14-736: Distributed Systems

Lecture 20 * Spring 2019 * Kesden

Today's lecture is based upon:

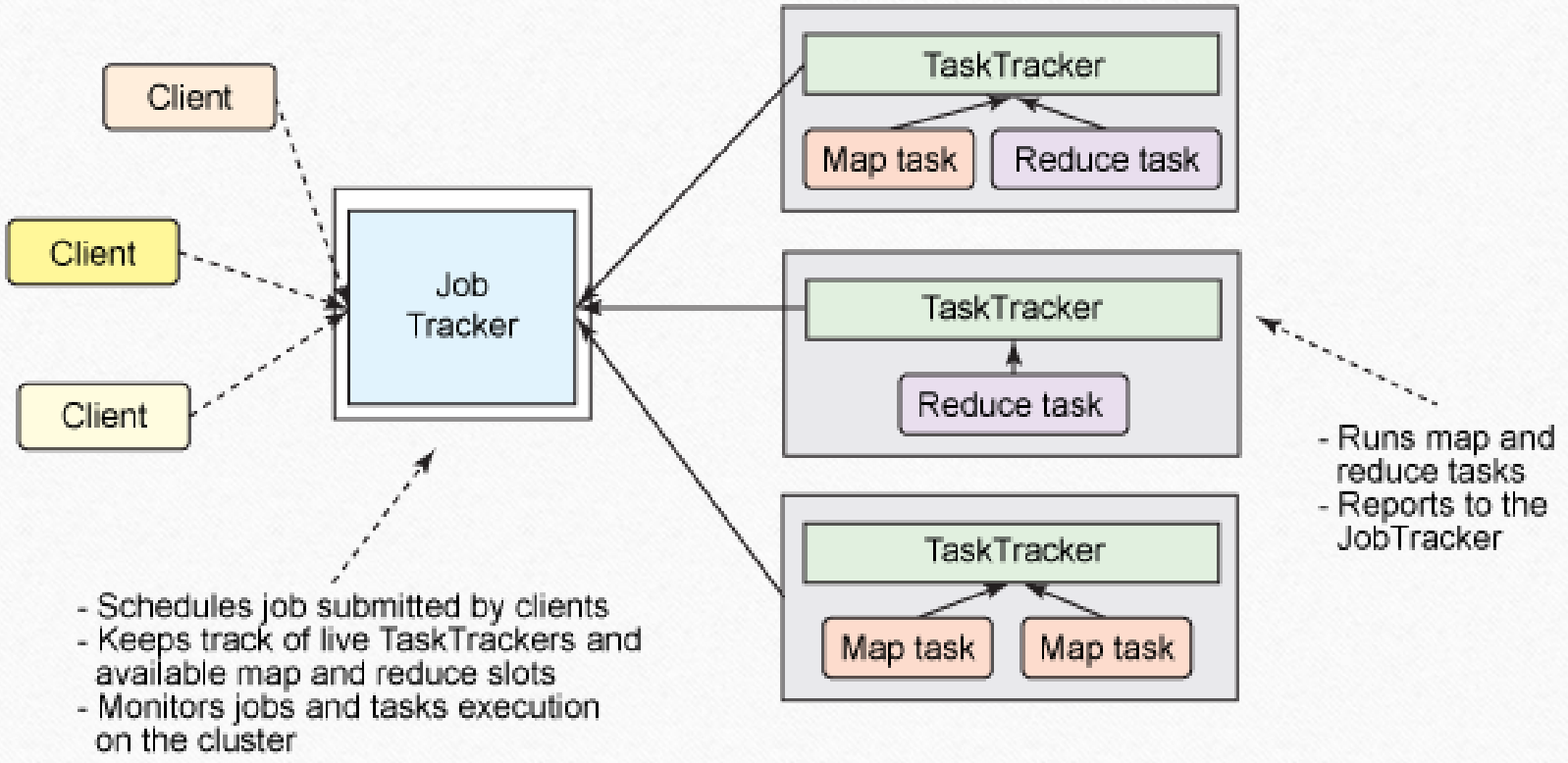
Kawa, Adam, "Introduction to Yarn", IBM, August 12, 2014.

https://www.ibm.com/developerworks/library/bd-yarn-intro/

Apache Hadoop

- The Hadoop infrastructure takes care of all complex aspects of distributed processing: parallelization, scheduling, resource management, inter-machine communication, handling software and hardware failures, and more.
- Thanks to this clean abstraction, implementing distributed applications that process terabytes of data on hundreds (or even thousands) of machines has never been so easy — even for developers with no previous experience with distributed systems.

