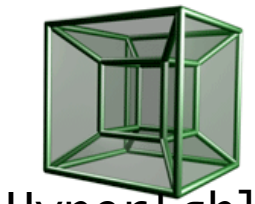
Multi-level scheduling

15-719

Greg Ganger
Garth Gibson
Majd Sakr

Context: many execution frameworks

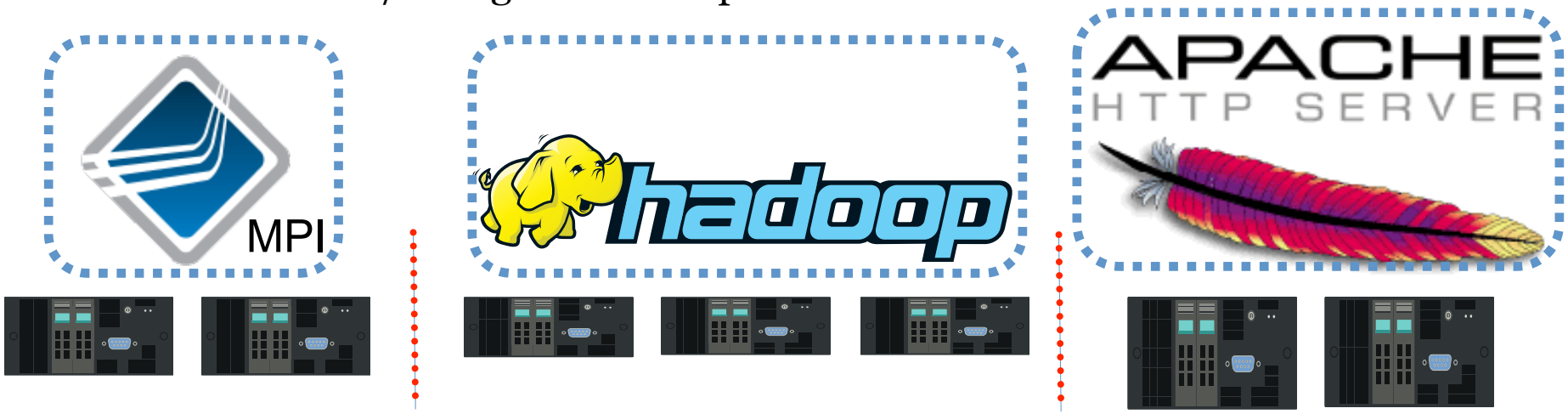
- There are many cluster resource consumers
  - Big Data frameworks, elastic services, VMs, …
  - Number going up, not down: GraphLab, Spark, …



Dryad

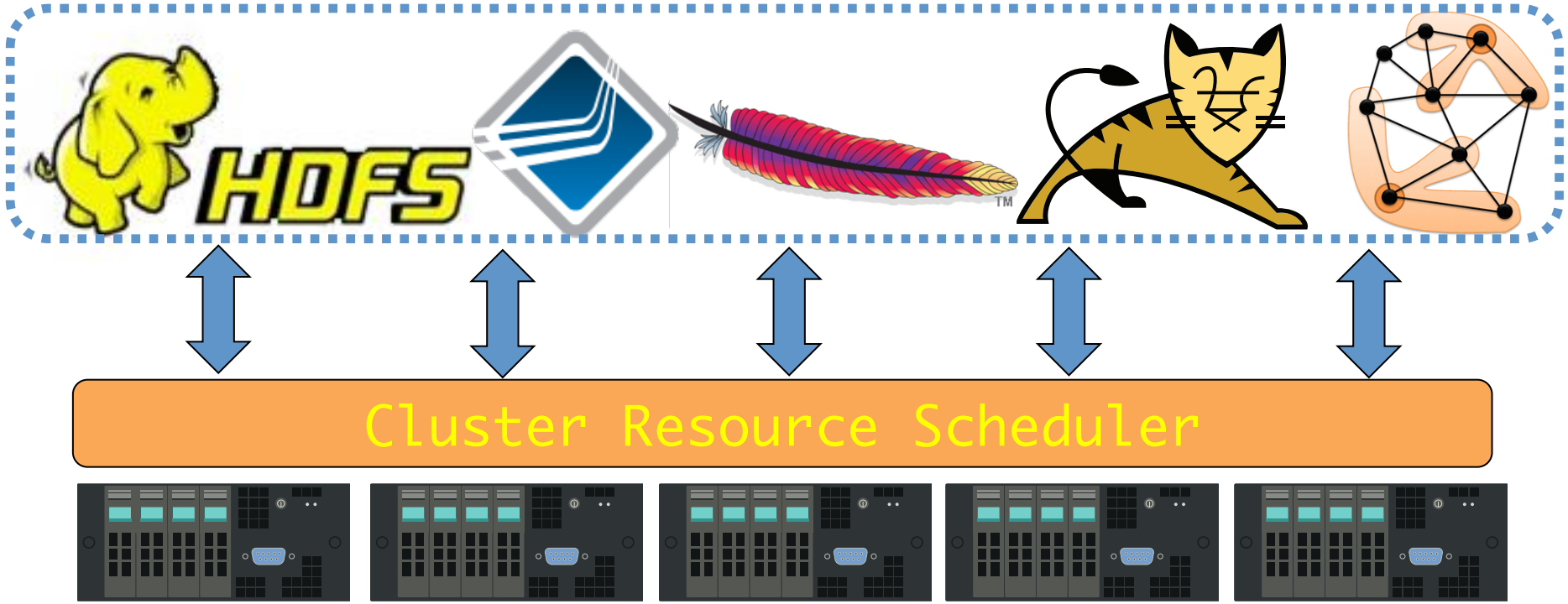
Traditional: separate clusters

- There are many cluster resource consumers
  - Big Data frameworks, elastic services, VMs, …
  - Number going up, not down: GraphLab, Spark, …
- Historically, each would get its own cluster
  - and use its own cluster scheduler
  - and hardware/configs could be specialized



Preferred: dynamic sharing of cluster

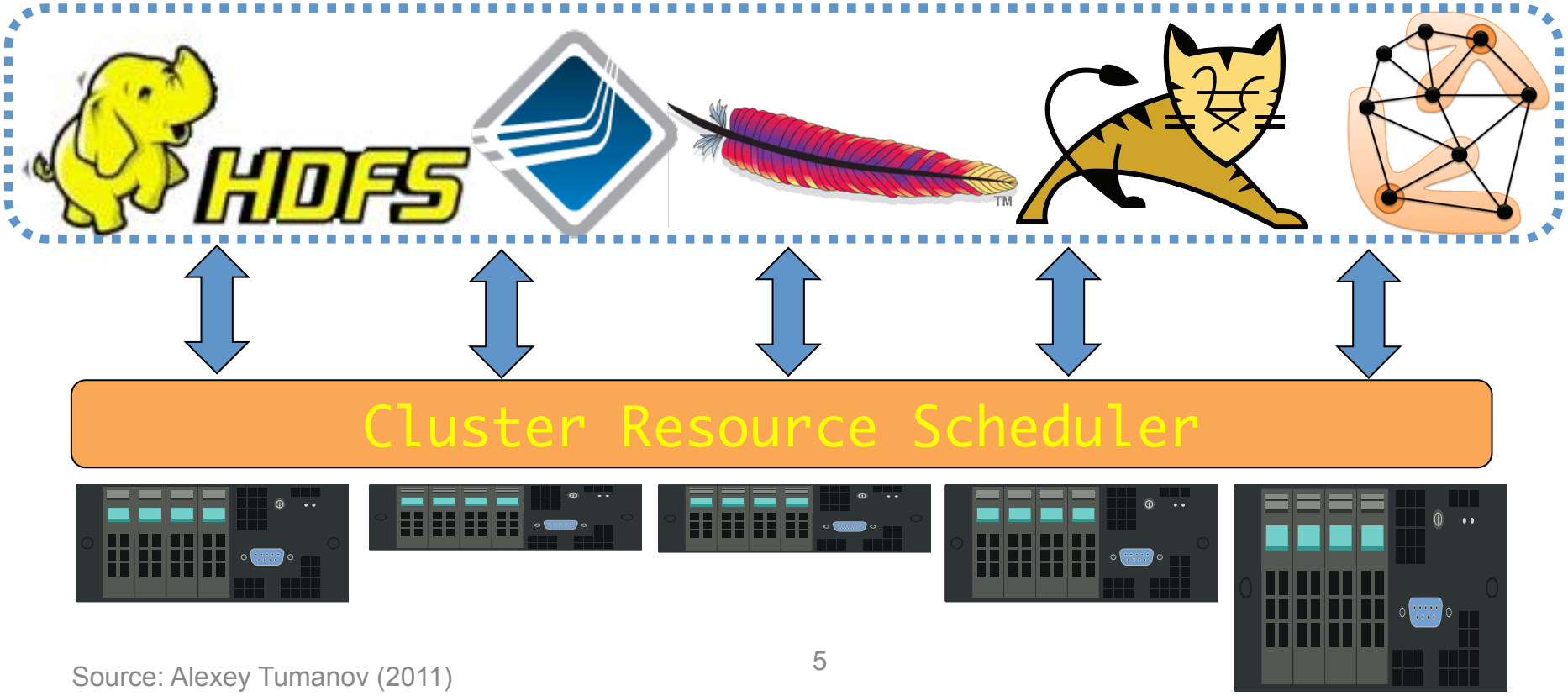
- Heterogeneous mix of activity types
  - Some long-lived HA services; others short-lived batch jobs w/ lots of tasks
- Each grabbing/releasing resources dynamically
  - Why? all the standard cloud efficiency story-lines



Source: Alexey Tumanov (2011)

