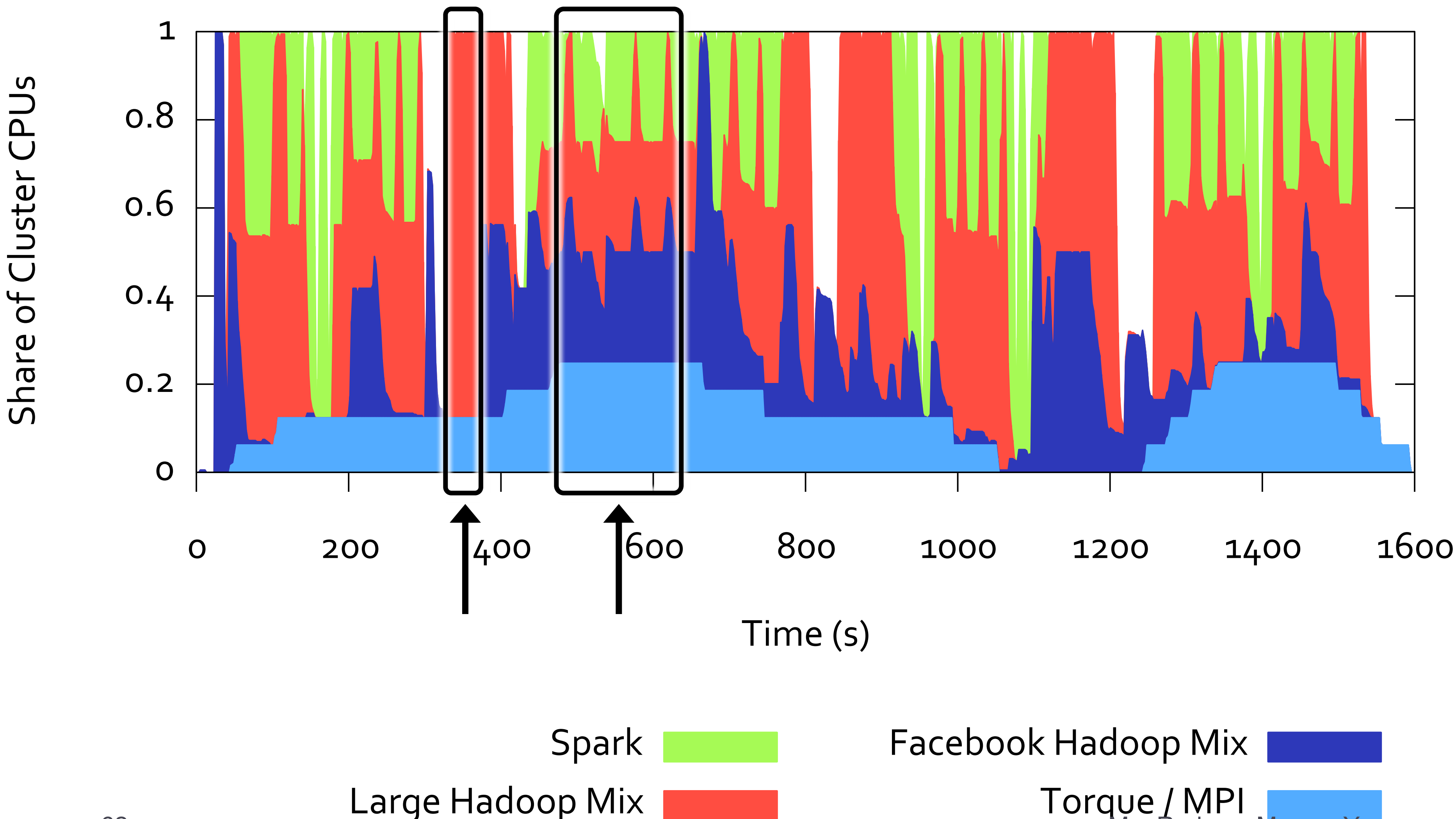
Scheduler Callbacks	Scheduler Actions
resourceOffer(offerId, offers) offerRescinded(offerId) statusUpdate(taskId, status) slaveLost(slaveId)	replyToOffer(offerId, tasks) setNeedsOffers(bool) setFilters(filters) getGuaranteedShare() killTask(taskId)

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

Results

- Utilization and performance vs static partitioning
- Framework placement goals: data locality
- Scalability
- Fault recovery