“Classic” Hadoop



Job Execution In “Classic” Hadoop

- In the MapReduce framework, the job execution is controlled by two types of processes:
  - A single master process called *JobTracker,* which coordinates all jobs running on the cluster and assigns map and reduce tasks to run on the TaskTrackers
  - A number of subordinate processes called *TaskTrackers,* which run assigned tasks and periodically report the progress to the JobTracker

The JobTracker's Roles In "Classic Hadoop"

- Management of computational resources in the cluster, which involves maintaining the list of live nodes, the list of available and occupied map and reduce slots, and allocating the available slots to appropriate jobs and tasks according to selected scheduling policy
- Coordination of all tasks running on a cluster, which involves instructing TaskTrackers to start map and reduce tasks, monitoring the execution of the tasks, restarting failed tasks, speculatively running slow tasks, calculating total values of job counters, and more

Limitations of Classic Hadoop

- The most serious limitations of this model are primarily related to scalability, resource utilization, and the support of workloads different from MapReduce.

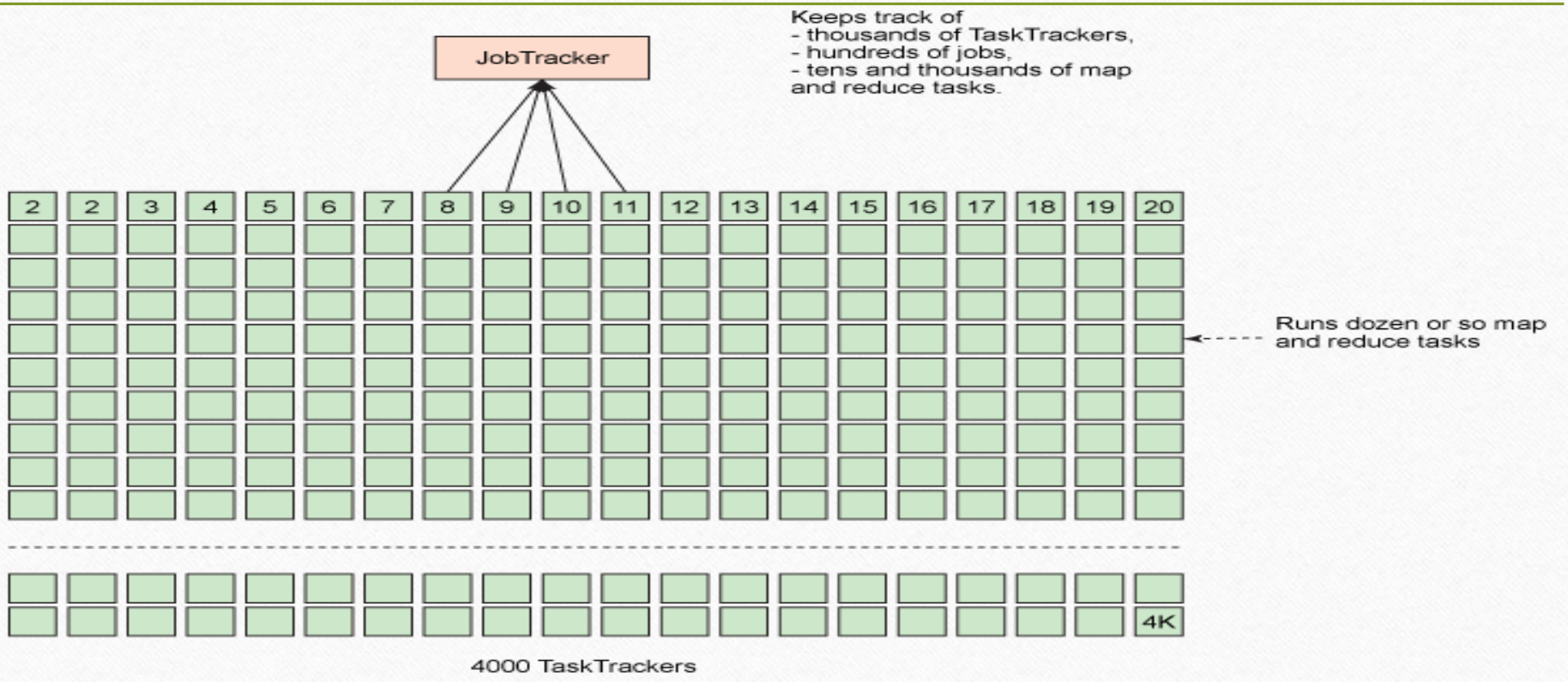
Scalability Limits of Classic Hadoop

- The large Hadoop clusters revealed a limitation involving a scalability bottleneck caused by having a single JobTracker. According to Yahoo!, the practical limits of such a design are reached with a cluster of 5,000 nodes and 40,000 tasks running concurrently. Due to this limitation, smaller and less-powerful clusters had to be created and maintained.

Why The Scalability Issue

- The large number of responsibilities given to a single process caused significant scalability issues, especially on a larger cluster where the JobTracker had to constantly keep track of thousands of TaskTrackers, hundreds of jobs, and tens of thousands of map and reduce tasks. The image below illustrates the issue.