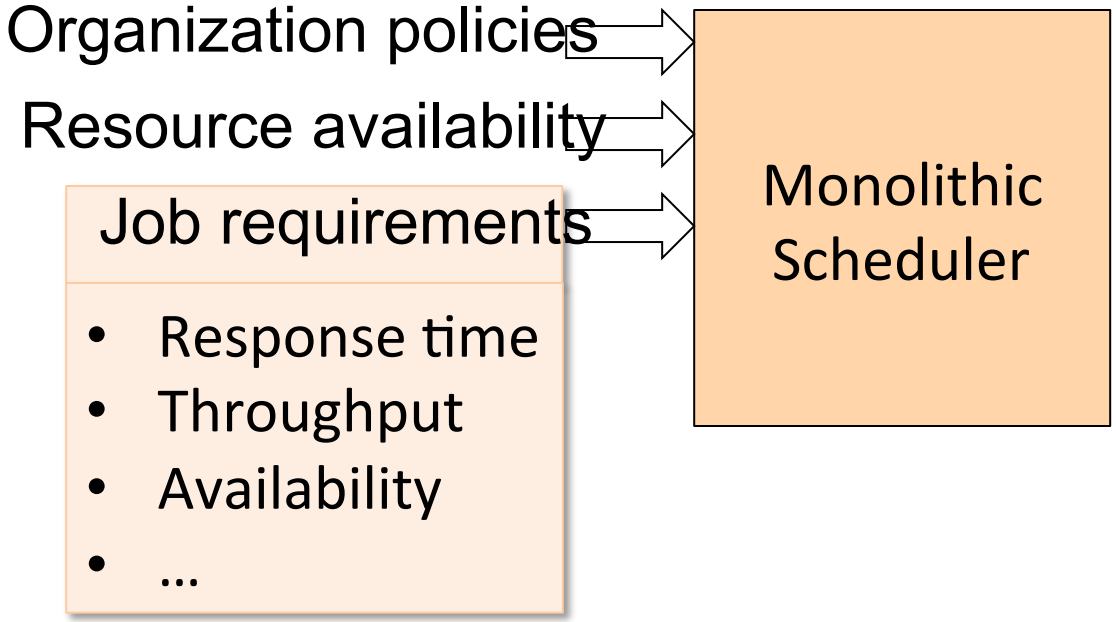
And, INTRA-cluster heterogeneity

- Have a mix of platform types, purposefully
  - Providing a mix of capabilities and features
  - Then, match work to platform during scheduling

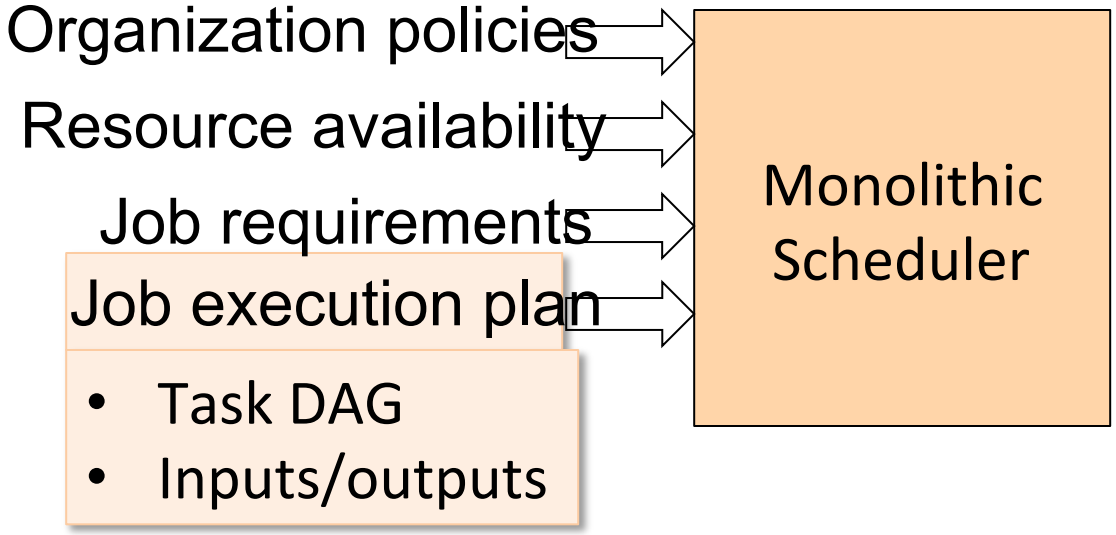


Source: Alexey Tumanov (2011)

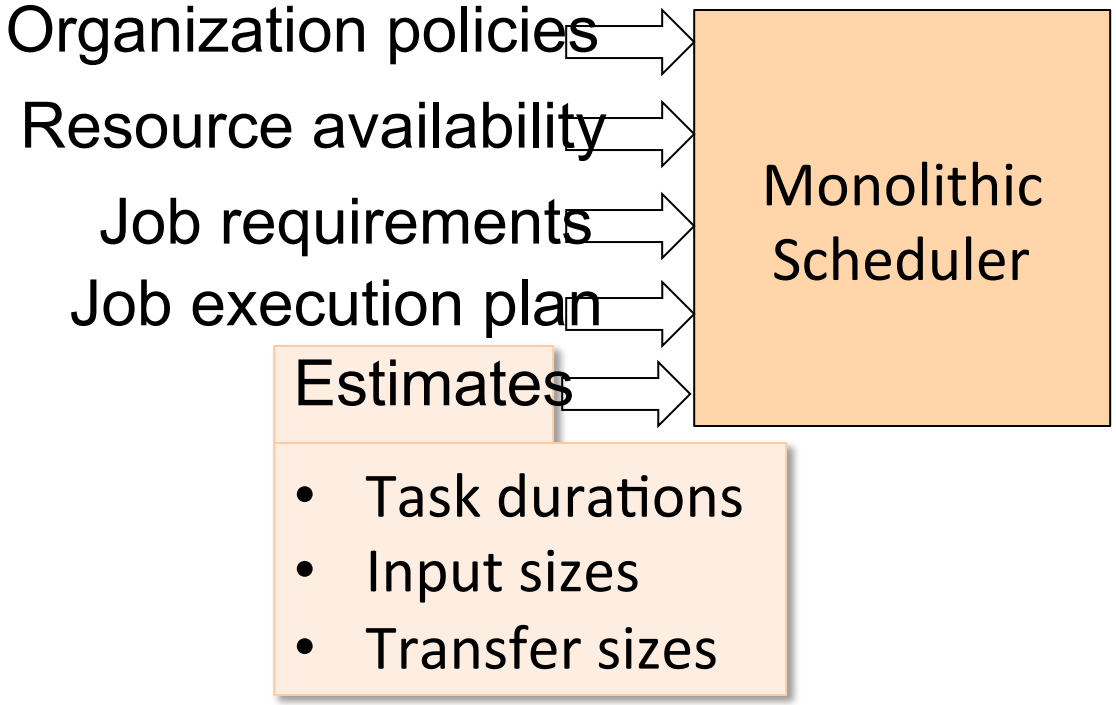
Could (try to) do with a monolithic scheduler



Could (try to) do with a monolithic scheduler

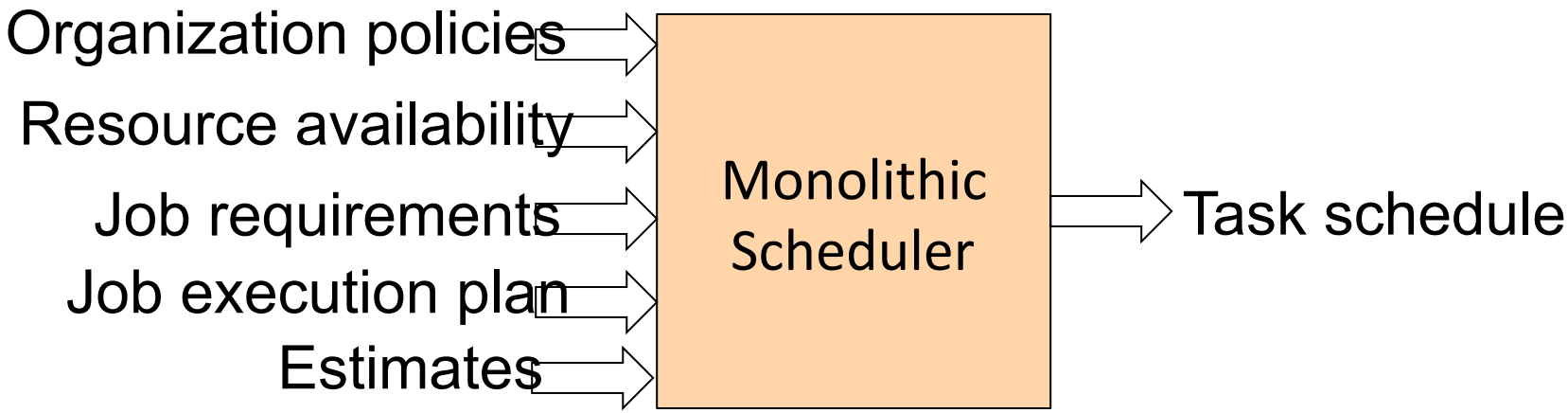


Could (try to) do with a monolithic scheduler



Source: Ion Stoica (2012)

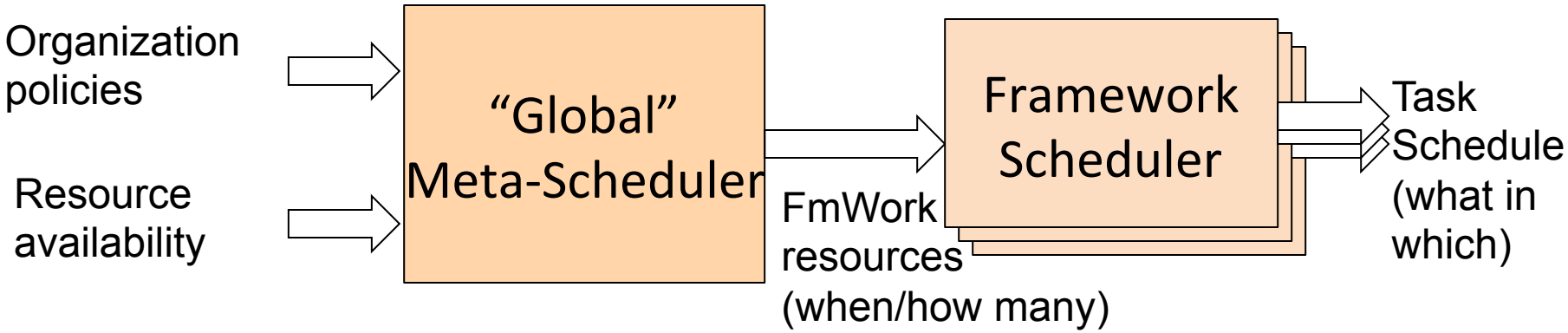
Could (try to) do with a monolithic scheduler



