
# YARN Architecture

# Contents

# MR Hashing Algorithm


The,1
The,1
Brown,1

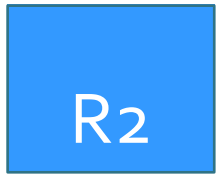

Fox,1
The,1
Brown,1


The,1
Dog,1
Brown,1

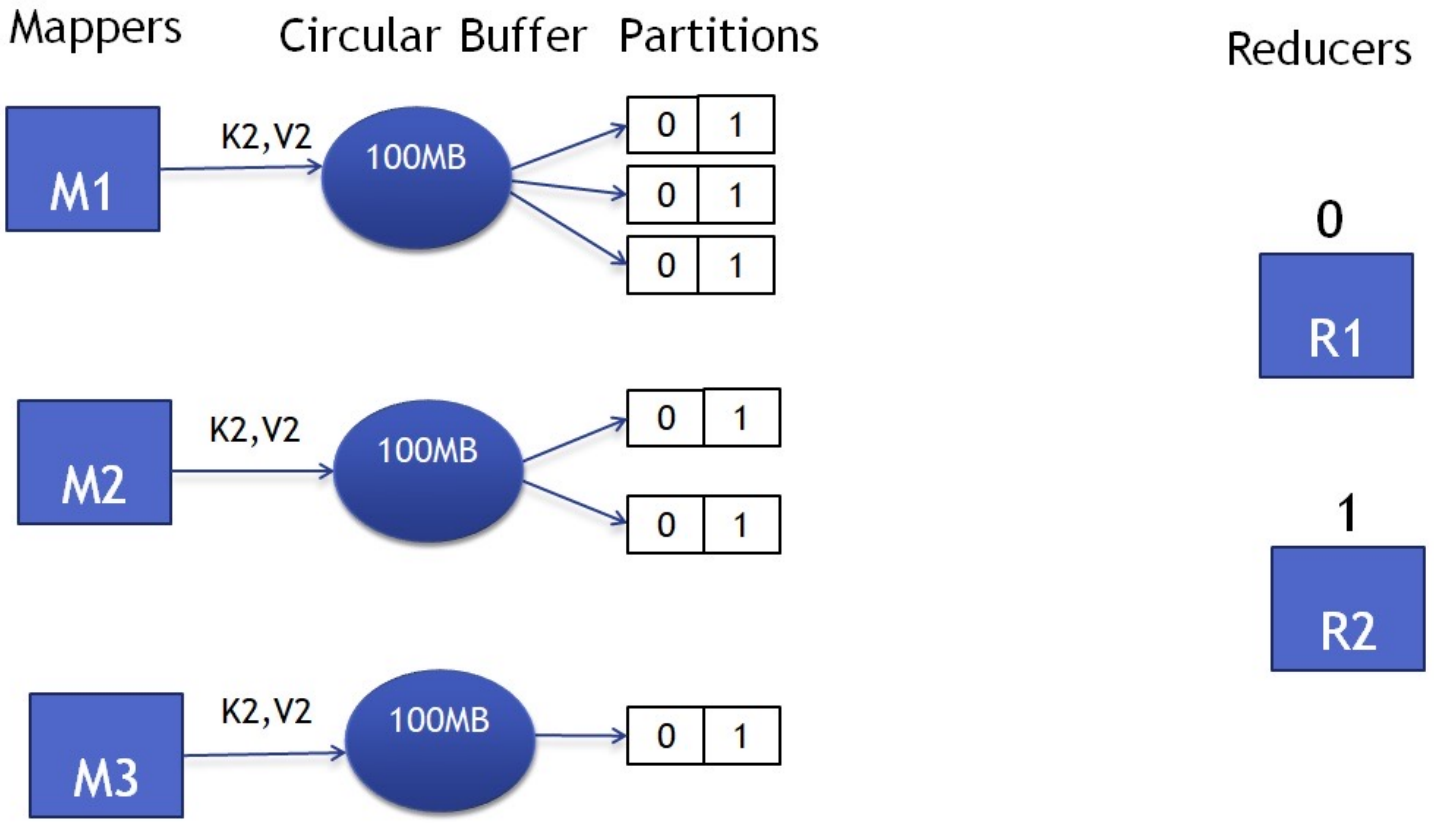
"The".hashcode() % No Of Reducers

0

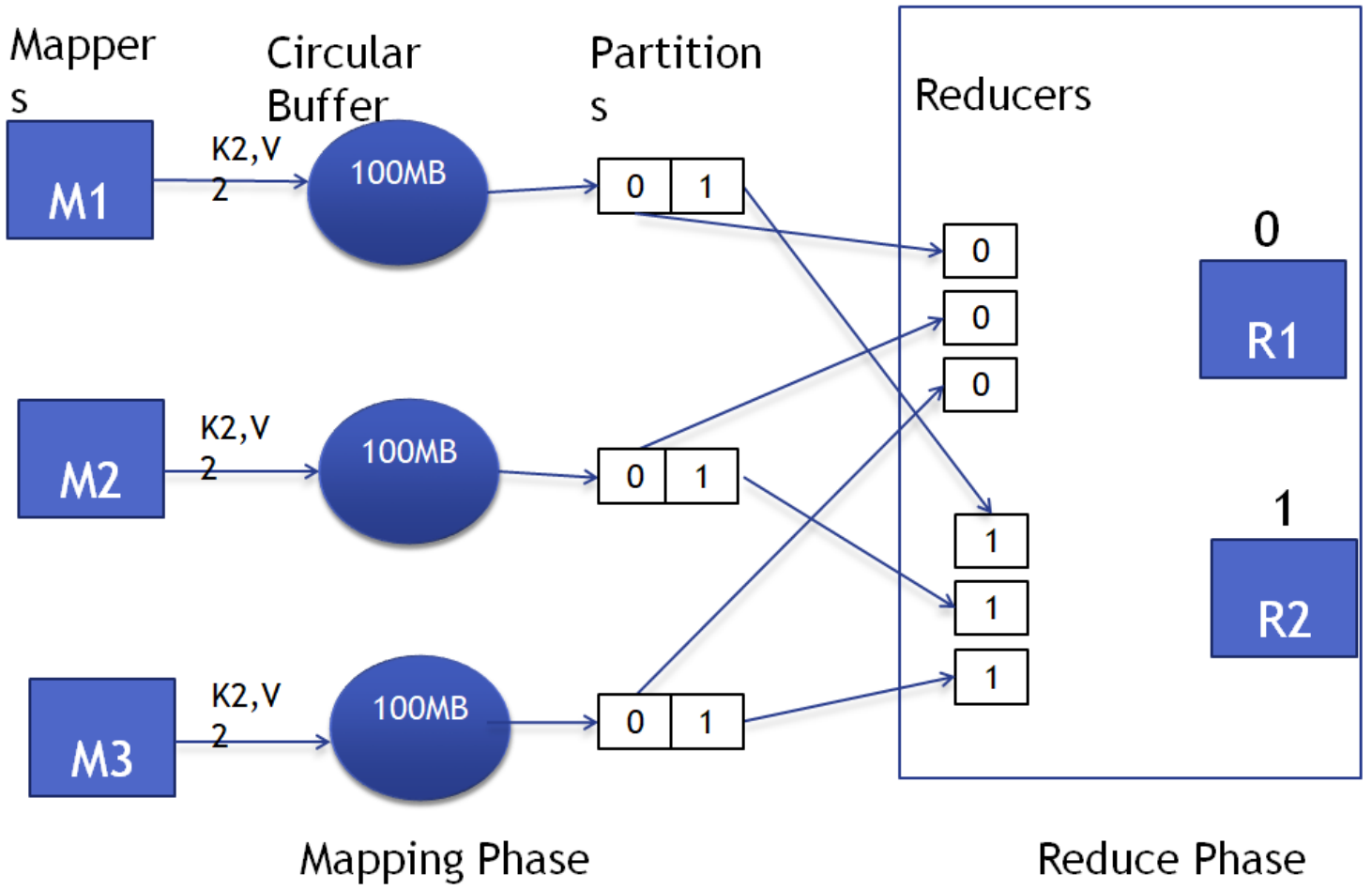

1



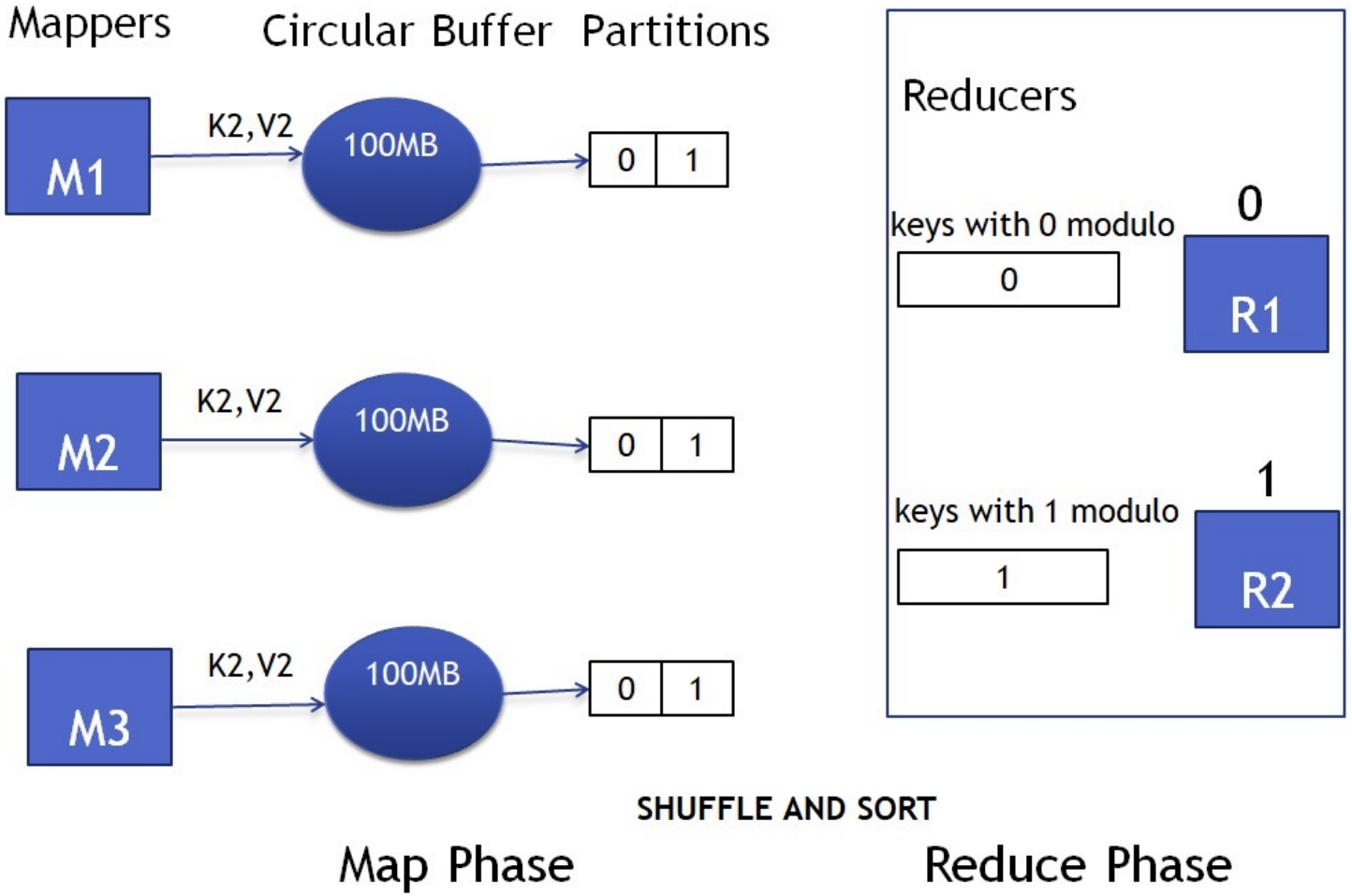
# Shuffle and Sort



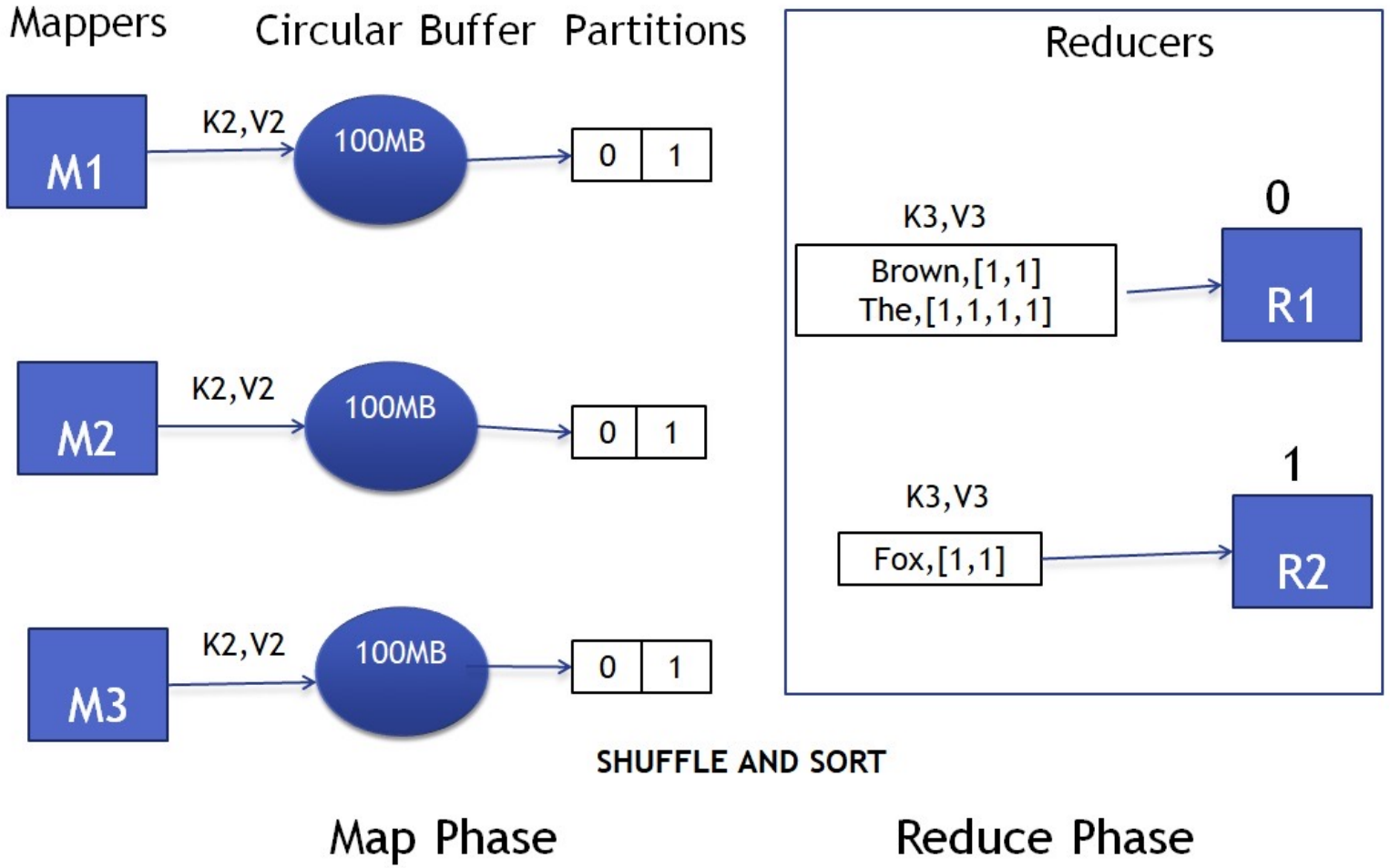
# Shuffle and Sort



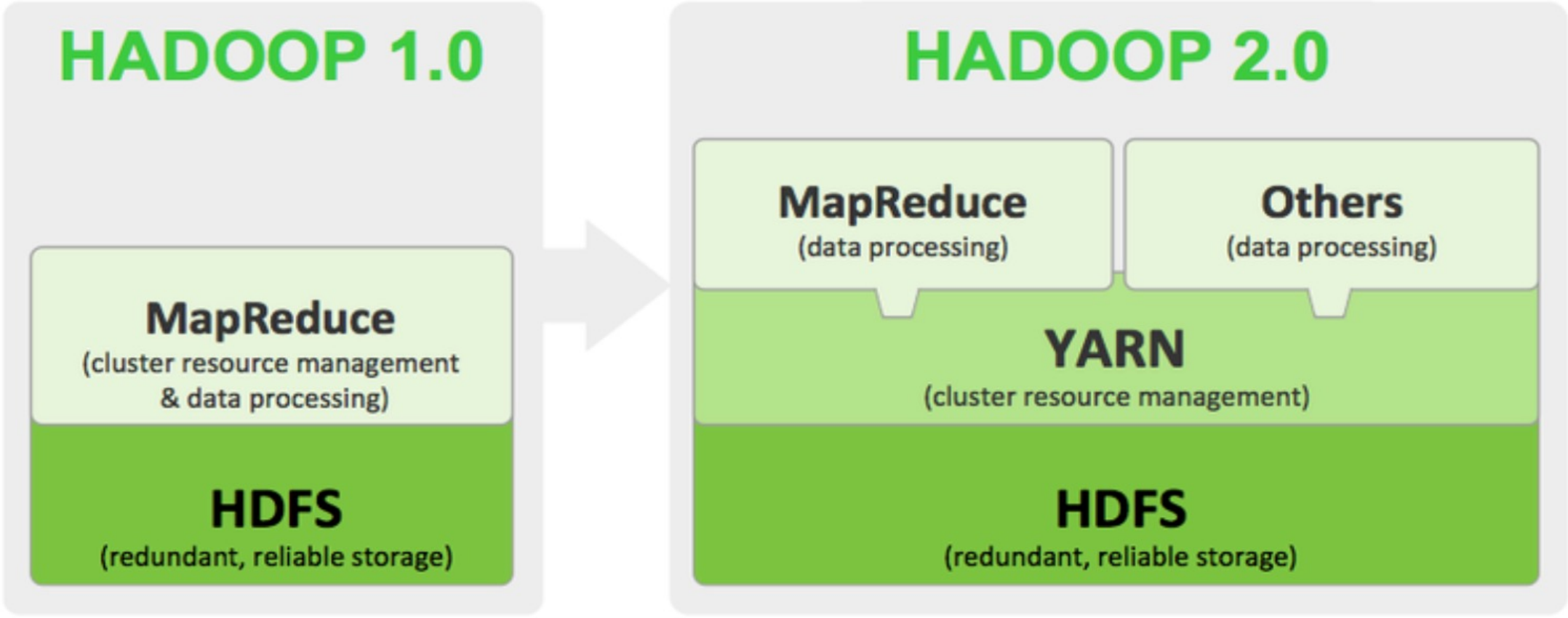
# Shuffle and Sort



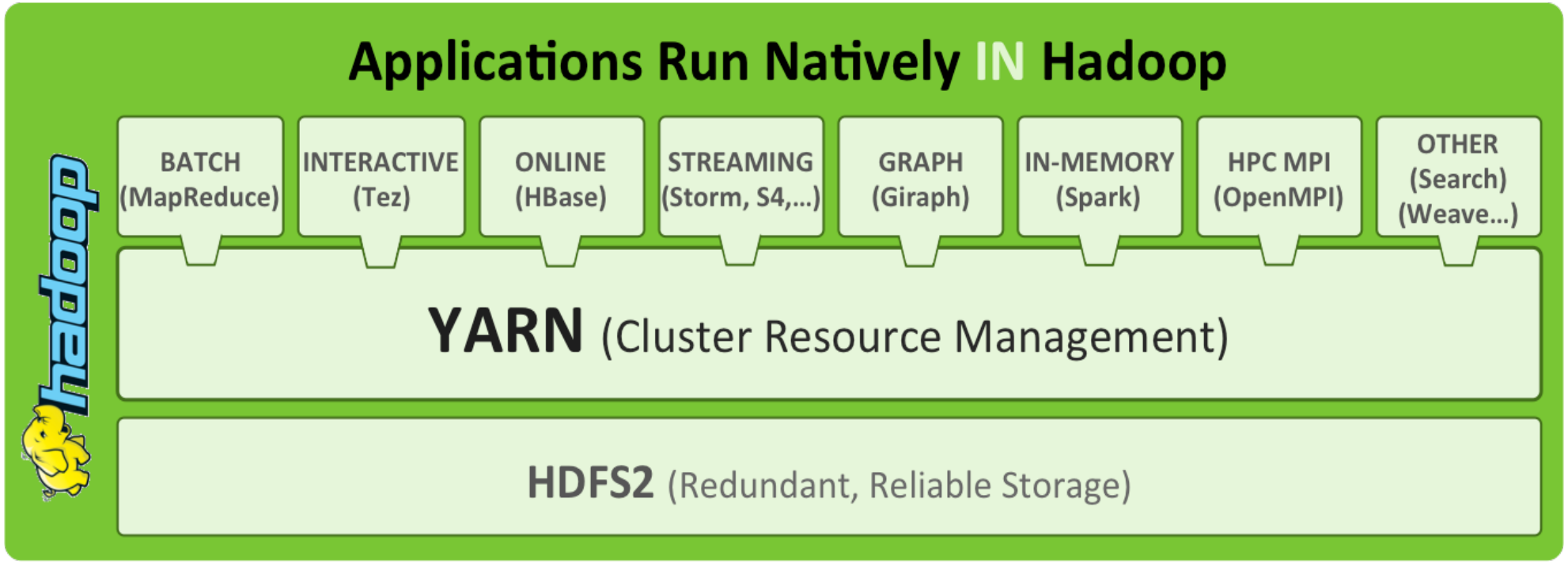
# Shuffle and Sort



# Hadoop Versions



# Hadoop YARN


Applications Run Natively IN Hadoop
BATCH (MapReduce)
INTERACTIVE (Tez)
ONLINE (HBase)
STREAMING (Storm, S4,…)
GRAPH (Giraph)
IN-MEMORY (Spark)
HPC MPI (OpenMPI)
OTHER (Search) (Weave…)
hadoop
YARN (Cluster Resource Management)