  - On the contrary, the TaskTrackers usually run only a dozen or so tasks, which were assigned to them by the hard-working JobTracker.

A Picture Worth For Visual Reinforcement



Partitioning Problem

- Neither smaller nor larger Classic Hadoop clusters used their computational resources with optimum efficiency.
- In Hadoop MapReduce, the computational resources on each slave node are divided by a cluster administrator into a fixed number of map and reduce slots, which are not fungible.
- With the number of map and reduce slots set, a node cannot run more map tasks than map slots at any given moment, even if no reduce tasks are running.
- It harms the cluster utilization because when all map slots are taken (and we still want more), we cannot use any reduce slots, even if they are available, or vice versa.

Why Only One Trick?

- Classic Hadoop was designed to run MapReduce jobs only.
- With the advent of alternative programming models (such as graph processing provided by Apache Giraph), there was an increasing need to support programming paradigms besides
- MapReduce that could run on the same cluster and share resources in an efficient and fair manner.

Goals

- Make Hadoop scalable
- Make it more efficient
- Let the infrastructure do other things

A Simple, Powerful Idea

- To address the scalability issue, a simple but brilliant idea was proposed:
  - Let's somehow reduce the responsibilities of the single JobTracker and delegate some of them to the TaskTrackers since there are many of them in a cluster.
- This concept was reflected in a new design by separating dual responsibilities of the JobTracker (cluster resource management and task coordination) into two distinct types of processes.