- Advantages: can (theoretically) achieve optimal schedule
- Disadvantages:
  - Complexity → hard to scale and ensure resilience of scheduler
  - Hard to anticipate future frameworks' requirements
    - Scheduler can only consider what it is programmed to consider
  - Need to refactor existing frameworks to yield control to central scheduler

Source: Ion Stoica (2012)

One alternative: two-level schedulers

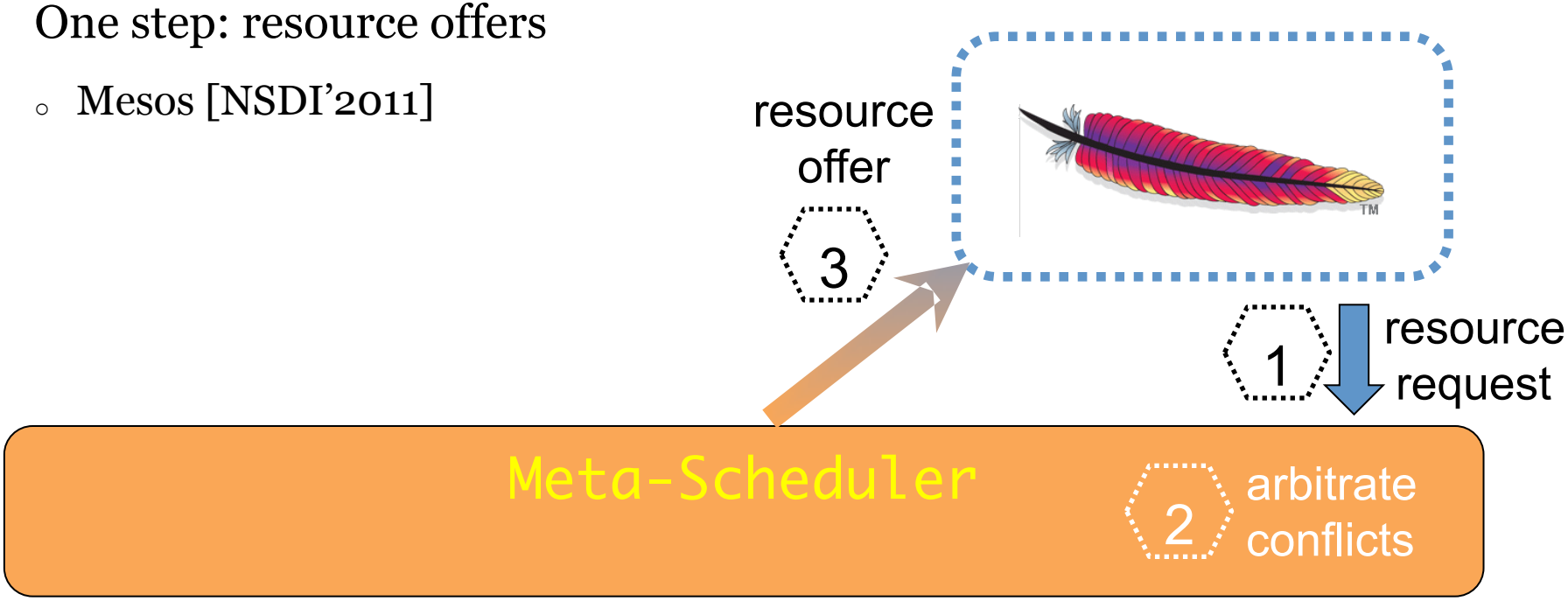


- Advantages:
  - Simple → easier to scale and make resilient
  - Easier to port existing frameworks, support new ones
- Disadvantages:
  - Distributed scheduling decision → may be suboptimal
  - Need to balance awareness with coordination overhead

Source: Ion Stoica (2012)

Two-level allocation decisions (how they can work)

- Framework - meta-scheduler interaction
  - *meta-scheduler:* determines **when** and **how much**
  - *framework:* chooses **which (**and what to do where)
- One step: resource offers
  - Mesos [NSDI'2011]



Source: Alexey Tumanov (2011)

Resource offer mechanics

- Unit of allocation: ***resource offer***
  - Vector of available resources on a node
  - E.g., node1: <1CPU, 1GB>, node2: <4CPU, 16GB>

- Meta-scheduler sends resource offers to frameworks

- Frameworks select which (if any) offers to accept and which tasks to run

Keep task scheduling in frameworks

Challenges with two-level schedulers

- Allocation changes
  - When circumstances change, the right decisions might too
    - e.g., new requests with higher priority or with restrictive constraints
  - How does the meta-scheduler arbitrate among framework schedulers?
- Planning ahead
  - lack of central planning of schedule can lead to distributed hoarding
- Limited visibility for frameworks into overall cluster state
  - this one is more easily fixed, by just making frequent requests
  - but, there's a performance cost

Alternate distributed scheduler arch: shared state

- Expose cluster state and schedule to all framework schedulers
  - Update their views when it changes
- Let each framework make decisions independently
  - Use optimistic concurrency control when trying to change schedule
- Allow scheduling into future
  - So, a hard-to-schedule job can be scheduled without distributed hoarding
  - Other schedulers can fill in the schedule before the one that is later
    - This is sometimes called “back filling” in scheduling

Challenges with shared state schedulers

- Performance overheads in maintaining shared state
  - May not be too much, but “it depends”
    - note that requesting offers is “pull-based” and shared state is “push-based”
- Can repeat work
  - Due to the optimistic concurrency control… may or may not be too bad
- Allocation changes
  - how does one arbitrate/negotiate among separate schedulers?

Wrap-up for this part

- So, schedulers can be centralized or distributed
  - Which do you think is the most common? Why?

- Hey, we're not done today yet!
- Next up: Majd on YARN, as a concrete example

Apache Hadoop YARN: Yet Another Resource Negotiator

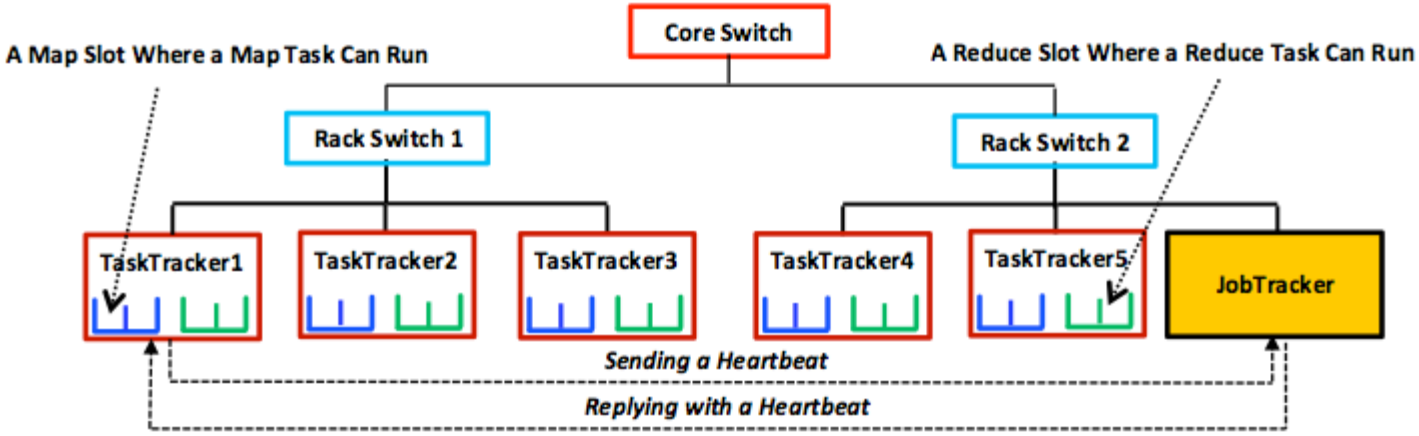
Majd Sakr, Garth Gibson, Greg Ganger

15-719/18-849b Advanced Cloud Computing
Spring 2017

February 22, 2017

Apache Hadoop MapReduce

- MapReduce jobs
- Single master for all jobs, JobTracker
  - Resource allocator and job scheduler
- One or many slaves, TaskTrackers
  - Configurable number of Map task slots and Reduce task slots



Apache Hadoop MapReduce

- Designed to run large MapReduce jobs
- Limitations:
  - Single Programming Model (MapReduce)
  - Centralized handling of jobs
    - SPOF, JobTracker failure kills all running & pending jobs
    - Scalability concerns
      - Bottleneck for ~10K jobs
  - Resources (task slots) were specific to either
    - Map tasks
    - Reduce tasks

Apache Hadoop YARN

- Supports multiple programming models
  - Dryad, Giraph, MapReduce, REEF, Spark, Storm
- Two-Level Scheduler
  - Cluster resource management detached from job management (meta-scheduler)
    - Cluster resource manager
  - One master per job (framework-scheduler)
    - Application lifecycle management
- Dynamic allocation of resources to run any tasks

YARN Requirements

1. Scalability
2. Multi-tenancy
3. Serviceability
4. Locality awareness
5. High cluster utilization
6. Reliability/Availability
7. Secure and auditable operation
8. Support for programming model diversity
9. Flexible resource model
10. Backward compatibility

YARN Architecture

- Resource Manager (***RM***)
  - Cluster resource scheduler
- Application Master (***AM***)
  - One per job
  - Job life-cycle management
- Node Manager (***NM***)
  - One per node
  - Container life-cycle management
  - Container resource monitoring