HDFS2 (Redundant, Reliable Storage)

# Anatomy of a MapReduce Job

- You can run a MapReduce job with a single method call: submit() on a Job object (you can also call waitForCompletion(), which submits the job if it hasn't been submitted already, then waits for it to finish).
- At the highest level, there are five independent entities:

  - The client, which submits the MapReduce job.
  - The YARN resource manager, which coordinates the allocation of compute resources on the cluster.
  - The YARN node managers which launch and monitor the compute containers on machines in the cluster.
  - The MapReduce application master, which coordinates the tasks running the MapReduce job. The application master and the MapReduce tasks run in containers that are scheduled by the resource manager and managed by the node managers.
  - The distributed file system which is used for sharing job files between the other entities.

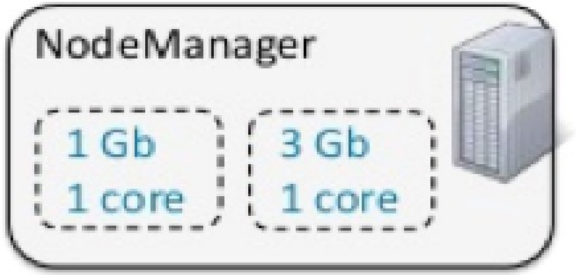
# YARN Daemons

- **Resource Manager (RM)**
  - Runs on master node
  - Global resource scheduler
  - Arbitrates system resources between competing applications


- **Node Manager (NM)**
  - Runs on slave nodes
  - Communicates with RM

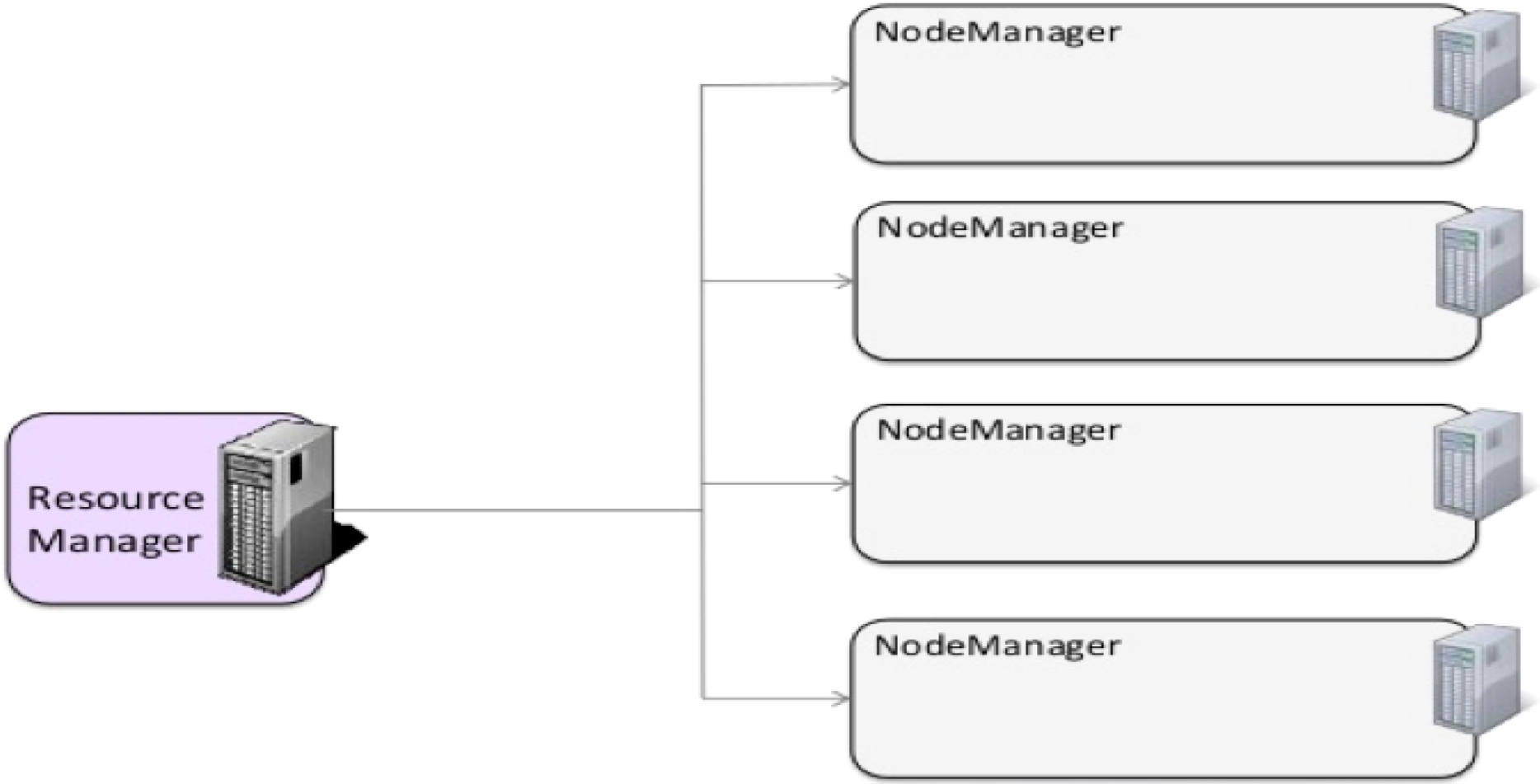
# YARN Components

- **Containers**
  - Created by the RM upon request
  - Allocate a certain amount of resources (memory, CPU) on a slave node
  - Applications run in one or more containers

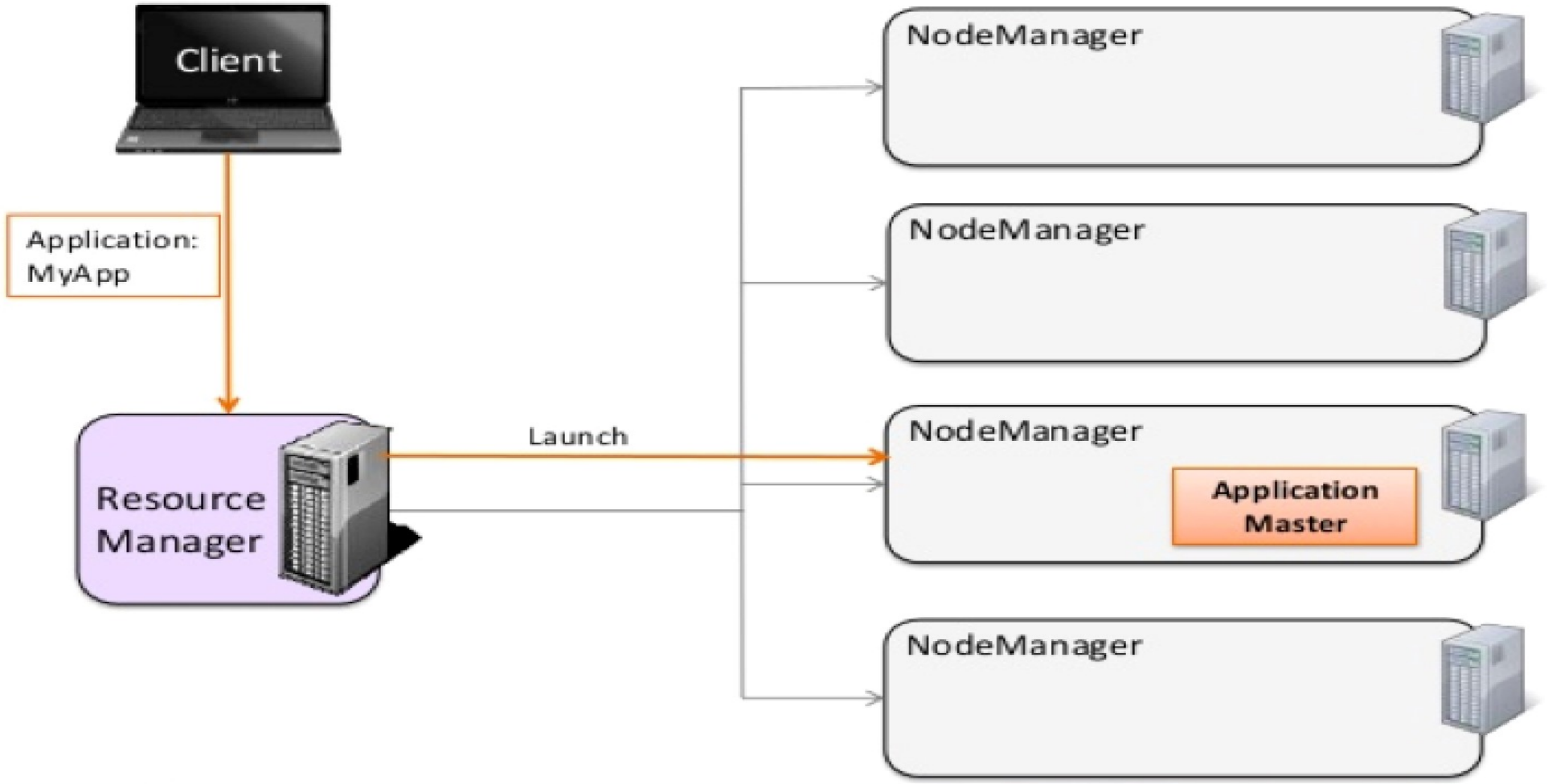



- **Application Master (AM)**
  - One per application
  - Framework/application specific
  - Runs in a container
  - Requests more containers to run application tasks