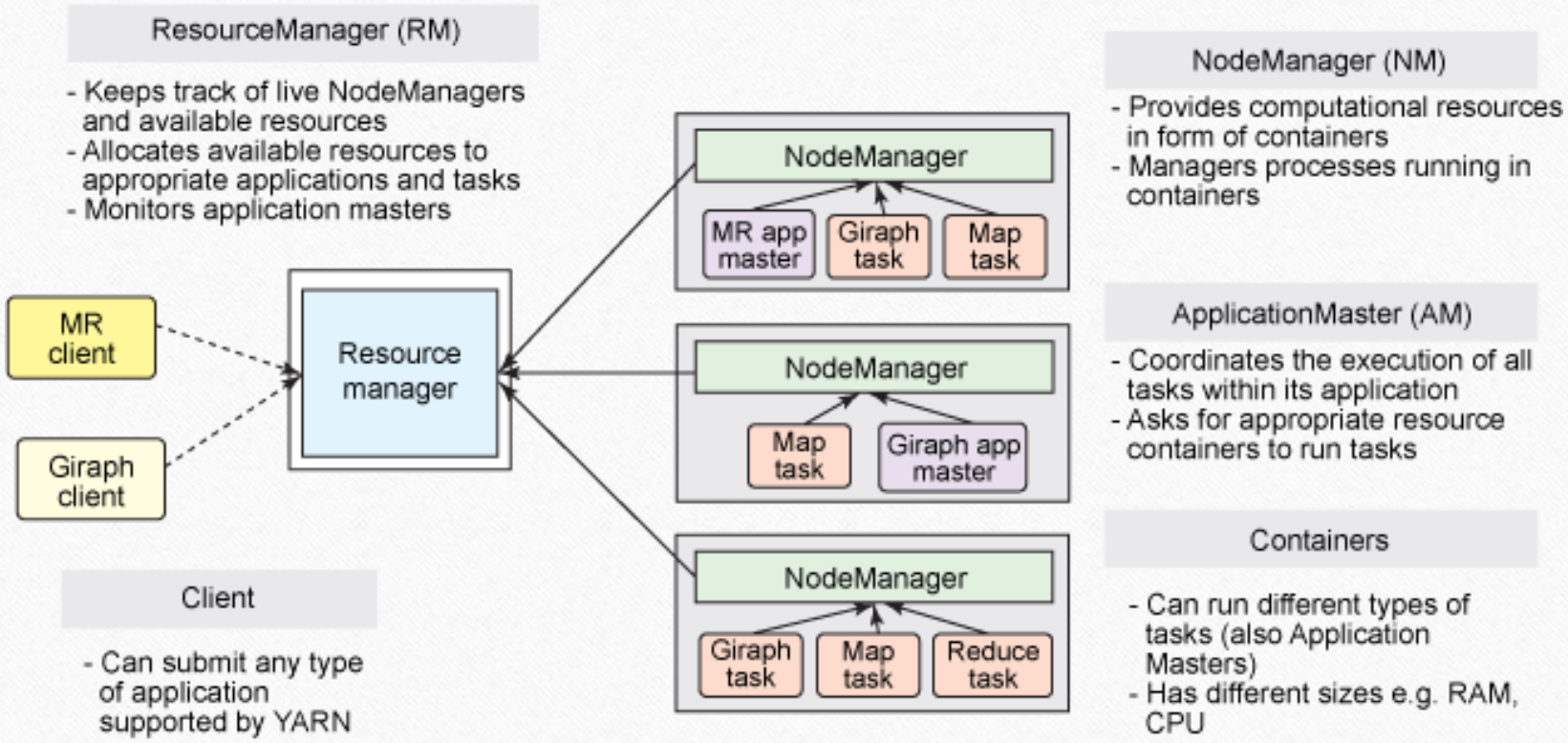
Distributing the JobTracker

- Instead of having a single JobTracker, a new approach introduces a cluster manager with the sole responsibility of tracking live nodes and available resources in the cluster and assigning them to the tasks.
- For each job submitted to a cluster, a dedicated and short-living JobTracker is started to control the execution of tasks within that job only.
  - Interestingly, the short-living JobTrackers are started by the TaskTrackers running on slave nodes.
  - Thus, the coordination of a job's life cycle is spread across all of the available machines in the cluster.
  - Thanks to this behavior, more jobs can run in parallel and scalability is dramatically increased.

New, YARN Terminology

- The following name changes give a bit of insight into the design of YARN:
  - ResourceManager instead of a cluster manager
  - ApplicationMaster instead of a dedicated and short-lived JobTracker
  - NodeManager instead of TaskTracker
  - A distributed application instead of a MapReduce job
- What do these new names suggest about how we view the system?

YARN Architecture



The *ResourceManager*

- A global *ResourceManager* runs as a master daemon, usually on a dedicated machine, that arbitrates the available cluster resources among various competing applications.
- The *ResourceManager* tracks how many live nodes and resources are available on the cluster and coordinates what applications submitted by users should get these resources and when.
- The *ResourceManager* is the single process that has this information so it can make its allocation (or rather, scheduling) decisions in a shared, secure, and multi-tenant manner (for instance, according to an application priority, a queue capacity, ACLs, data locality, etc.).

The *ResourceManager*

- Although the *ResourceManager* does not perform any monitoring of the tasks within an application, it checks the health of the *ApplicationMasters*.
  - If the ApplicationMaster fails, it can be restarted by the *ResourceManager* in a new container.

The *ApplicationMaster*

- When a user submits an application, an instance of a lightweight process called the *ApplicationMaster* is started to coordinate the execution of all tasks within the application.
  - This includes monitoring tasks, restarting failed tasks, speculatively running slow tasks, and calculating total values of application counters.
  - These responsibilities were previously assigned to the single *JobTracker* for all jobs.
- The *ApplicationMaster* and tasks that belong to its application run in resource containers controlled by the *NodeManagers*.