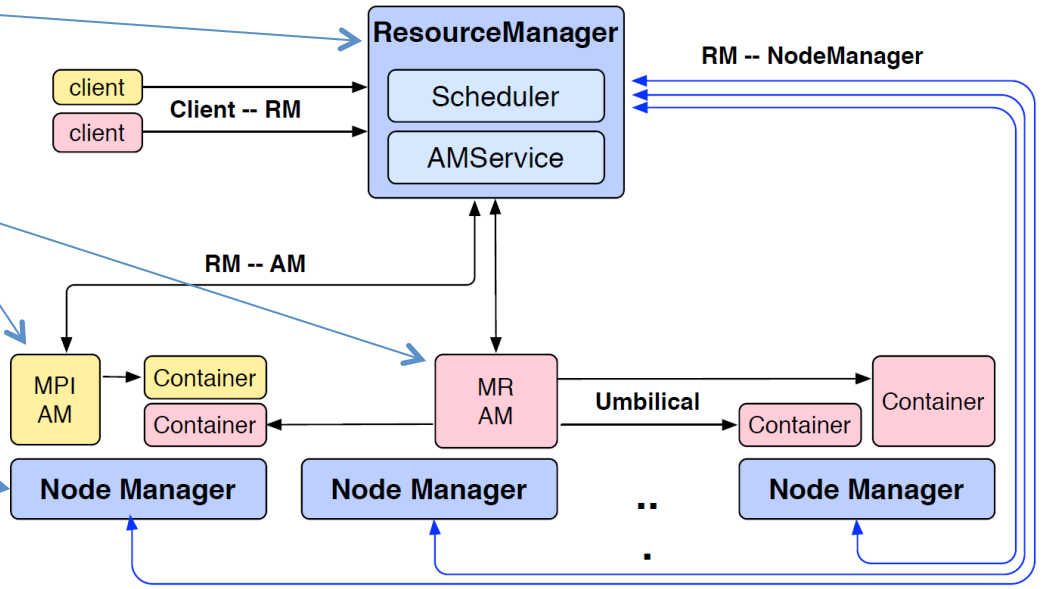


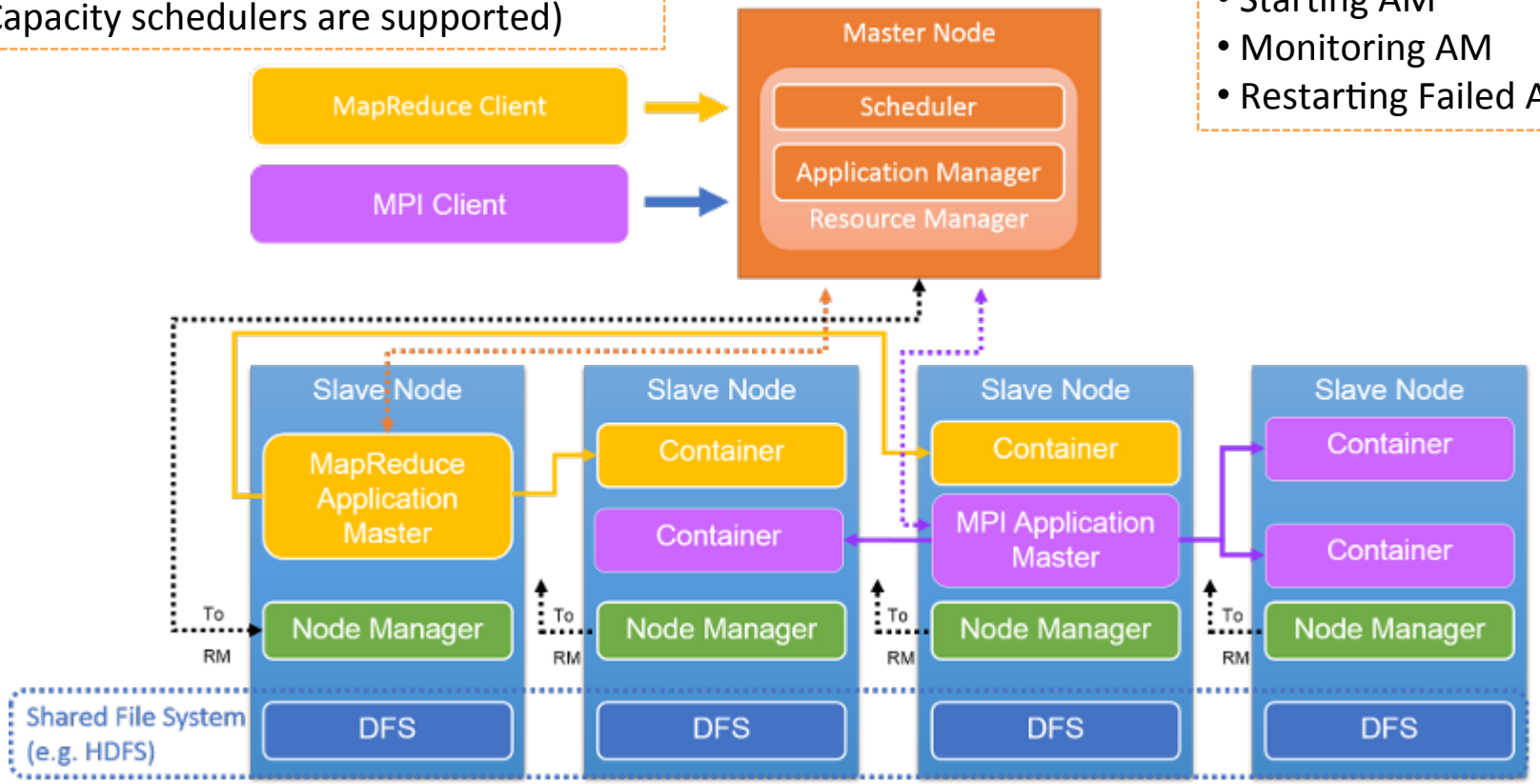
Figure 1: YARN Architecture (in blue the system components, and in yellow and pink two applications running.)

Vavilapalli, et al., "Apache Hadoop YARN: yet another resource negotiator." SOCC '13 http://doi.acm.org/10.1145/2523616.2523633

YARN

Scheduler: Manages and enforces the resource scheduling policy (Fair and Capacity schedulers are supported)

<u>Application Manager</u>
Manages running the AM:
- Starting AM
- Monitoring AM
- Restarting Failed AM



Resource Manager

- One per cluster
- Request-based scheduler
- Tracks resource usage and node liveness
- Enforces allocation and arbitrates contention among competing jobs
  - Fair, Capacity
  - Locality
- Dynamically allocates leases to applications
- Interacts with NodeManagers to get to assemble a global view
- Can reclaim allocated resources by
  - Collaborating with AMs
  - Killing containers directly through the NM

Application Master

- One per job
  - Manages lifecycle of a job
  - Creates a logical plan of the job
  - Requests resources through a heartbeat to the RM
  - Receives a resource lease from the RM
  - Creates a physical plan
  - Coordinates execution
  - Plans around faults

Application Master

- At any given time, there will be as many running AMs as jobs
- Each AM manages the job's individual tasks
  - Starting, monitoring, and restarting tasks
  - Each task runs within a ***container*** on each NM
    - Containers can be compared to slots in Hadoop MapReduce
      - Static allocation of slots vs. dynamic allocation of containers
      - Slots were for specific tasks (map or reduce) vs. containers
- The AM acquires resources dynamically in the form of containers from the RM's scheduler before contacting corresponding NMs to start a job's tasks
  - Each container has a number of non-static attributes
    - CPU
    - Memory
    - …

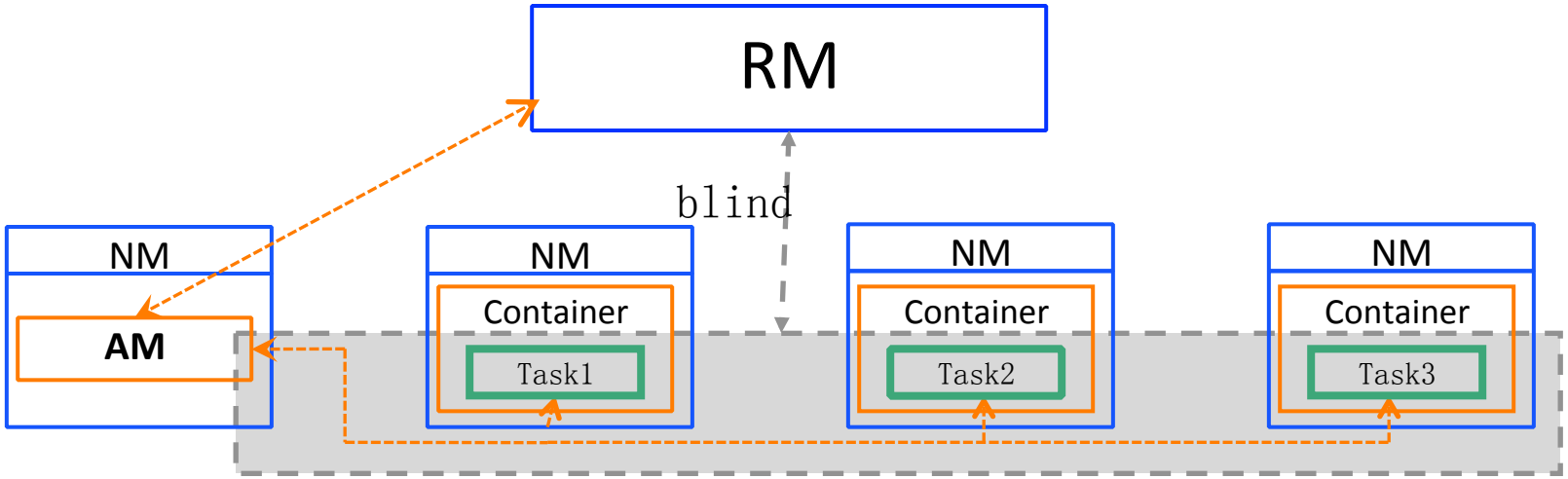
Node Managers & Containers

- Node Manager manages container lifecycle and monitors containers
  - One per node
  - Authenticates container leases
  - Monitors container execution
  - Reports usage through heartbeat to RM
  - Kills containers as directed by RM or AM