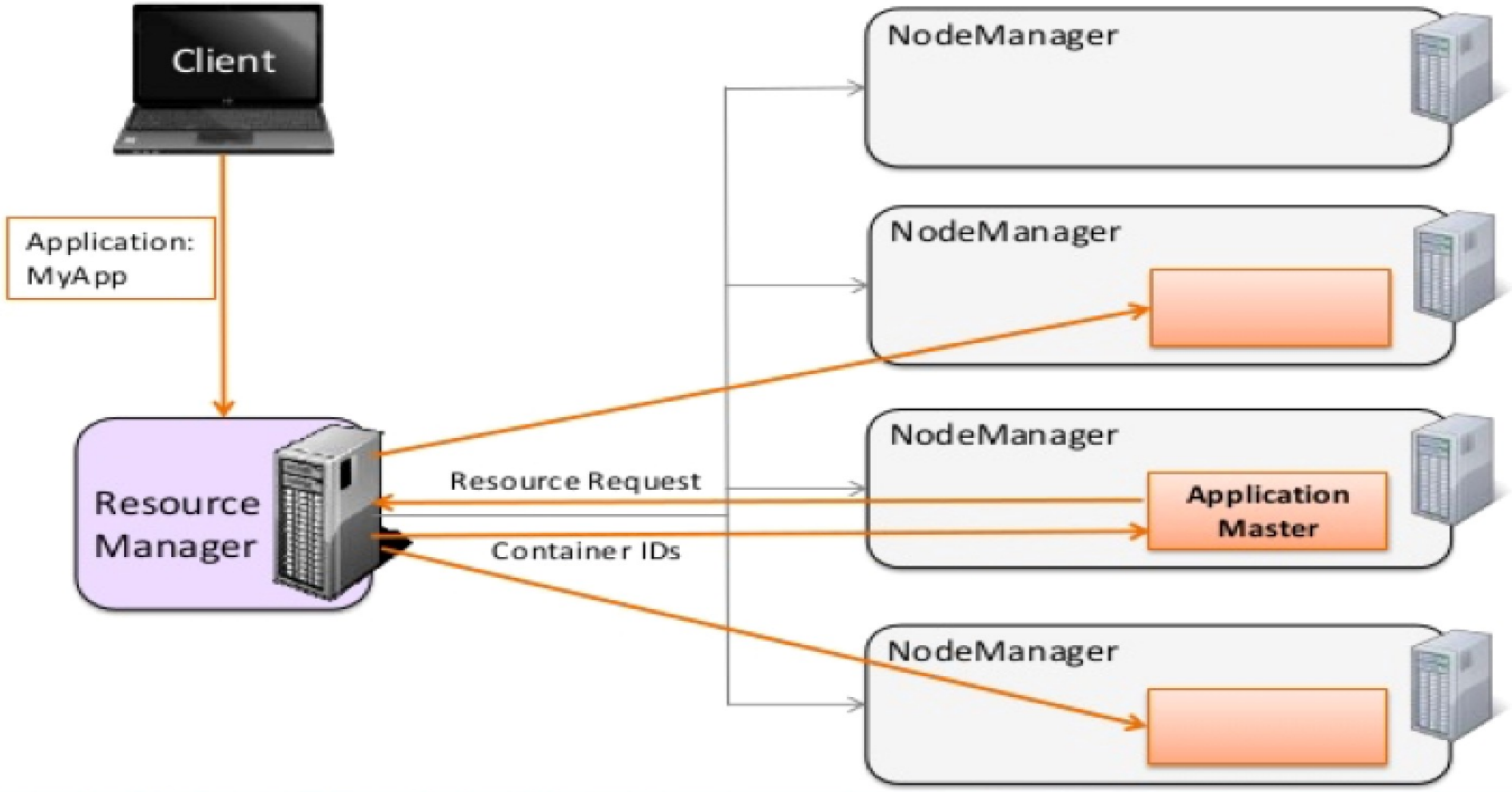
# Running an application in YARN



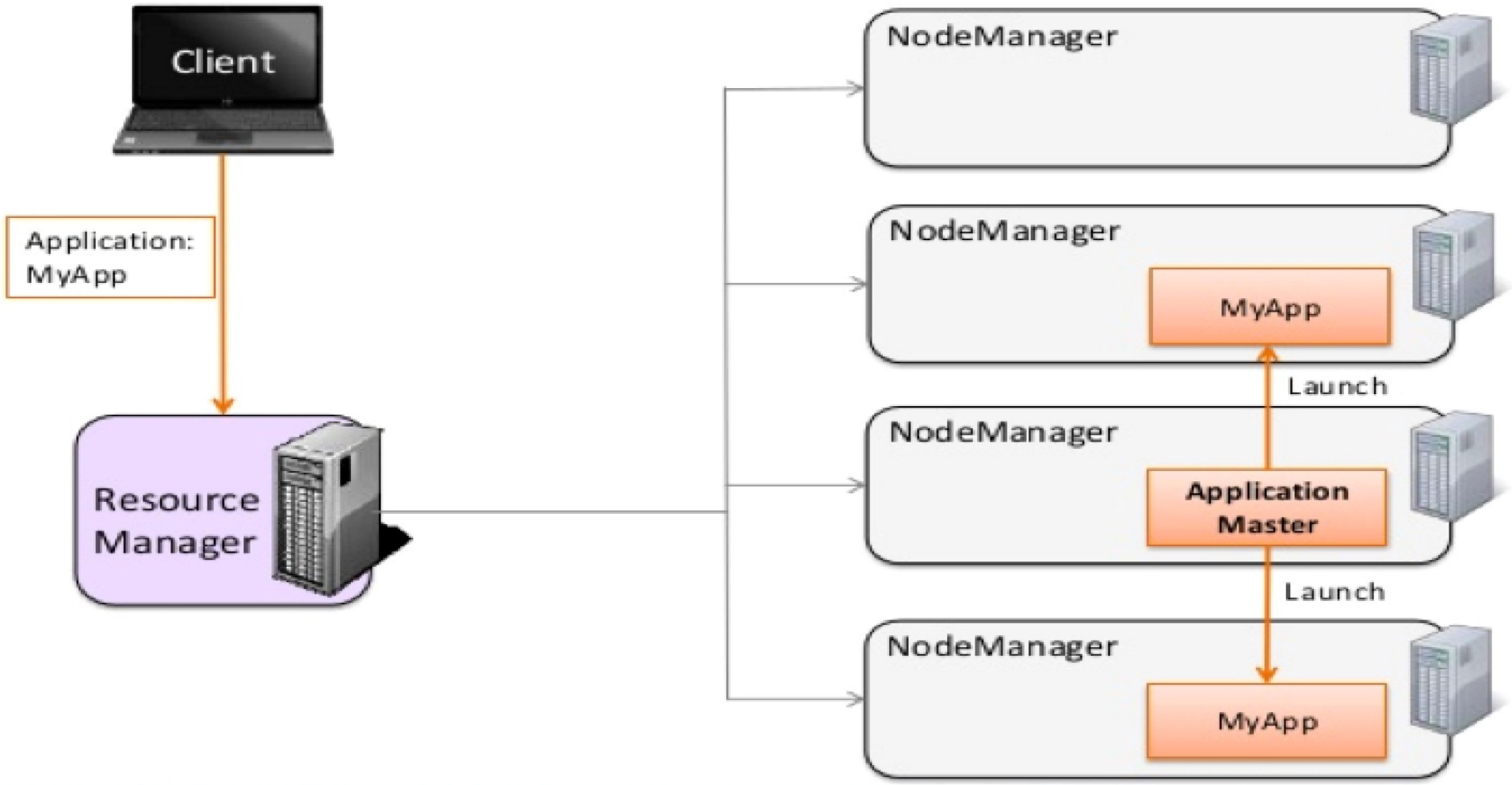
# Running an application in YARN



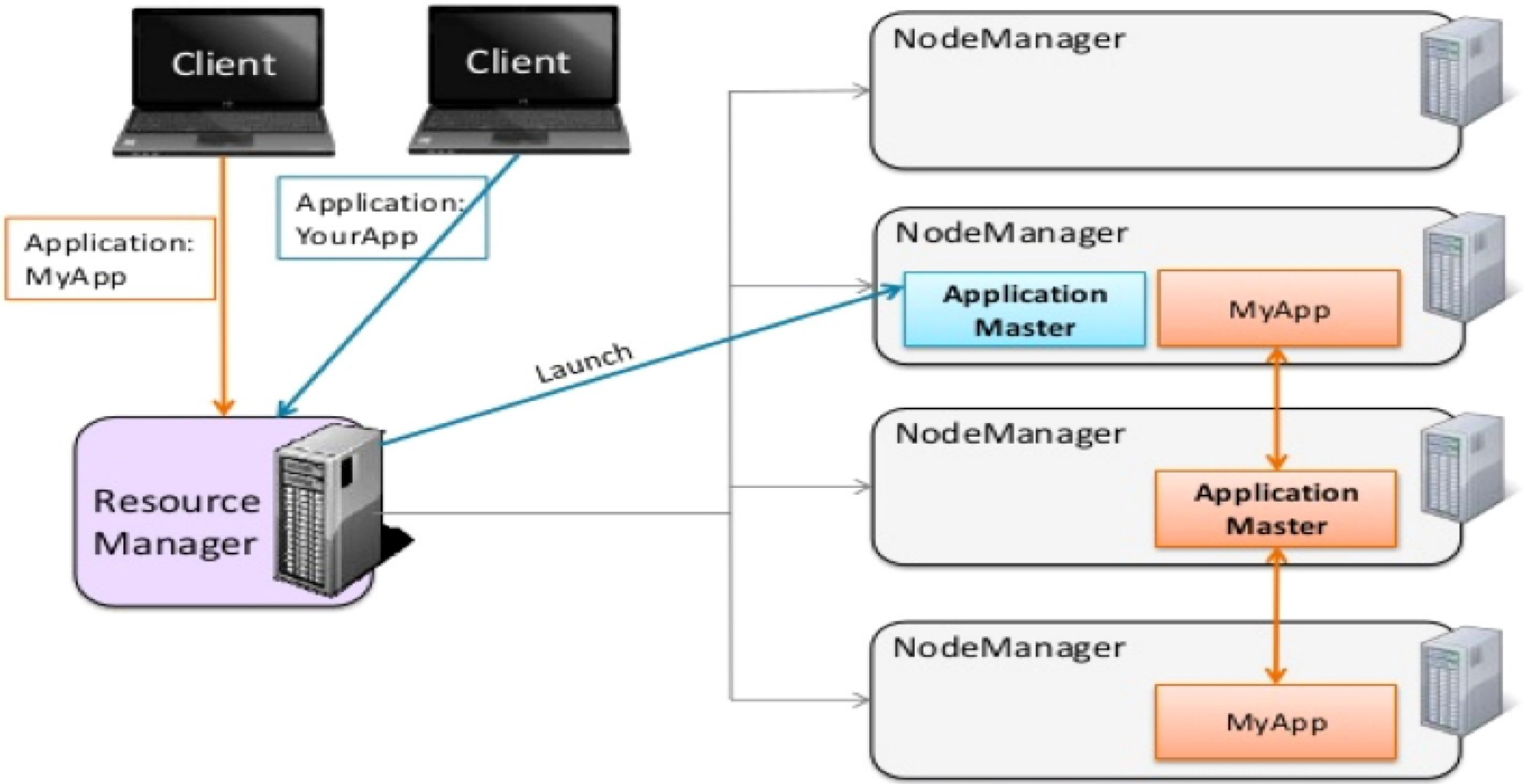
# Running an application in YARN



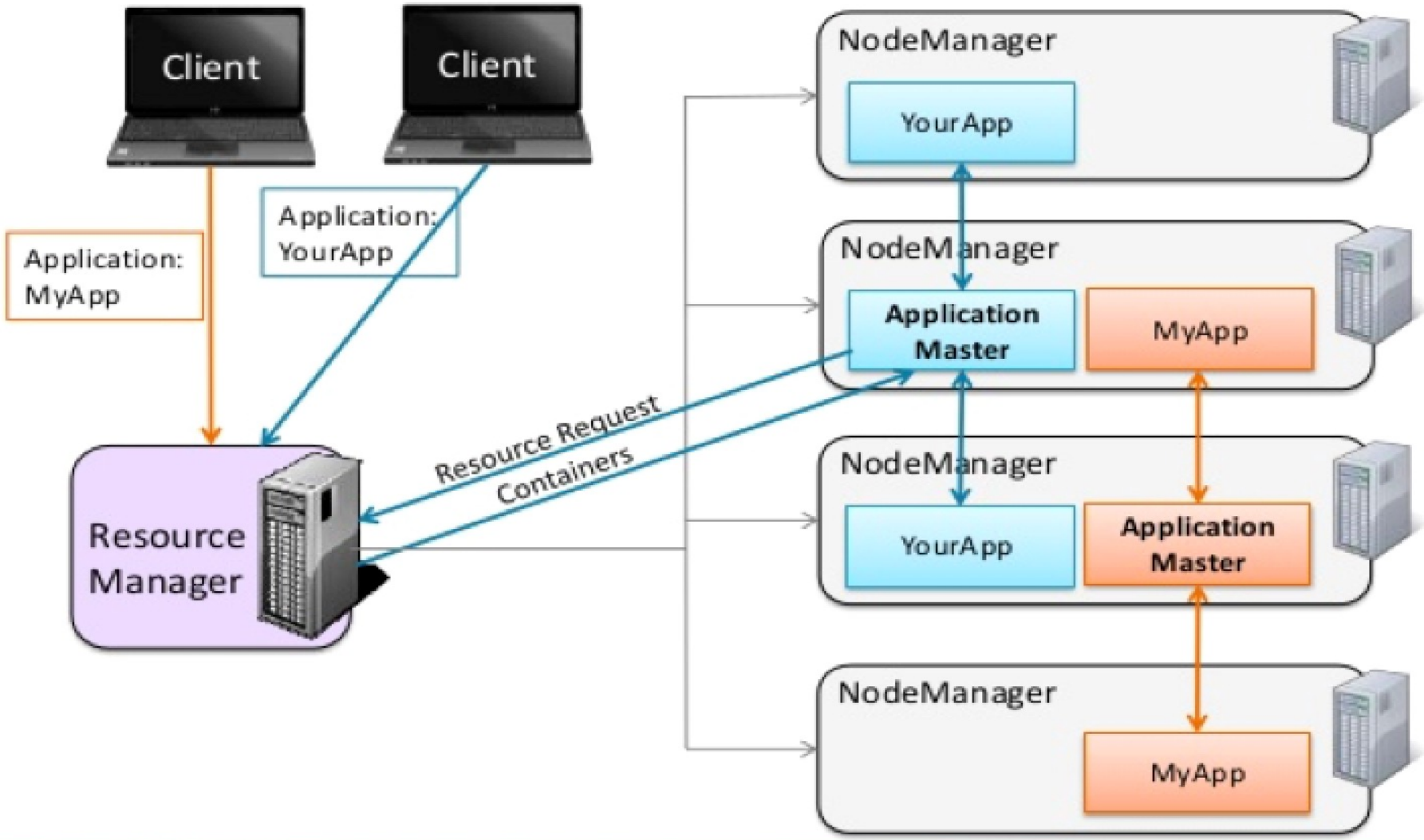
# Running an application in YARN



# Running an application in YARN



# Running an application in YARN



# Resource Manager

## Resource Manager Things to Know

- **What it does**
  - Manages nodes
    - Tracks heartbeats from NodeManagers
  - Manages containers
    - Handles AM requests for resources
    - De-allocates containers when they expire or the application completes
  - Manages ApplicationMasters