Dynamic Resource Sharing



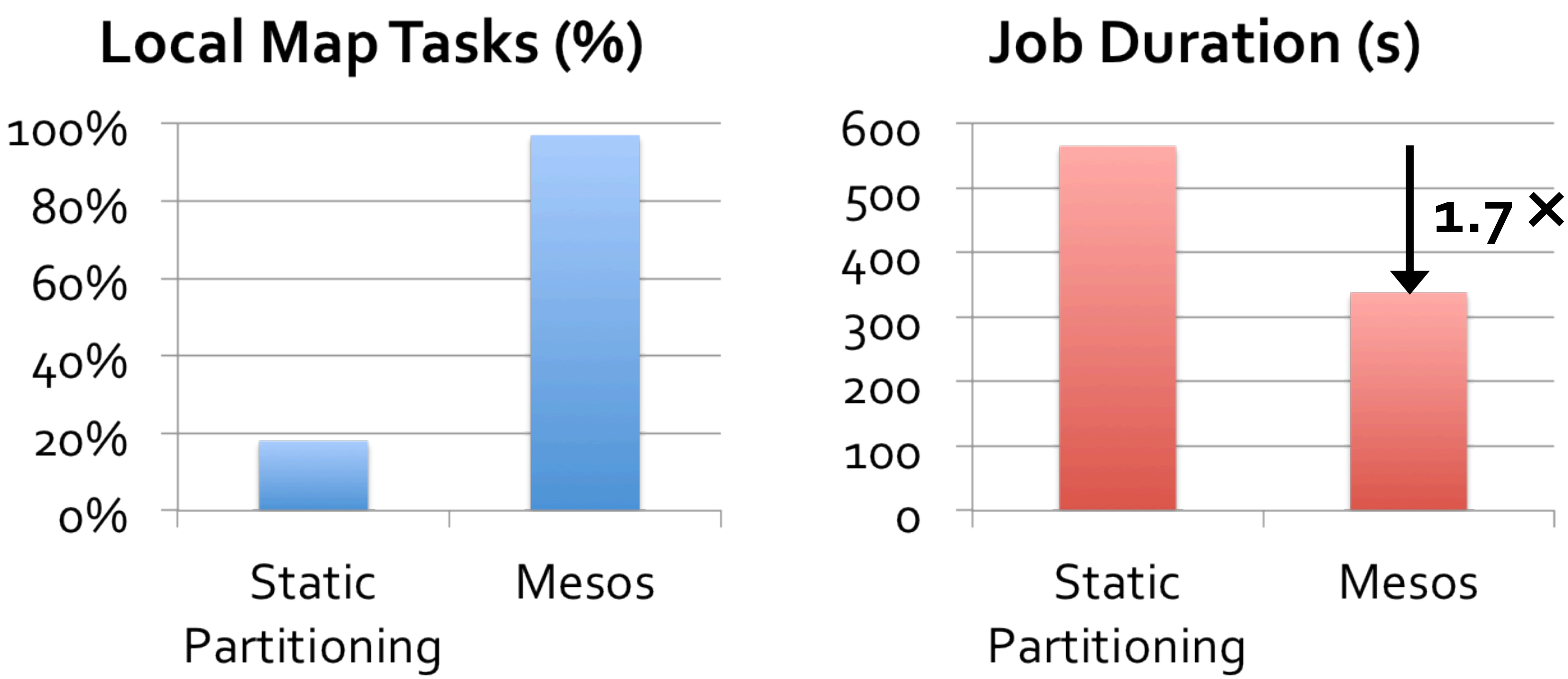
Mesos vs Static Partitioning

- Compared performance with statically partitioned cluster where each framework gets 25% of nodes

Framework	Speedup on Mesos
Facebook Hadoop Mix	1.14×
Large Hadoop Mix	2.10×
Spark	1.26×
Torque / MPI	0.96×