Node Managers & Containers

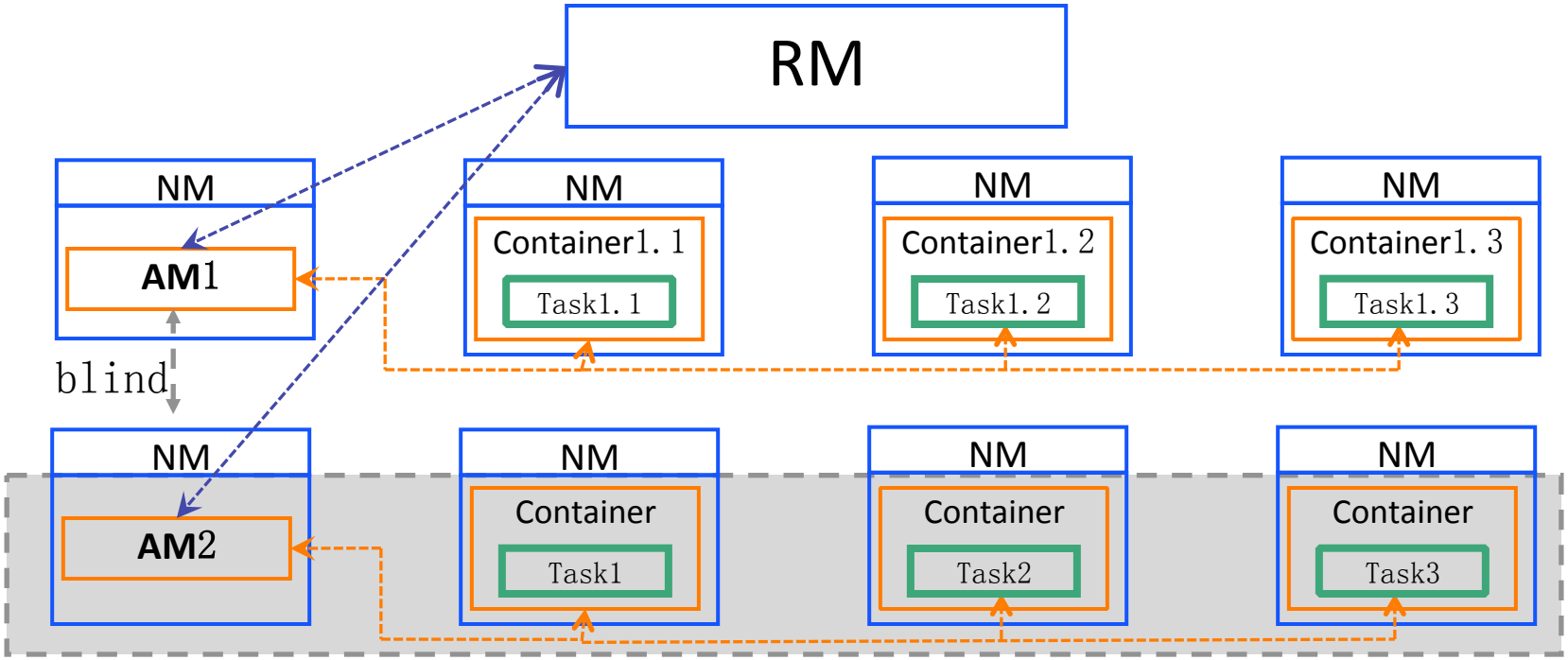
- Container represents a lease for an allocated resource in the cluster
  - Logical bundle of resources bound to a node
- The RM is the sole authority to allocate any container to applications
- The allocated container is always on a single NM and has a unique ContainerId
- A container includes details such as:
  - ContainerId for the container, which is globally unique
  - NodeId of the node on which it is allocated
  - Resource allocated to the container
  - Priority at which the container was allocated
  - ContainerState of the container
  - ContainerToken of the container, used to securely verify authenticity of the allocation
  - ContainerStatus of the container

Visibility of the Resource Manager (RM)



- When a job is submitted, the RM assigns a jobID to it and allocates a container to run the corresponding AM.
- The AM then asks for resources to run its job. After it gets the lease, the AM starts tasks and assigns tasks to containers.
- RM is blind to the tasks running within an application.

Visibility of the Application Master (AM)

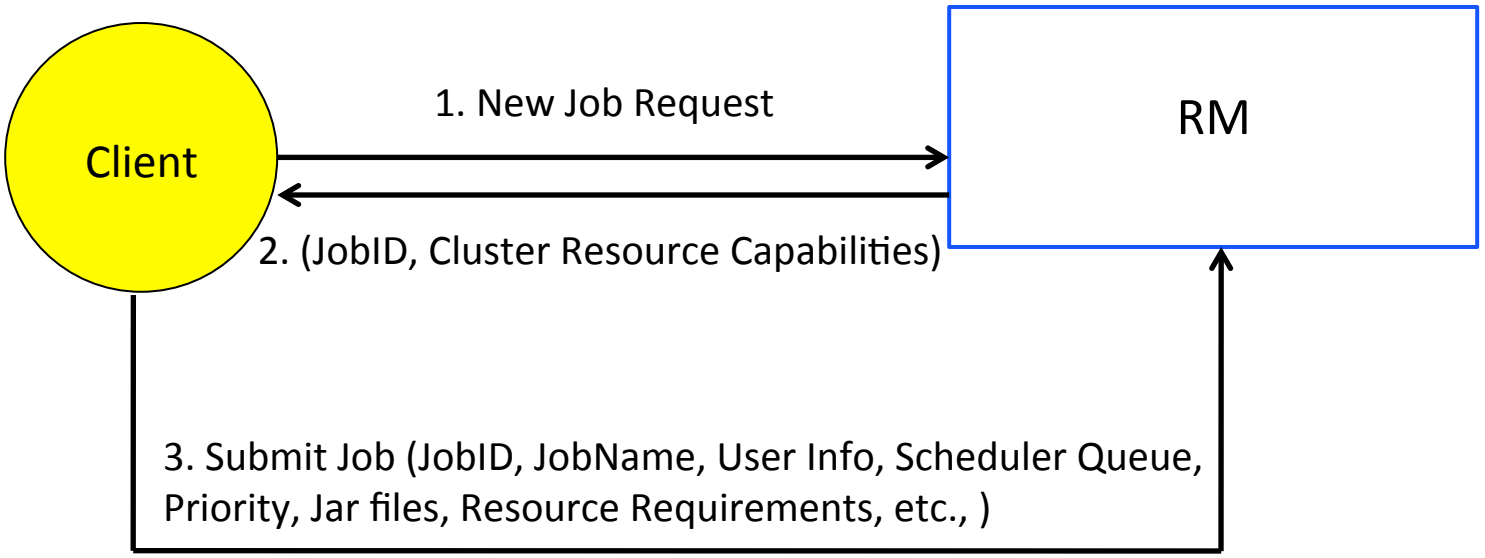


- AM is like the job tracker in Hadoop 1.0
  - Creates and manages task lifecycle
  - Monitors task status
- AM has no view of other running applications.

Protocols

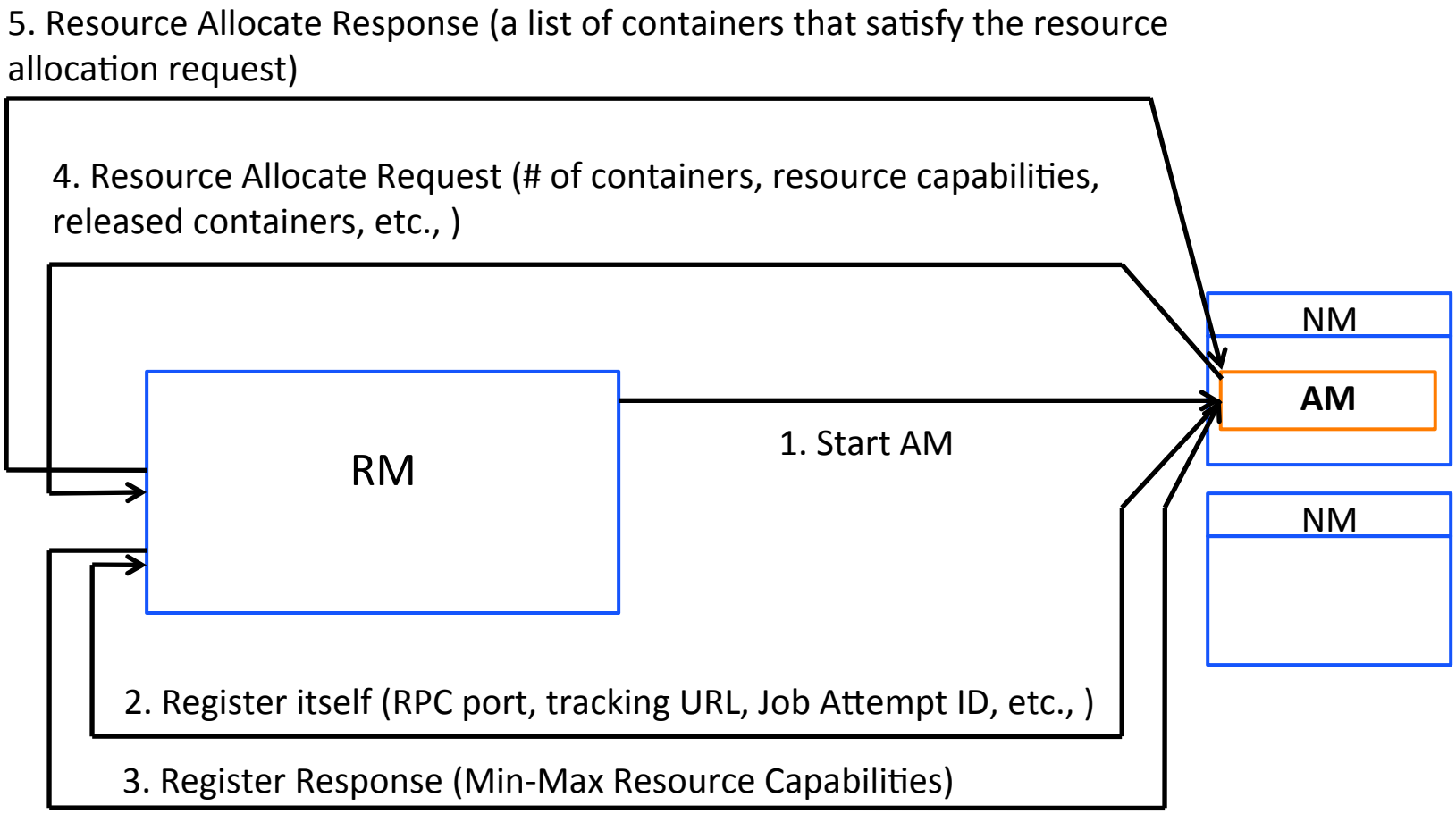
- YARN interfaces:
  - Client-RM Protocol: This is the protocol for the client to communicate with the RM to launch a new job, check on the status of the job, and/or kill a job
  - AM-RM Protocol: This is the protocol used by the AM to register/unregister itself with the RM, as well as to request resources from the RM scheduler to run its tasks
  - AM-NM Protocol: This is the protocol used by the AM to communicate with the NM to start/stop containers
  - NM-RM Protocol: This is the protocol used by the NM to communicate its status to the RM
- All client-facing MapReduce interfaces are unchanged, which means that there is no need to make any source code changes to run on top of YARN