    - Creates a container for AMs and tracks heartbeats
  - Manages security
    - Supports Kerberos

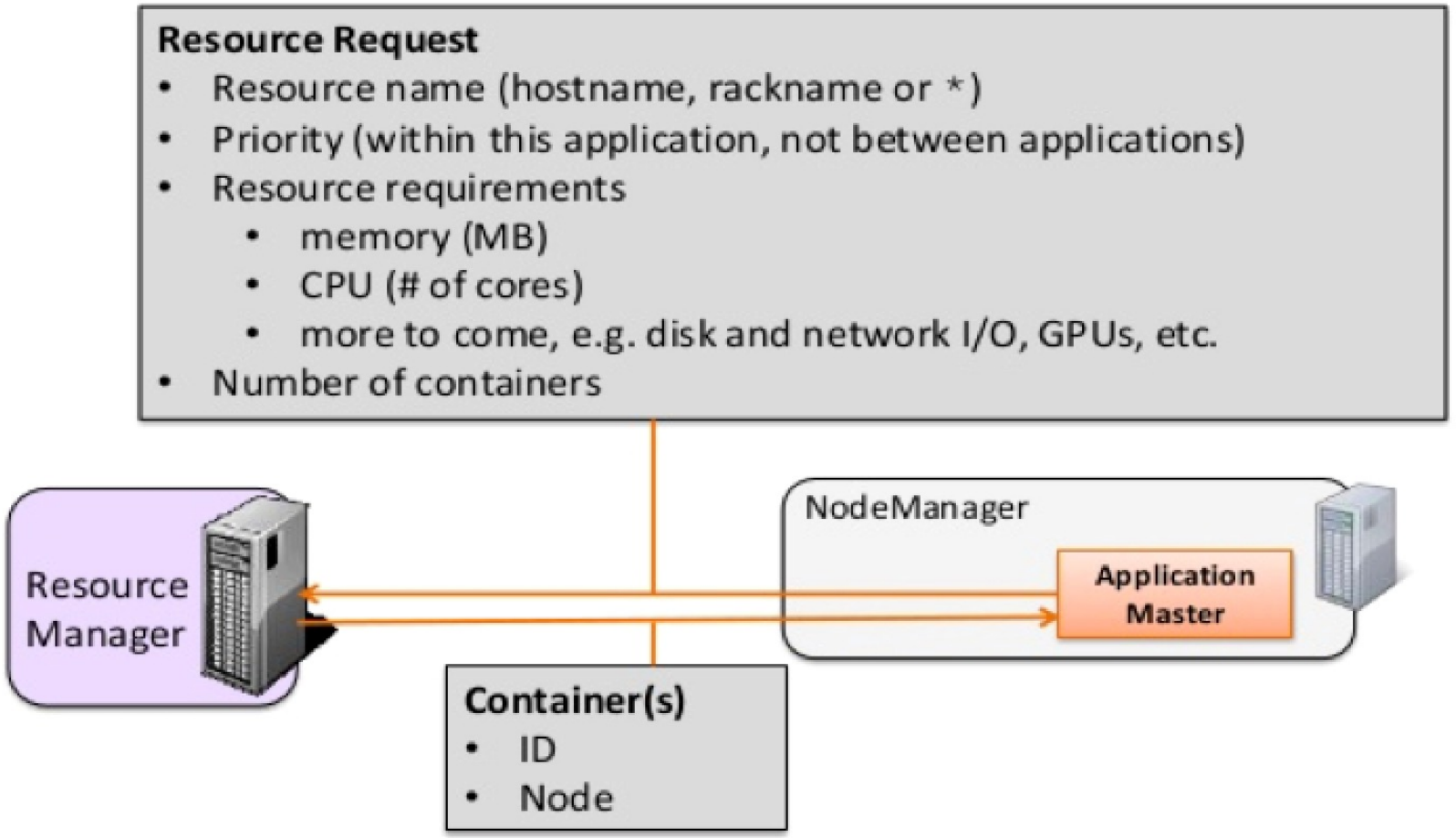
# Node Manager

## Node Manager Things to Know


- **What it does**
  - Communicates with the RM
    - Registers and provides info on node resources
    - Sends heartbeats and container status
  - Manages processes in containers
    - Launches AMs on request from the RM
    - Launches application processes on request from AM
    - Monitors resource usage by containers; kills run-away processes
  - Provides logging services to applications
    - Aggregates logs for an application and saves them to HDFS
  - Runs auxiliary services
  - Maintains node level security via ACLs