Data Locality with Resource Offers

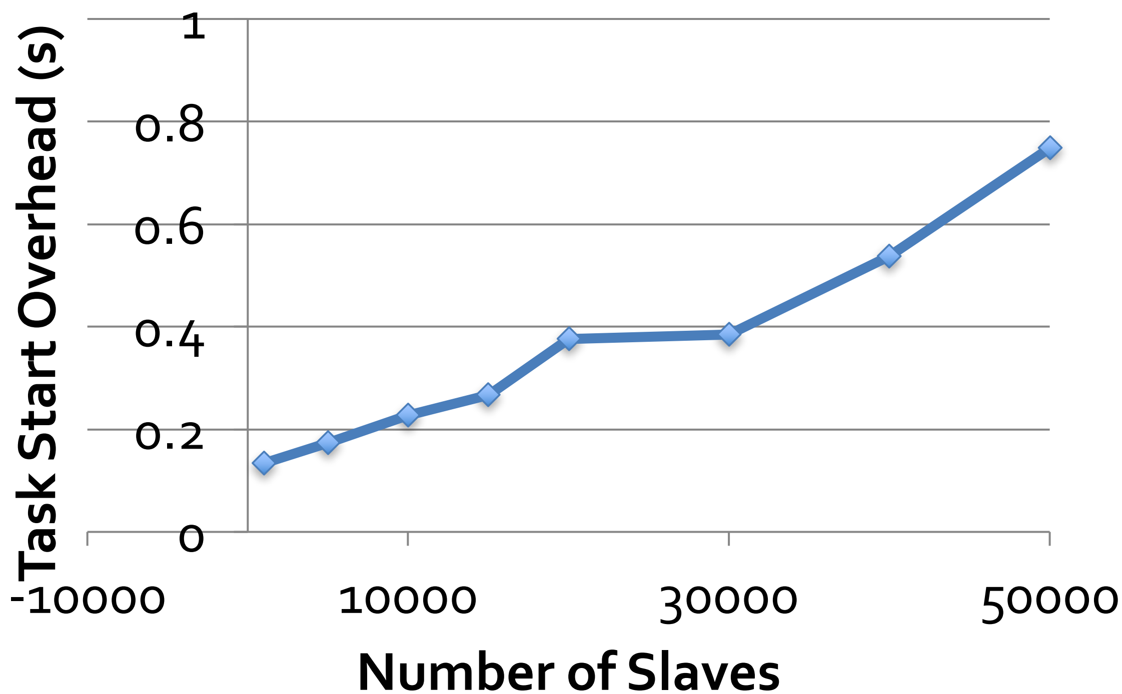
- Ran 16 instances of Hadoop on a shared HDFS cluster
- Used delay scheduling [EuroSys '10] in Hadoop to get locality (wait a short time to acquire data-local nodes)



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

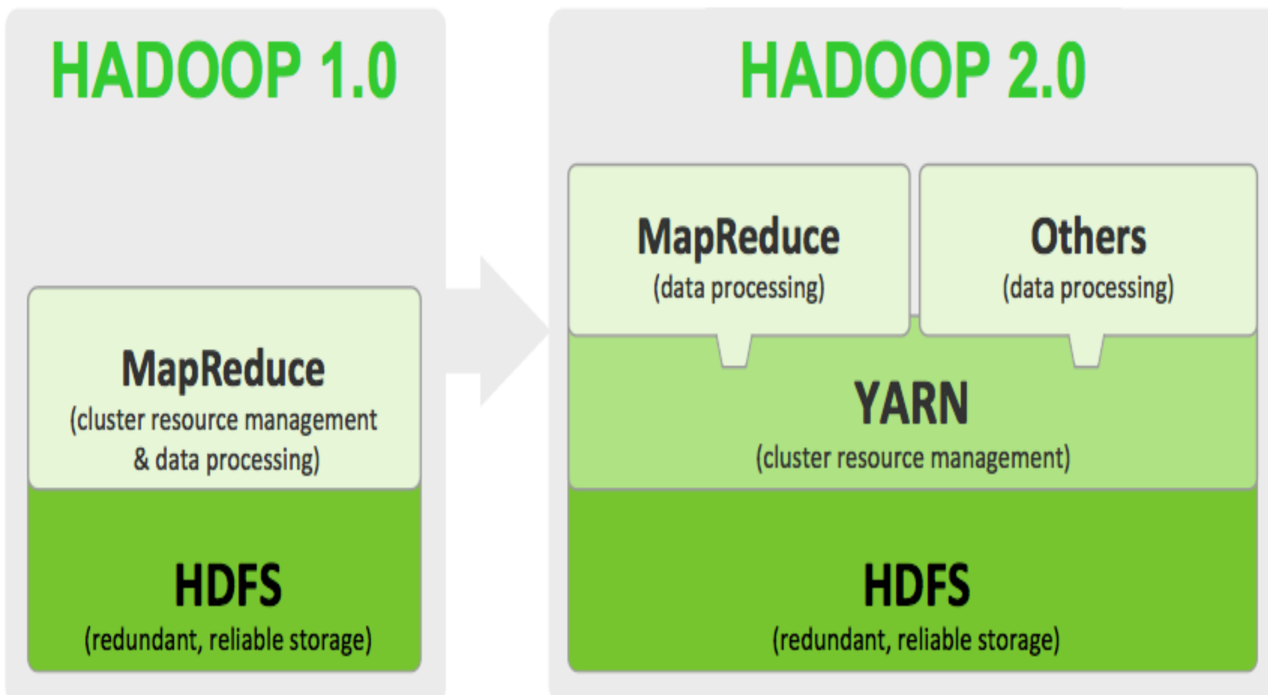
Conclusion

- Mesos shares clusters efficiently among diverse frameworks thanks to two design elements:
  - Fine-grained sharing at the level of tasks
  - Resource offers, a scalable mechanism for application-controlled scheduling
- Enables co-existence of current frameworks and development of new specialized ones
- In use at Twitter, UC Berkeley, Conviva and UCSF