Client-RM Protocol

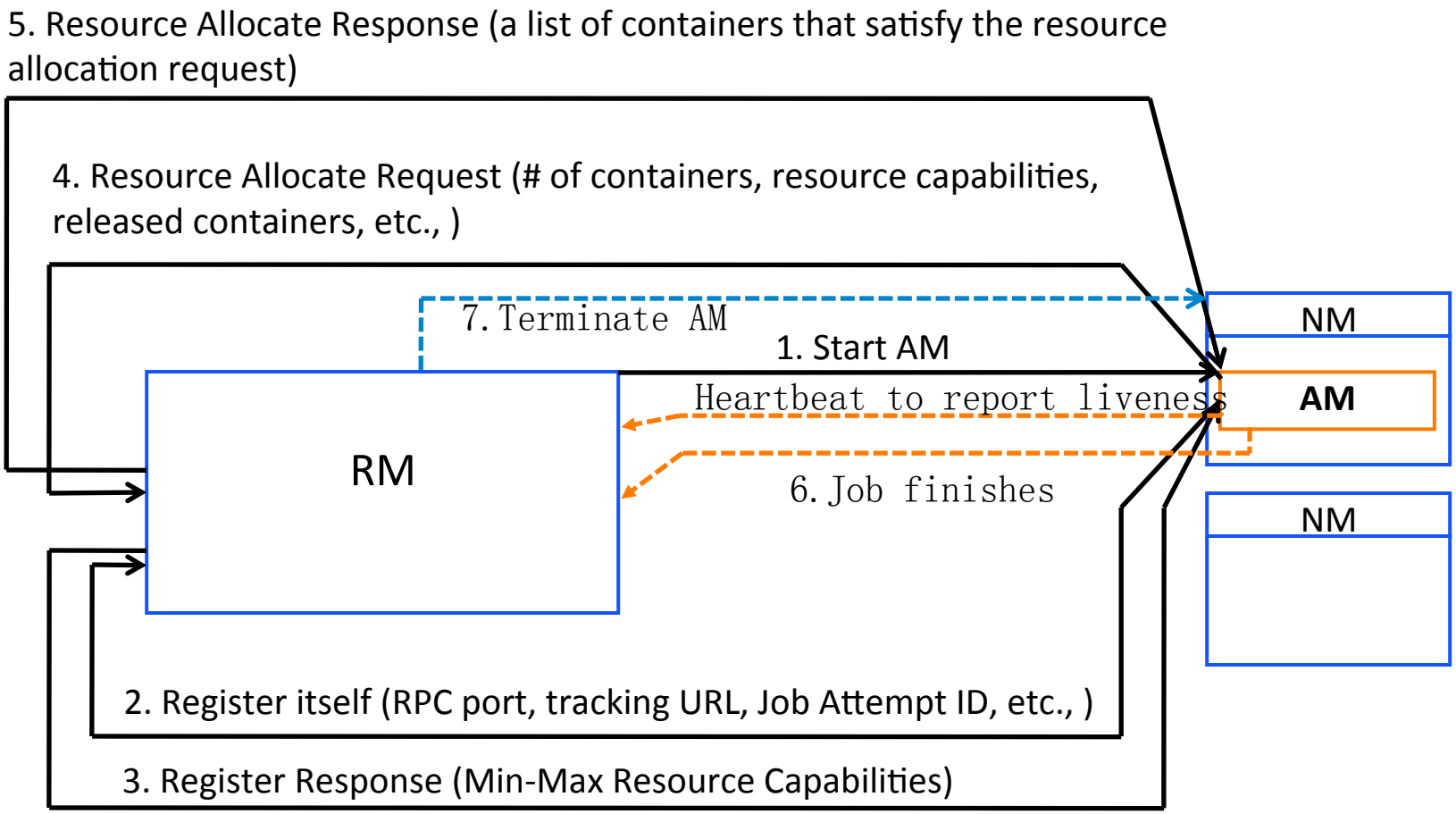


- When the RM receives the job submission context (i.e., request 3 in the above example), it finds an available container (the job's first container) for running an AM for the requested job

RM-AM Protocol



RM-NM Protocol



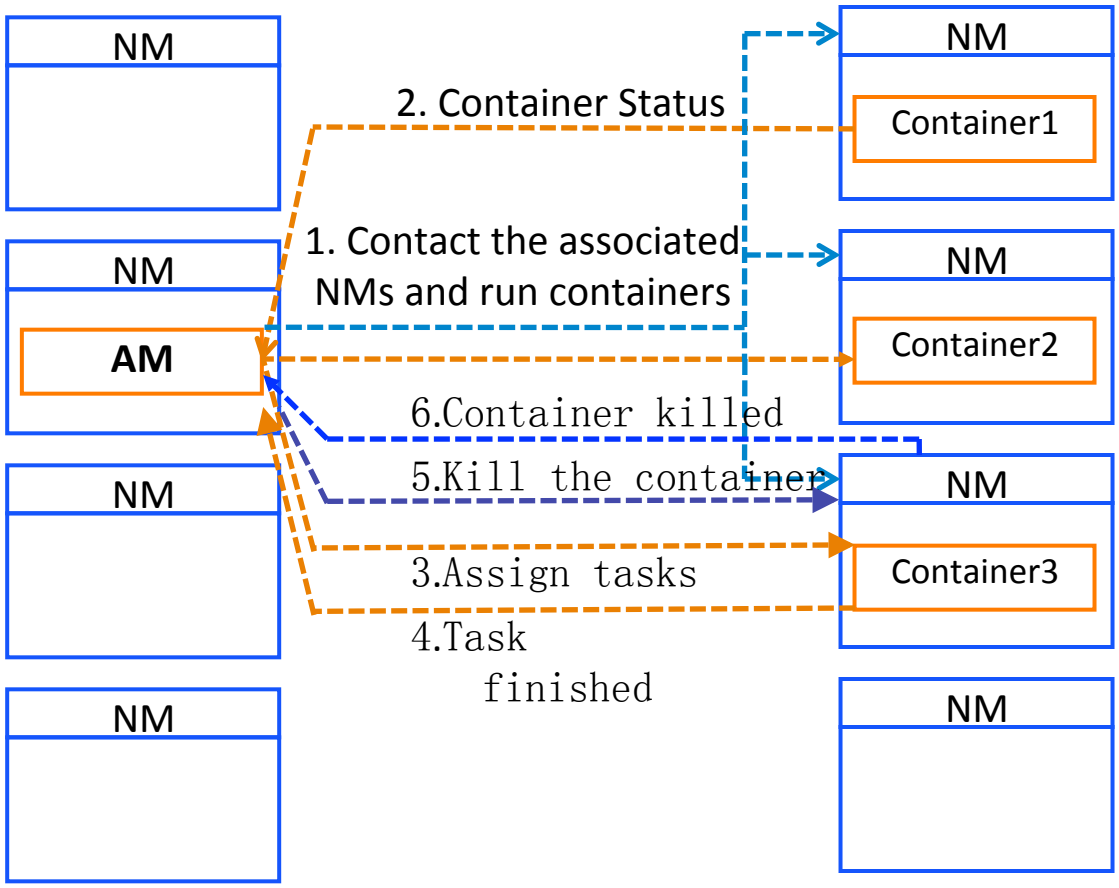
Resource Request: An Example

Priority	(Host, Rack, *)	Resource Requirements (memory in GB, # CPUs)	Number of Containers
1	Node12	1GB, 1CPU	5
1	Rack11	1GB, 1 CPU	8
2	*	2GB, 1 CPU	3

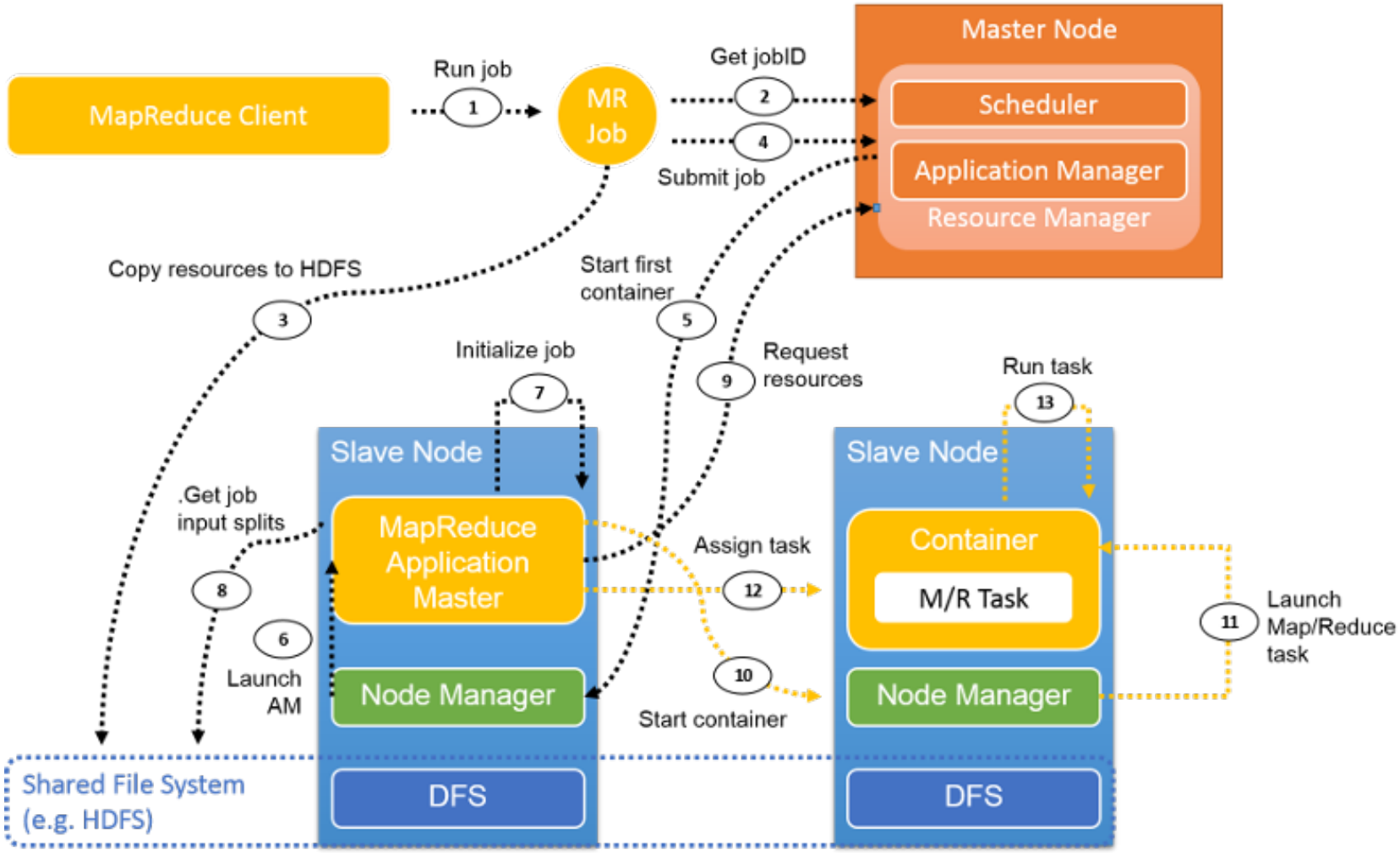
- In the MapReduce case, the MapReduce AM takes the input-splits and presents to the RM Scheduler an inverted table keyed on the hosts, with limits on total containers it needs in its life-time, which is subject to change