# Resource Requests

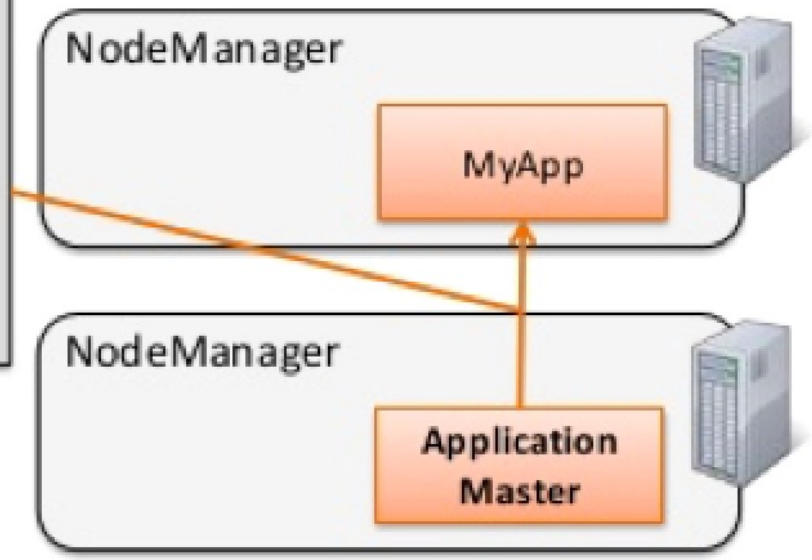
## Resource Requests



# Launching Application Process

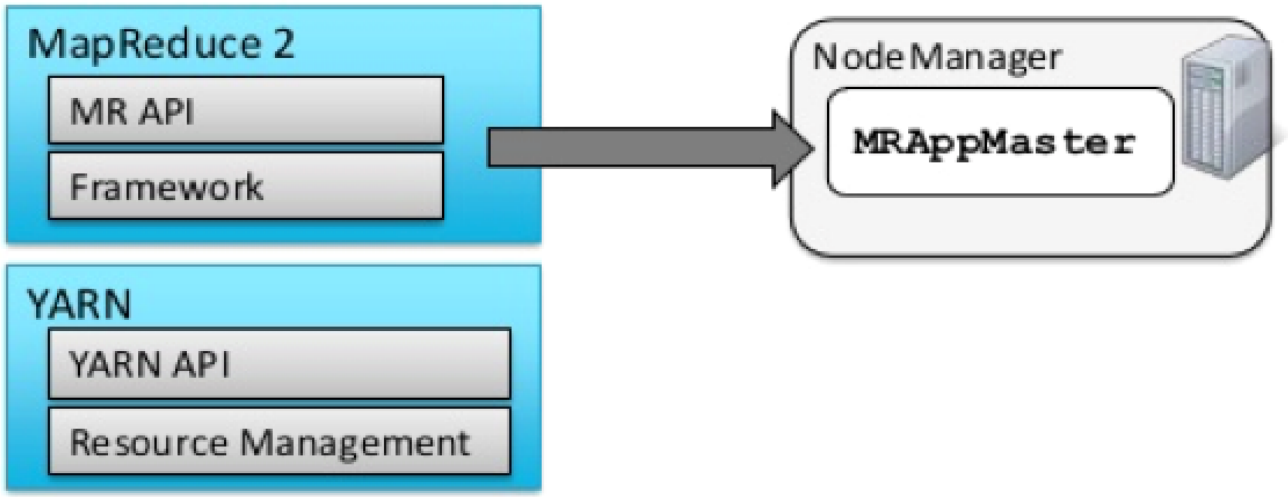
**Container Launch Context**

- Container ID
- Commands (to start application)
- Environment (configuration)
- Local Resources (e.g. application binary, HDFS files)

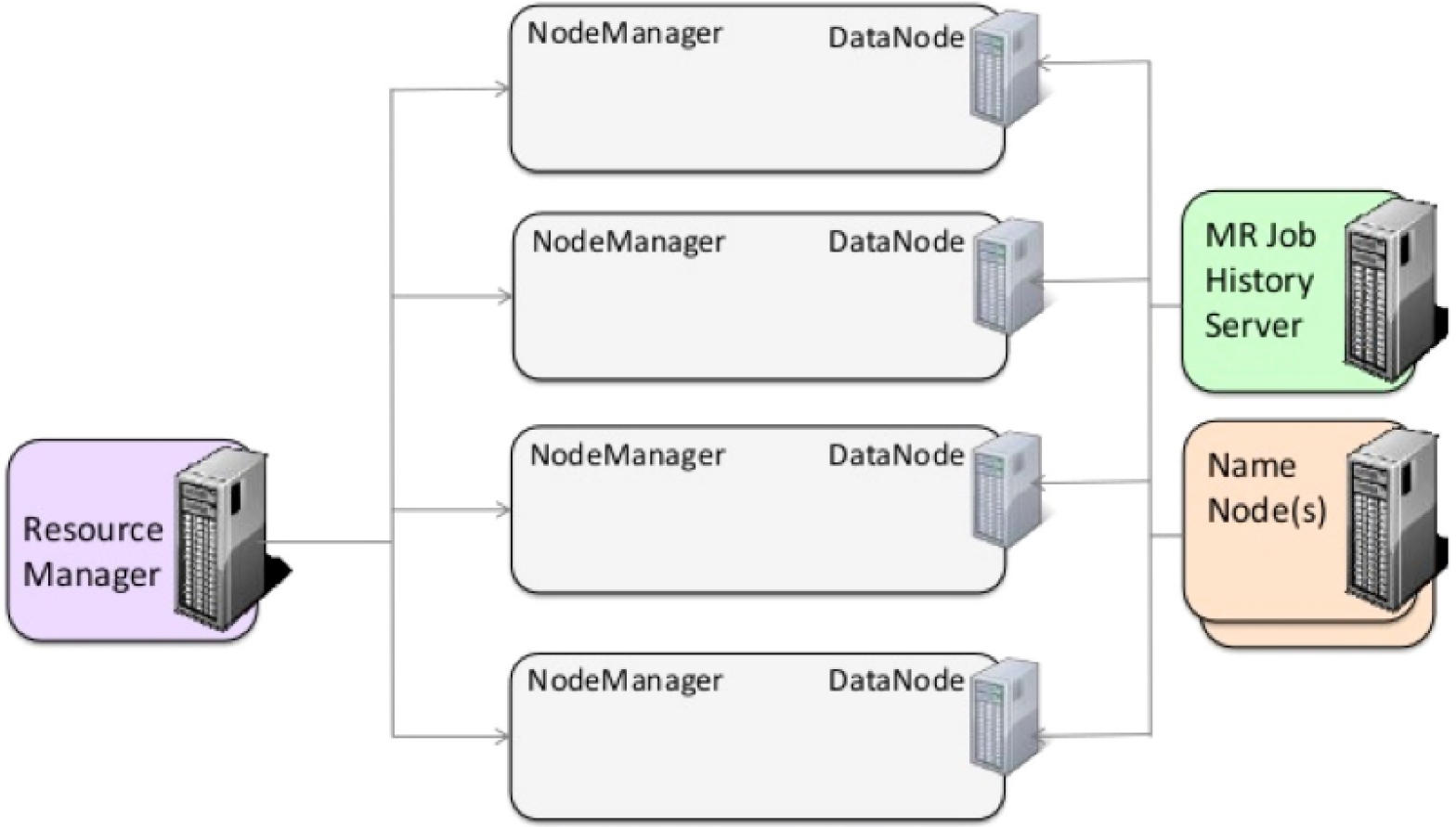


# Running MR2 on YARN

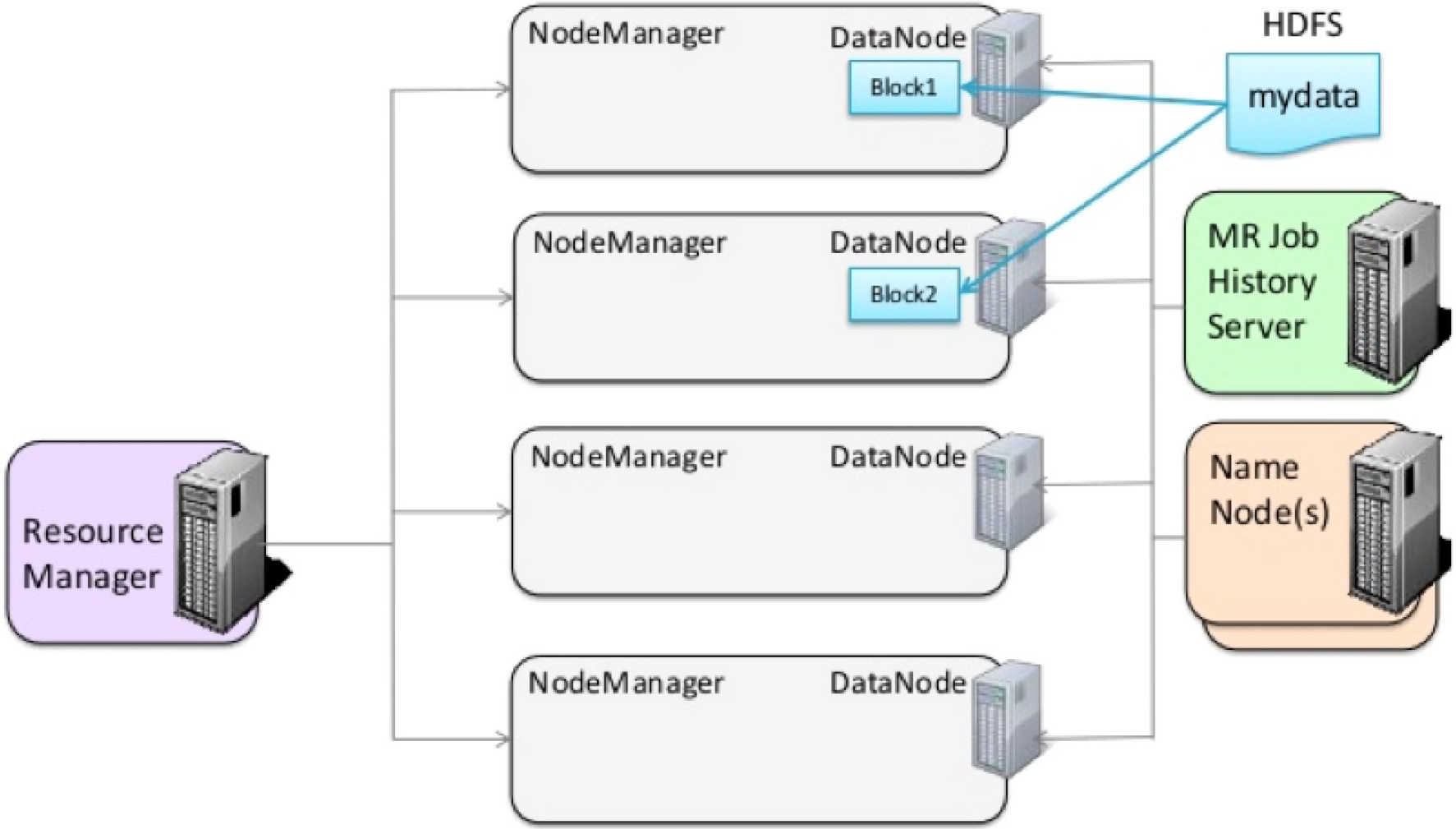
- **YARN does not know or care what kind of application it is running**
  - Could be MR or something else (e.g. Impala)
- **MR2 uses YARN**
  - Hadoop includes a MapReduce ApplicationMaster (MRAppMaster) to manage MR jobs
  - Each MapReduce job is an a new instance of an application