4: Yarn

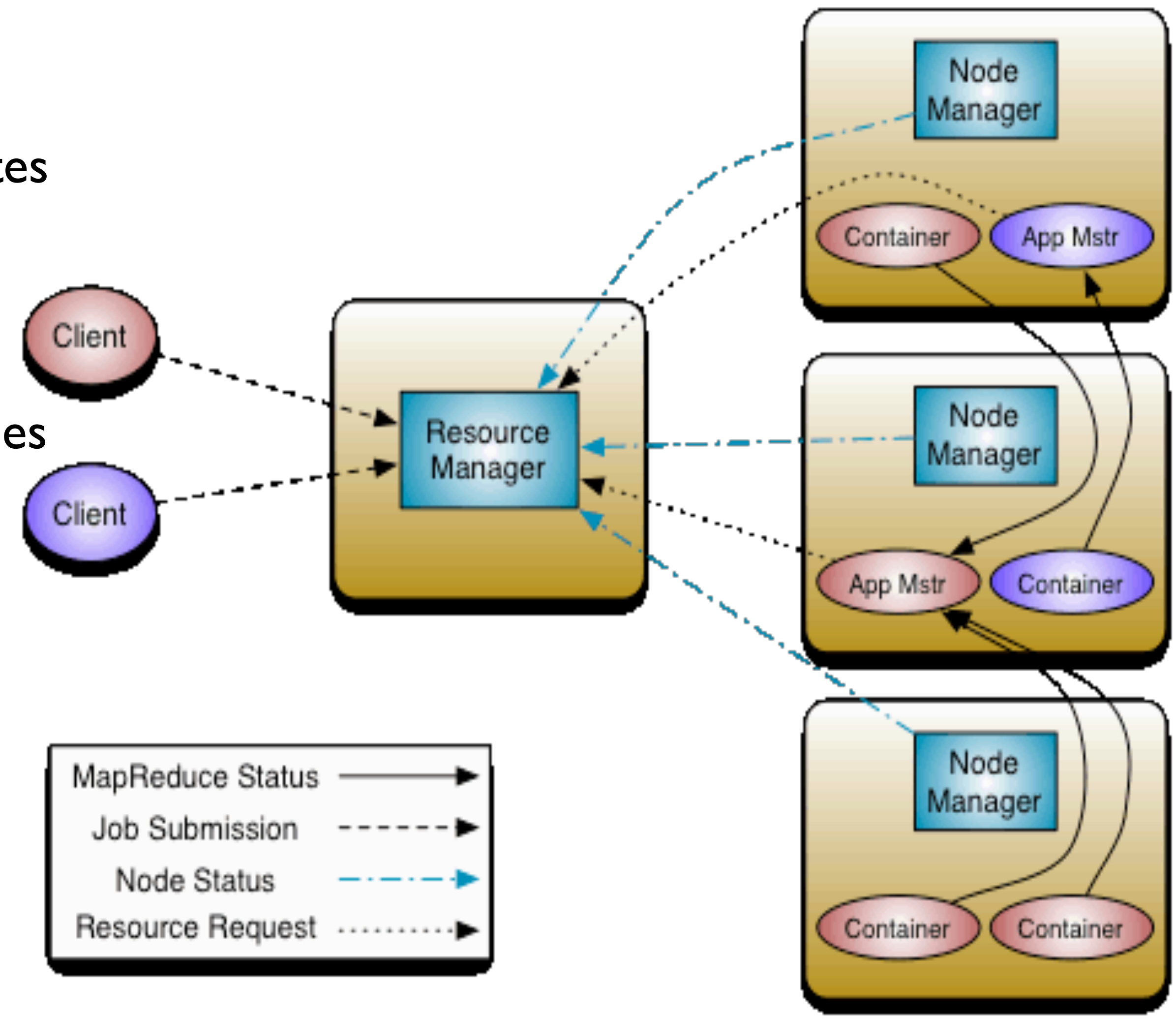
YARN - Yet Another Resource Negotiator

- Next version of MapReduce or MapReduce 2.0 (MRv2)
- In 2010 group at Yahoo! Began to design the next generation of MR



YARN architecture

- **Resource Manager**
  - Central Agent – Manages and allocates cluster resources
- **Node Manager**
  - Per-node agent – Manages and enforces node resource allocations
- **Application Master**
  - Per Application
  - Manages application life cycle and task scheduling



YARN – Resource Manager Failure

- After a crash a new Resource Manager instance needs to brought up (by an administrator)
- It recovers from saved state
- State consists of
  - node managers in the systems
  - running applications
- State to manage is much more manageable than that of Job Tracker.
  - Tasks are not part of Resource Managers state.
  - They are handled by the application master.