- The protocol understood by the Scheduler is
  *<priority, (host, rack, *), resources, #containers>*

AM-NM Protocol



The Lifecycle of a MR Job in YARN



The Lifecycle of a MR Job in YARN

Job submission

1. The MapReduce client uses the same API as Hadoop version 1.0 to submit a job to YARN.
2. The new job ID is retrieved from the RM. However, sometimes a jobID in YARN is also called applicationID.
3. Necessary job resources, such as the job JAR, configuration files, and split information are copied to a shared file system in preparation to run the job.
4. The job client calls submitApplication() on the RM to submit the job.

The Lifecycle of a MR Job in YARN

Job initialization

5. The RM passes the job request to its Scheduler. The Scheduler allocates resources to run a container where the Application Master (AM) will reside. Then the RM sends the resource lease to some Node Manager (NM).
6. The NM receives a message form the RM and launches a container for the AM.
7. The AM takes the responsibility of initializing the job. Several bookkeeping objects are created to monitor the job. Afterwards, while the job is running, the AM will keep receiving updates with the progress of its tasks.
8. The AM interacts with the shared file system (e.g. HDFS) to get its input splits and other information which were copied to the shared file system in Step 3.

The Lifecycle of a MR Job in YARN

Task assignment

9. The AM computes the number of map tasks (based on the number of input splits) and the number of reduce tasks (configurable). The AM submits the resource request for the map and reduce tasks along with its heartbeat to the RM. A request includes preferences in terms of data locality (for map tasks), the amount of memory and the number of CPUs in each container.
10. After the RM responds with container leases, the AM communicates with the NMs.
11. The NMs start the containers.
12. The AM assigns a task to this container based on its knowledge of locality.
13. The task runs in the container. The MapReduce AM monitors the individual tasks to completion, requests alternate resources if any of the tasks fail or stop responding.

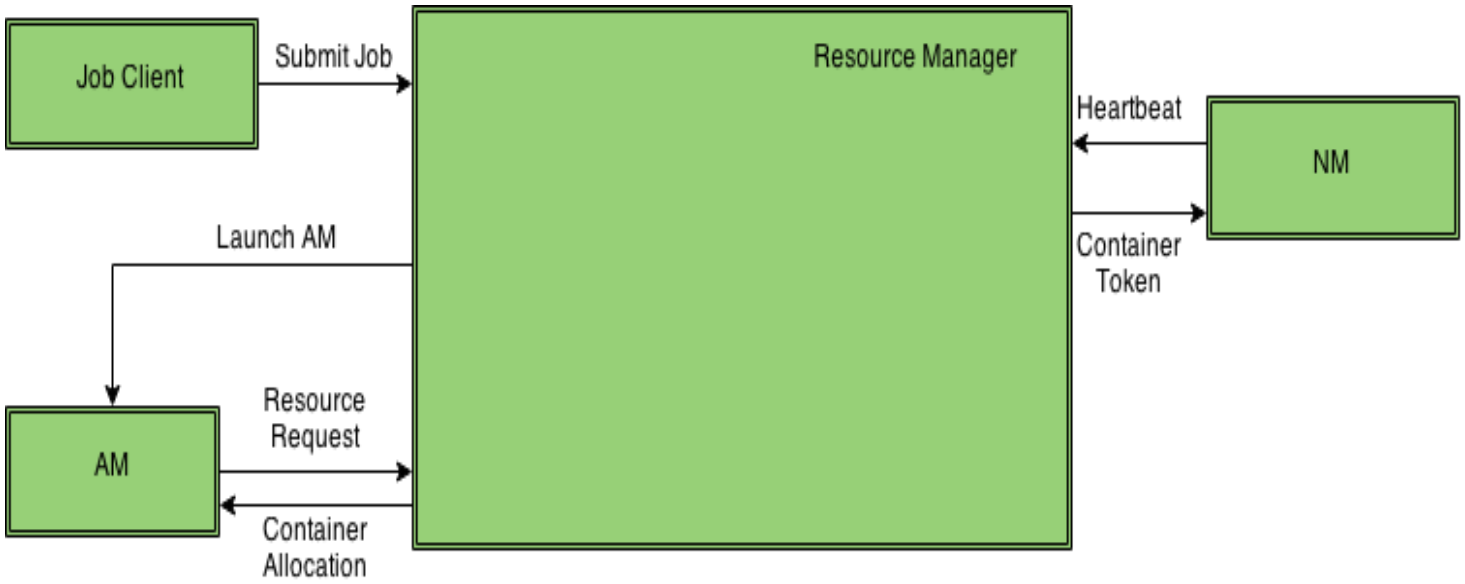
The Lifecycle of a MR Job in YARN

Job Completion

- The MapReduce AM also runs appropriate task cleanup code of completed tasks
- Once the entire map and reduce tasks are complete, the MapReduce AM runs the requisite job commit
- The MapReduce AM informs the RM then exits since the job is complete

Scheduling in YARN

- The resource manager has a pluggable scheduler.
- The default version of YARN has three schedulers
  - FIFO Scheduler, Fair Scheduler and Capacity Scheduler.
- These schedulers have queues which keep track of the requests from different application masters.



YARN Schedulers – 1

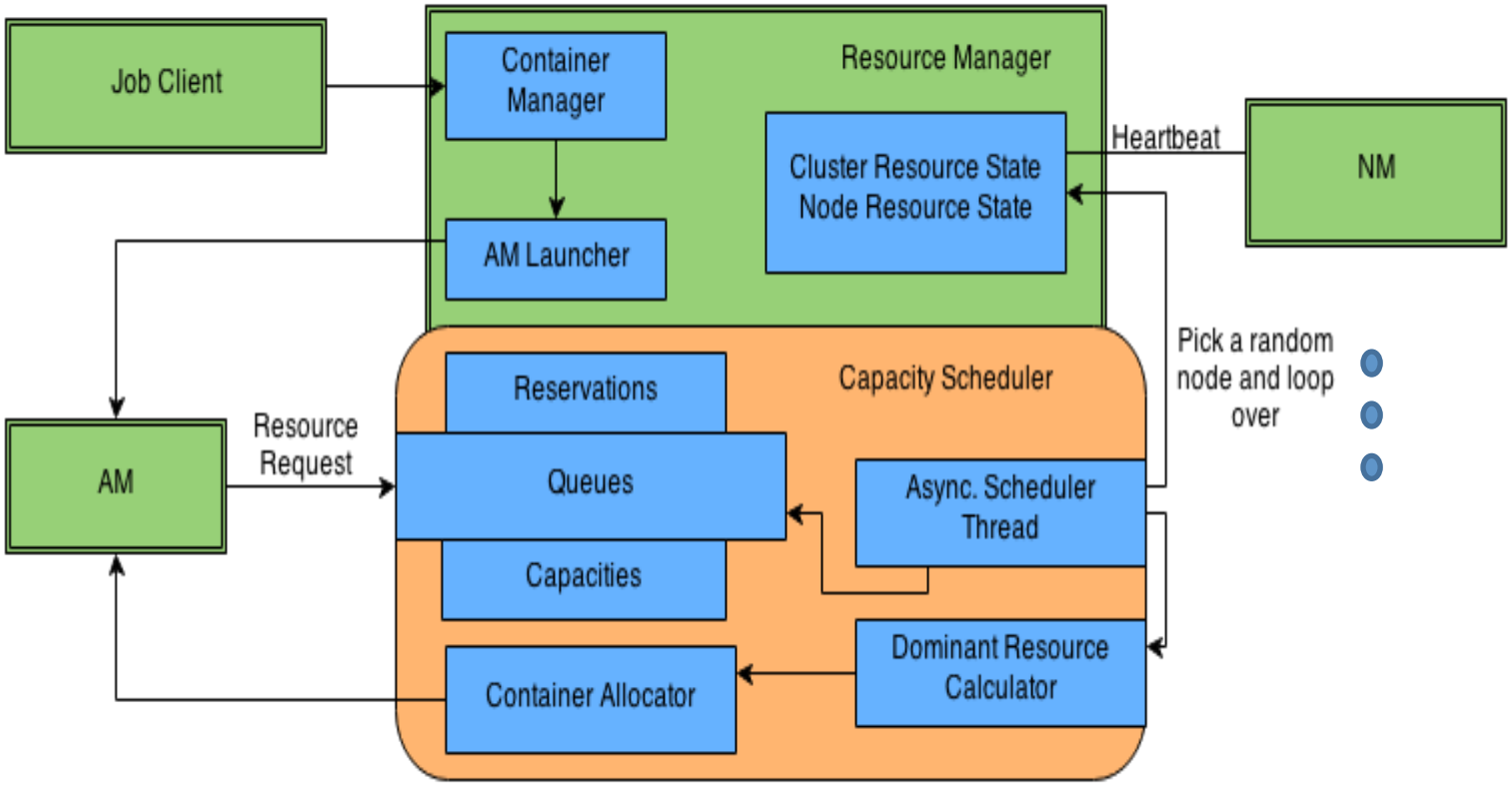
- FIFO Scheduler
  - Has a single first in first out queue used to schedule container requests.