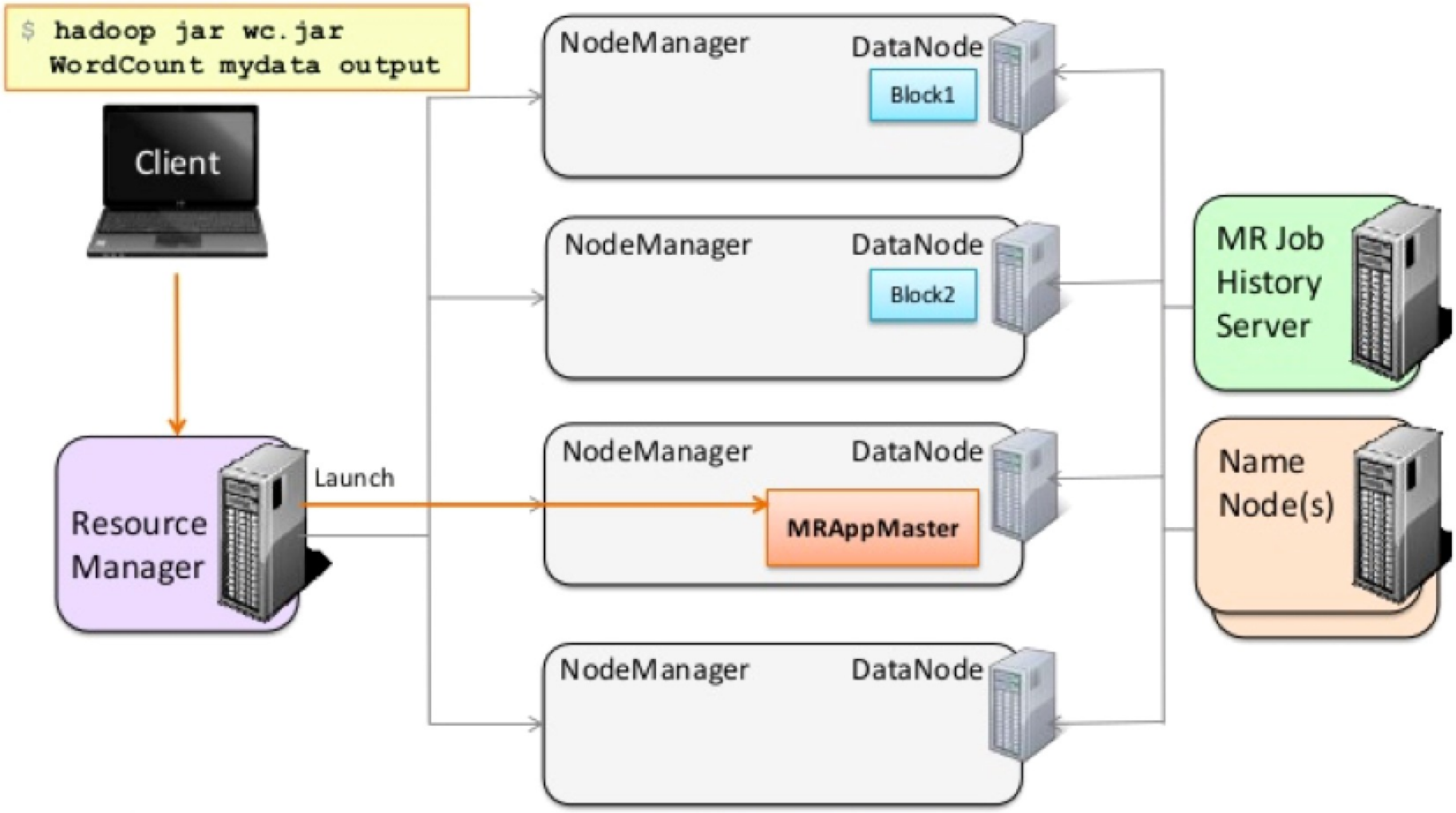
# Running a MR Job in YARN



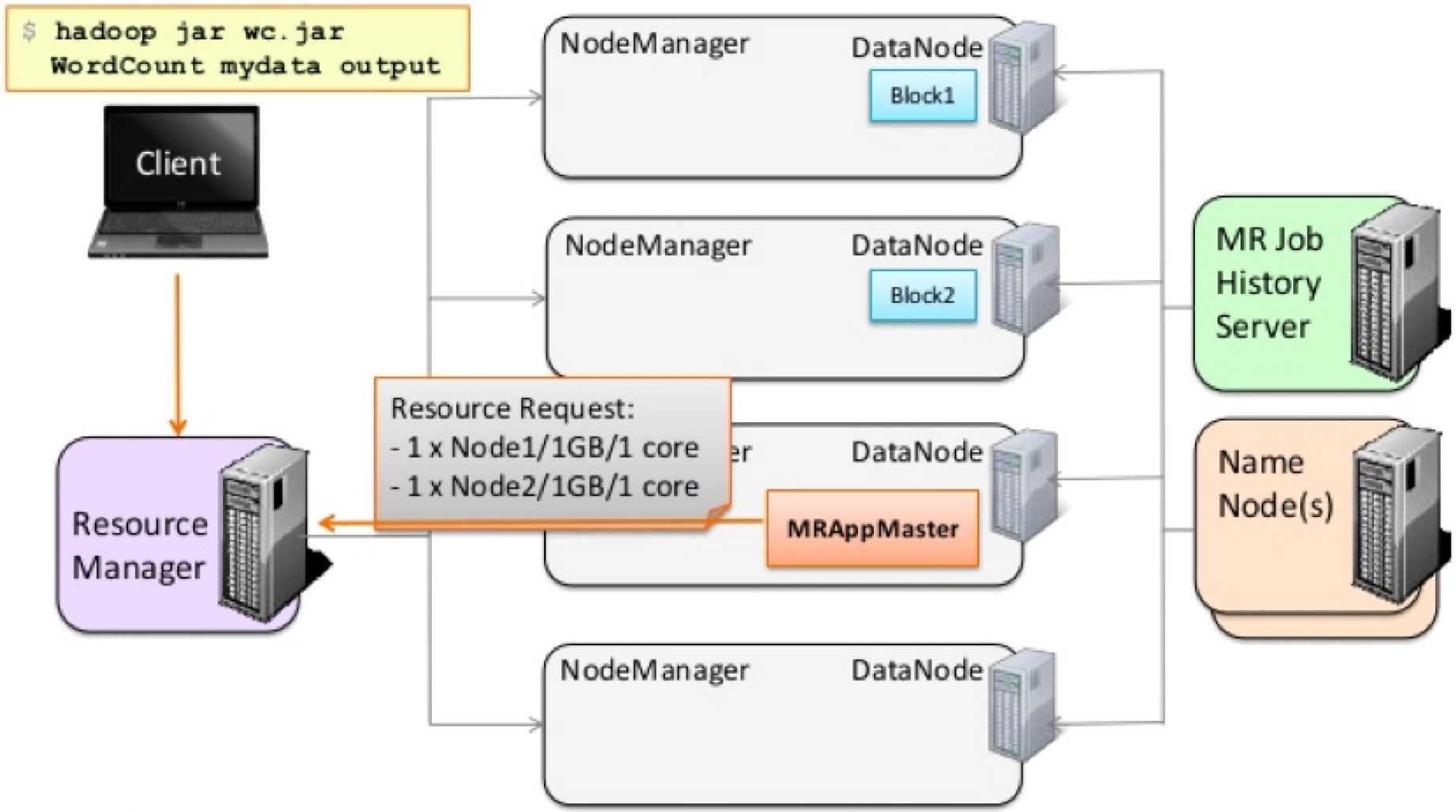
# Running a MR Job in YARN



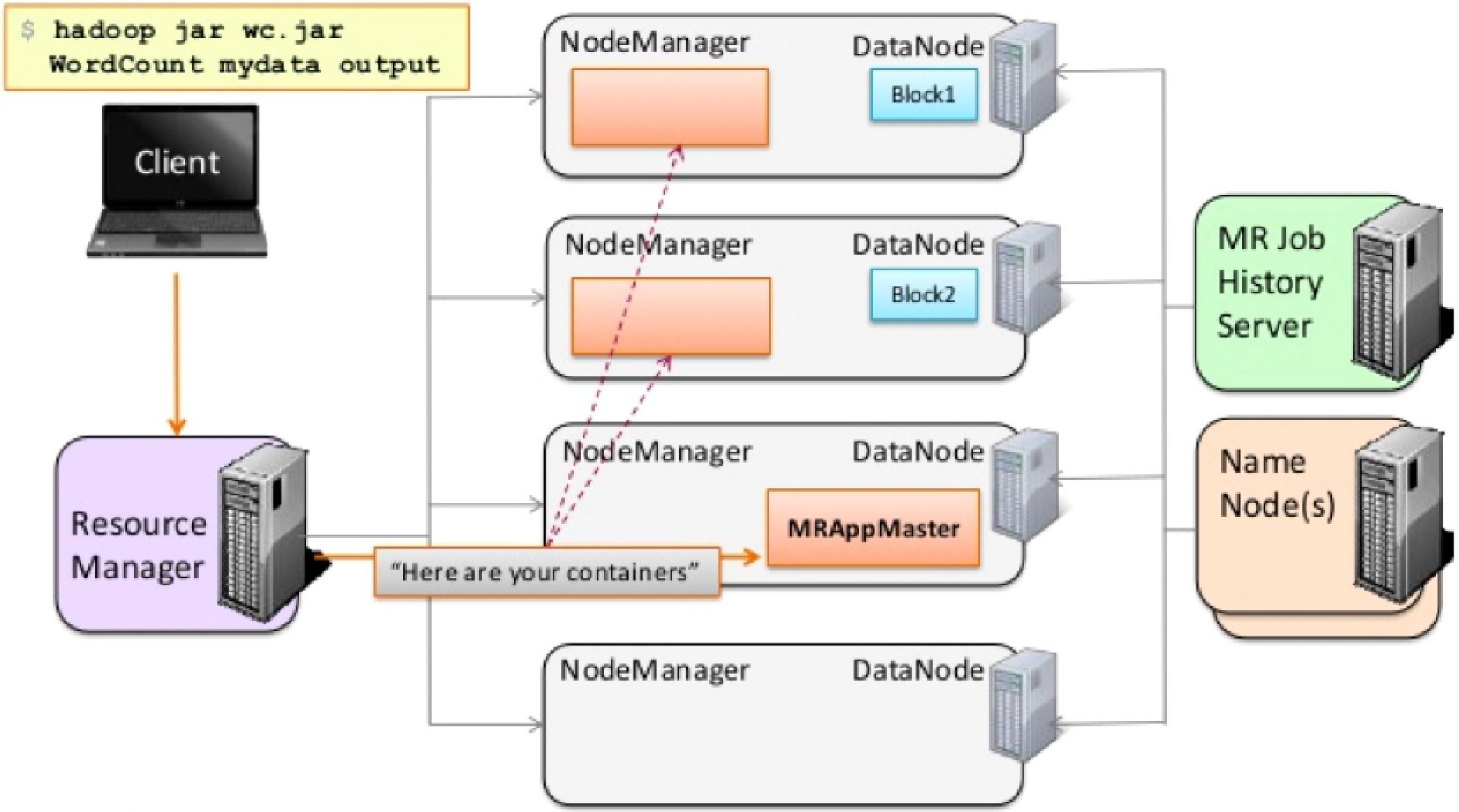