The *ApplicationMaster*

- The *ApplicationMaster* spends its whole life negotiating containers to launch all of the tasks needed to complete its application.
- It also monitors the progress of an application and its tasks, restarts failed tasks in newly requested containers, and reports progress back to the client that submitted the application.
- After the application is complete, the *ApplicationMaster* shuts itself down and releases its own container.

Responsibilities In One Sentence

- One can say that the *ResourceManager* takes care of the *ApplicationMasters*, while the *ApplicationMasters* takes care of tasks.

The *NodeManager*

- The *NodeManager* is a more generic and efficient version of the *TaskTracker*.
- Instead of having a fixed number of map and reduce slots, the *NodeManager* has a number of dynamically created resource containers.
  - The size of a container depends upon the amount of resources it contains, such as memory, CPU, disk, and network IO.
  - The number of containers on a node is a product of configuration parameters and the total amount of node resources (such as total CPUs and total memory) outside the resources dedicated to the slave daemons and the OS.

Newfound Flexibility

- The *ResourceManager*, the *NodeManager*, and a container are not concerned about the type of application or task. All application framework-specific code is simply moved to its *ApplicationMaster* so that any distributed framework can be supported by YARN — as long as someone implements an appropriate *ApplicationMaster* for it.
- Thanks to this generic approach, the dream of a Hadoop YARN cluster running many various workloads comes true. Imagine: a single Hadoop cluster in your data center that can run *MapReduce, Giraph, Storm, Spark, Tez/Impala, MPI*, and more.

Newfound Flexibility

- The *ApplicationMaster* can run any type of task inside a container.
  - For example, the MapReduce *ApplicationMaster* requests a container to launch a map or a reduce task,
  - The Giraph *ApplicationMaster* requests a container to run a Giraph task.
  - You can also implement a custom *ApplicationMaster* that runs specific tasks and, in this way, invent a shiny new distributed application framework that changes the big data world.

What About MapReduce?

- In YARN, MapReduce is simply a distributed application (but still a very popular and useful one)
- …introducing….MRv2.
- MRv2 is simply the re-implementation of the classical MapReduce engine, now called MRv1, that runs on top of YARN.

Additional Benefits?

- The single-cluster approach obviously provides a number of advantages, including:
  - Higher cluster utilization, whereby resources not used by one framework could be consumed by another
  - Lower operational costs, because only one "do-it-all" cluster needs to be managed and tuned
  - Reduced data motion, as there's no need to move data between Hadoop YARN and systems running on different clusters of machines
  - Managing a single cluster also results in a greener solution to data processing. Less data center space is used, less silicon wasted, less power used, and less carbon emitted simply because we run the same calculation on a smaller but more efficient Hadoop cluster