- Fair Scheduler
  - Has multiple queues and tries to fairly allocate resources to the queues.
  - Uses the Dominant Resource Fairness algorithm which ensures that the queue with the lowest share of a particular resource gets the resource.
  - Queues are configurable by the cluster administrator.

YARN Schedulers – 2

- Capacity Scheduler
  - Has multiple queues and tries to allocate resources to the queues such that each queue's capacity constraint is not violated.
  - During initial configuration, the administrator can split the capacity of the cluster's resources among these queues
    - For example, queue_1 gets 25% and queue_2 gets 75% of the resources).
    - So the scheduler will allocate resources such that these capacity configurations are not violated.
    - These queues can belong to different tenants in which case they have access to that particular queue's configuration and settings.

YARN Capacity Scheduler



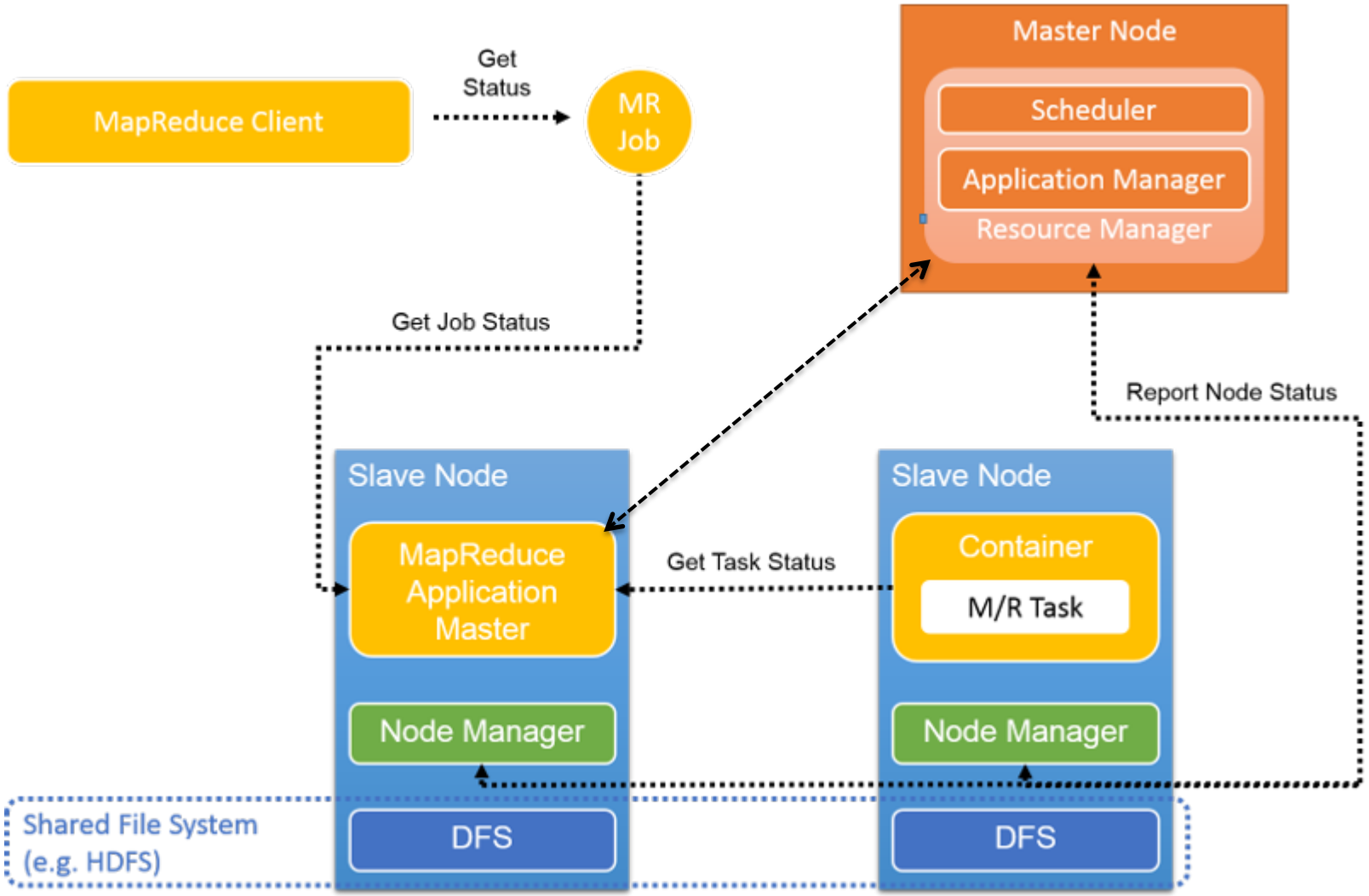
Resource Scheduling – 1

- Resource manager has an asynchronous schedule thread running inside it
  - Responsible for scheduling the container requests from these queues inside the schedulers onto the nodes
- The schedule thread gets a random node from the list of nodes maintained by the resource manager and tries to schedule an application's request on to the node
- The actual container request which gets to run on that particular node is chosen by the scheduler
  - Fair or Capacity

Resource Scheduling – 2

- Once the request is chosen from the queue by the scheduler it checks whether the particular request can be satisfied by the given node
  - This includes checking if the node has enough memory, vcores and locality
    - Same node as the one requested by the application master (AM)
    - Node in the same rack as the requested node
  - If the request can be satisfied, then the container is allocated onto the node and the RM generates a token for the container
    - RM sends token to the AM and the NM
  - If the request cannot be satisfied, then the queue waits for another node to be chosen by the scheduler thread
- Late binding

Heartbeat and Status Reporting in Yarn



Heartbeats

AM to RM:

ResourceRequest: {
Priority: 20,
Resource: {
vCores: 1,
memory: 1024
},
Num Containers: 2,
Desired Host: 192.1.1.1,
Relax Locality: true
}

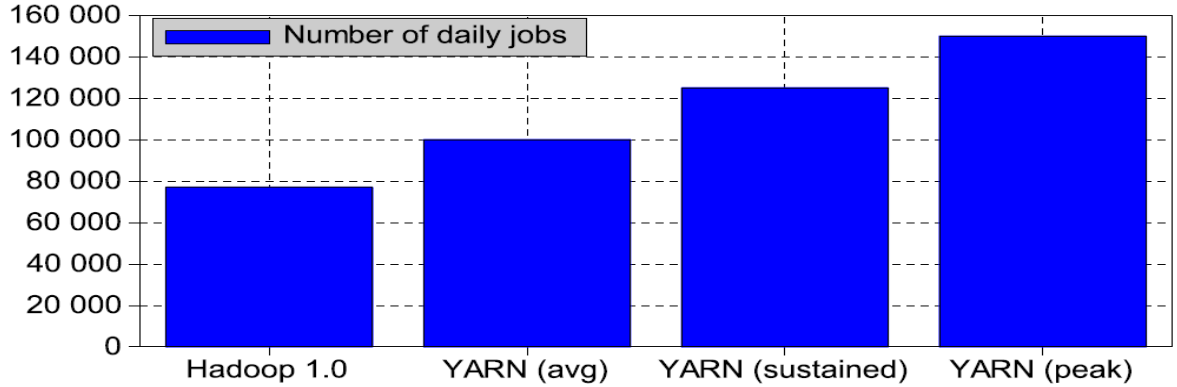
NM to RM:

Register: {
Resource: {
vCores: 1,
memory: 1024
}
}

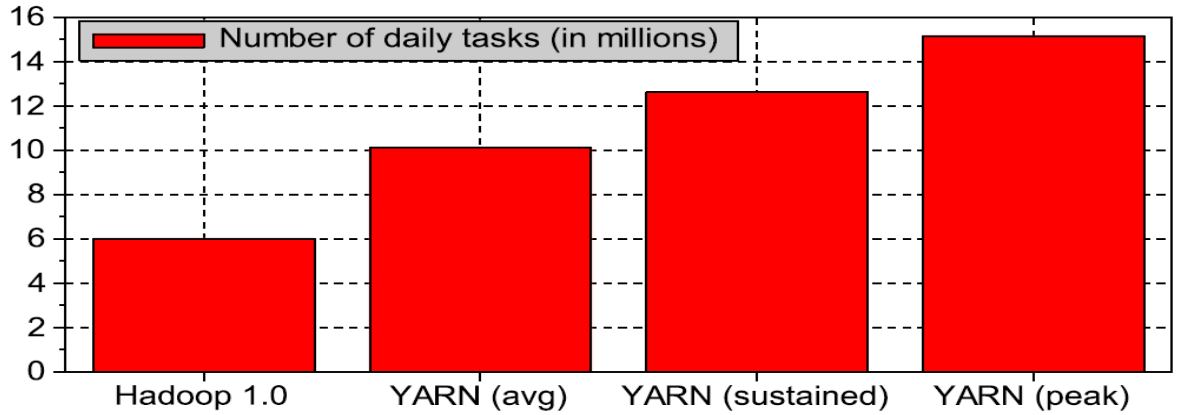
Fault Tolerance

- RM Failure
  - SPOF
  - Can recover from persistent storage
    - Kills all containers including AMs
    - Launches instances for each AM
- NM Failure
  - RM detects through heartbeat timeout
  - Marks all containers on NM killed
  - Reports failure to all running AMs
  - AMs are responsible for node failures
- AM Failure
  - RM restarts AM
  - AM has to resync with all running tasks or all running tasks are killed
- Container failure
  - Framework (AM) responsibility

Hadoop MapReduce vs. Hadoop YARN



(a) Daily jobs



(b) Daily tasks

Figure 2: YARN vs Hadoop 1.0 running on a 2500 nodes production grid at Yahoo!.

Vavilapalli, et al., "Apache Hadoop YARN: yet another resource negotiator." SOCC '13 http://doi.acm.org/10.1145/2523616.2523633

Extensions

- Gang scheduling needs
- Soft/hard constraints to express arbitrary co-location or disjoint placement.
- Heterogeneous resources
- Cost model
- …