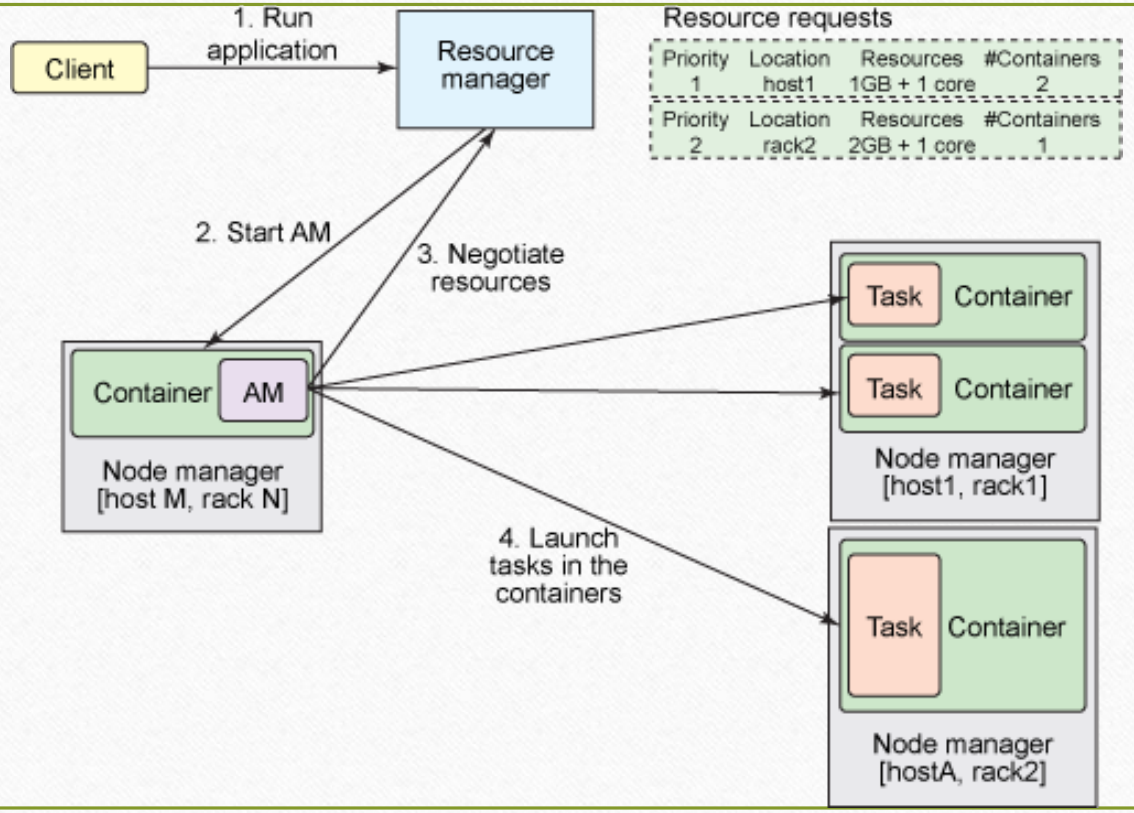
Additional Improvement: Uberization

- *Uberization* is the possibility to run all tasks of a MapReduce job in the ApplicationMaster's JVM if the job is small enough. This way, you avoid the overhead of requesting containers from the ResourceManager and asking the NodeManagers to start (supposedly small) tasks.

Additional Improvement: Better Recovery

- An application recovery after the restart of *ResourceManager* (YARN-128).
- The *ResourceManager* stores information about running applications and completed tasks in HDFS.
- If the *ResourceManager* is restarted, it recreates the state of applications and re-runs only incomplete tasks.

Application Submission



Submitting an Application

- Suppose that users submit applications to the *ResourceManager* by typing the hadoop jar command in the same manner as in MRv1.
  - The *ResourceManager* maintains the list of applications running on the cluster and the list of available resources on each live *NodeManager*.
  - The *ResourceManager* needs to determine which application should get a portion of cluster resources next.
    - The decision is subjected to many constraints, such as queue capacity, ACLs, and fairness. The *ResourceManager* uses a pluggable Scheduler.
  - The Scheduler focuses only on scheduling; it manages who gets cluster resources (in the form of containers) and when, but it does not perform any monitoring of the tasks within an application so it does not attempt to restart failed tasks

Accepting a New Application Submission

- When the *ResourceManager* accepts a new application submission, one of the first decisions the *Scheduler* makes is selecting a container in which *ApplicationMaster* will run.
- After the *ApplicationMaster* is started, it will be responsible for a whole life cycle of this application.
- First and foremost, it will be sending resource requests to the *ResourceManager* to ask for containers needed to run an application's tasks.

Resource Request

- A resource request is simply a request for a number of containers that satisfies some resource requirements, such as:
  - An amount of resources, today expressed as megabytes of memory and CPU shares
  - A preferred location, specified by hostname, rackname, or * to indicate no preference
  - A priority within this application, and not across multiple applications

Granting a Container

- If and when it is possible, the *ResourceManager* grants a container (expressed as container ID and hostname) that satisfies the requirements requested by the *ApplicationMaster* in the resource request.
  - A container allows an application to use a given amount of resources on a specific host.
- After a container is granted, the *ApplicationMaster* will ask the NodeManager (that manages the host on which the container was allocated) to use these resources to launch an application-specific task.
  - This task can be any process written in any framework (such as a MapReduce task or a Giraph task).
  - The NodeManager does not monitor tasks; it only monitors the resource usage in the containers and, for example, it kills a container if it consumes more memory than initially allocated.

Summary

- YARN offers clear advantages in scalability, efficiency, and flexibility compared to the classical MapReduce engine in the first version of Hadoop.
- Both small and large Hadoop clusters greatly benefit from YARN.
- To the end user (a developer, not an administrator), the changes between MRv1 and MRv2 are almost invisible because it's possible to run unmodified MapReduce jobs using the same MapReduce API and CLI.