# Running a MR Job in YARN



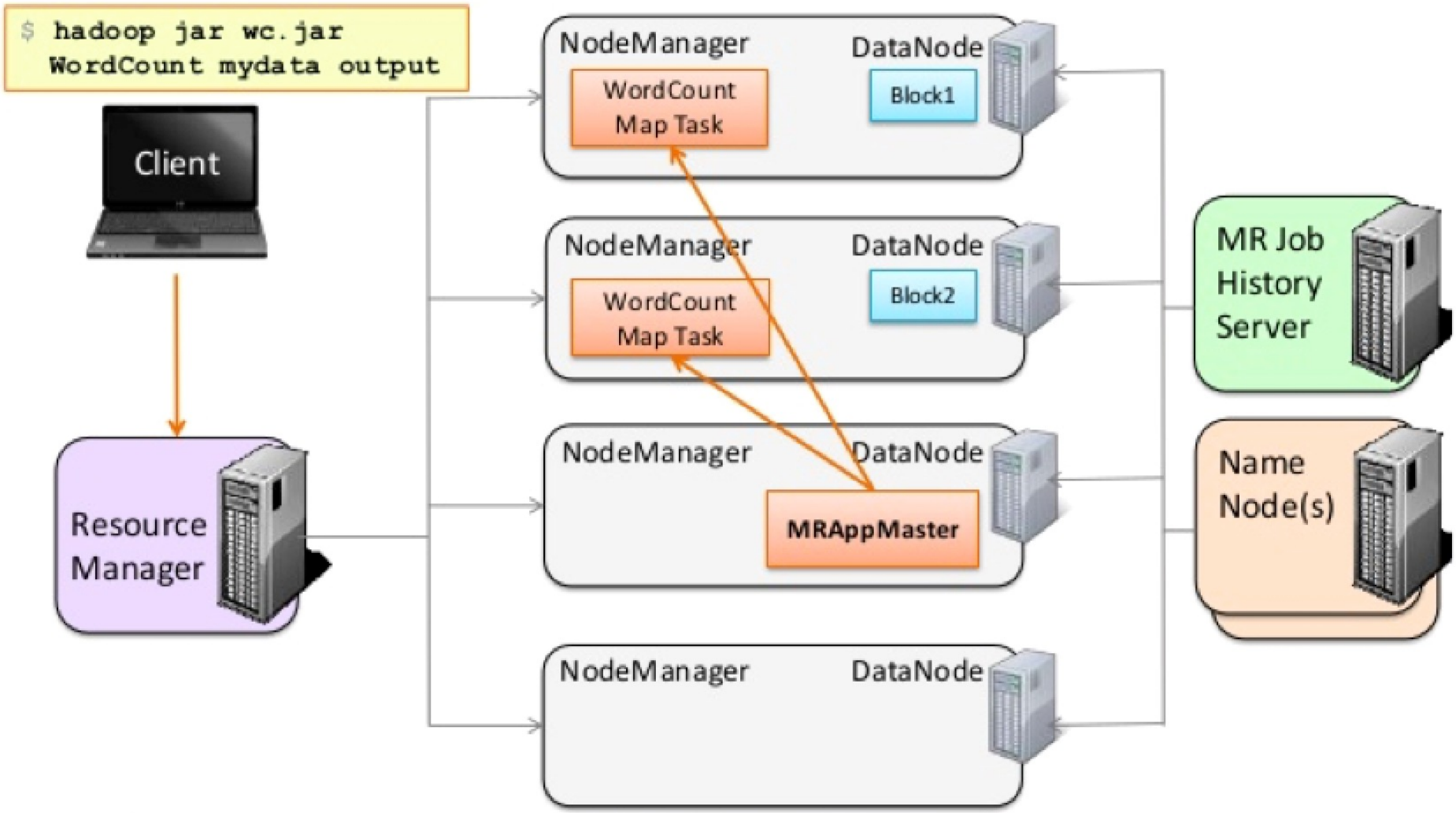
# Running a MR Job in YARN



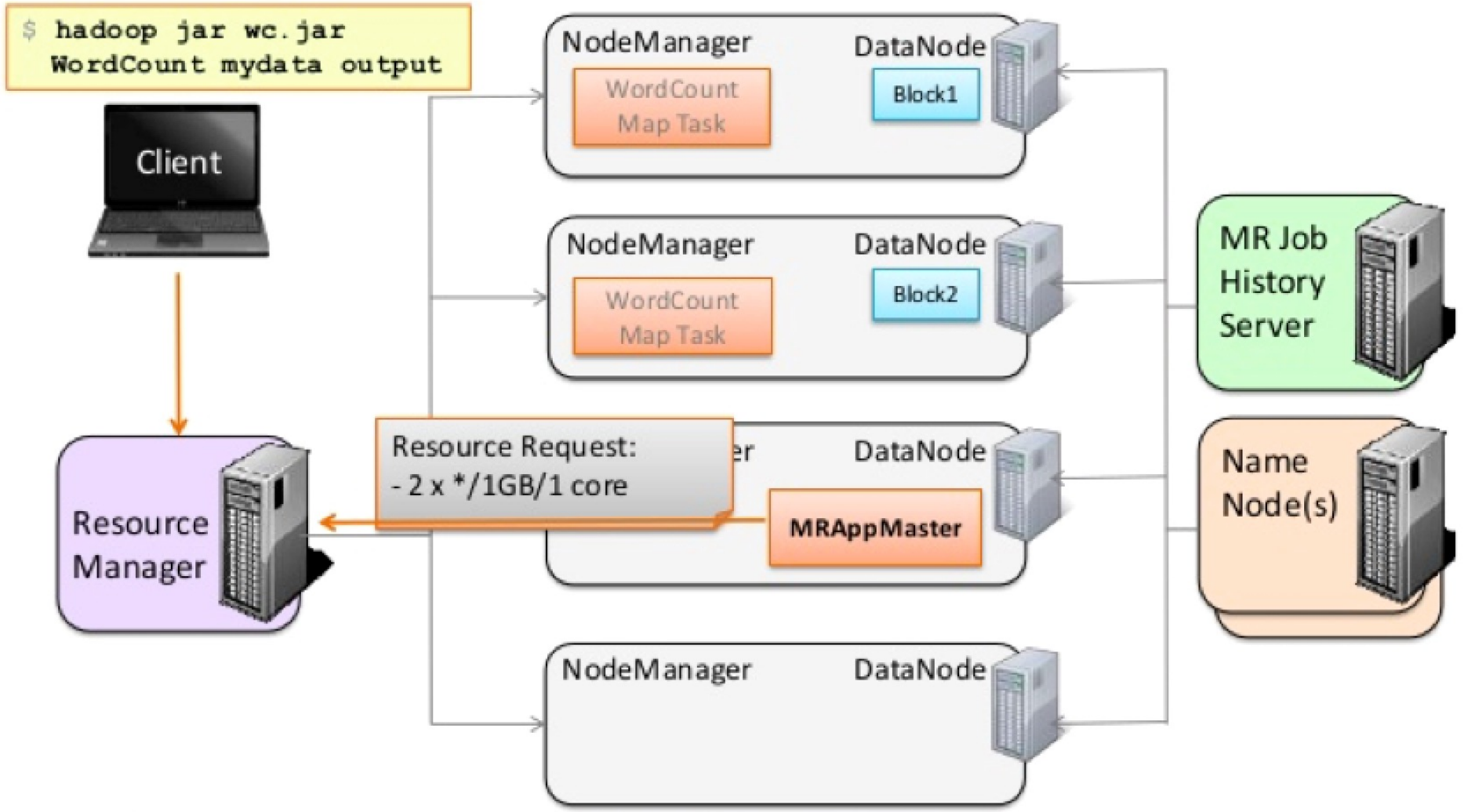
# Running a MR Job in YARN



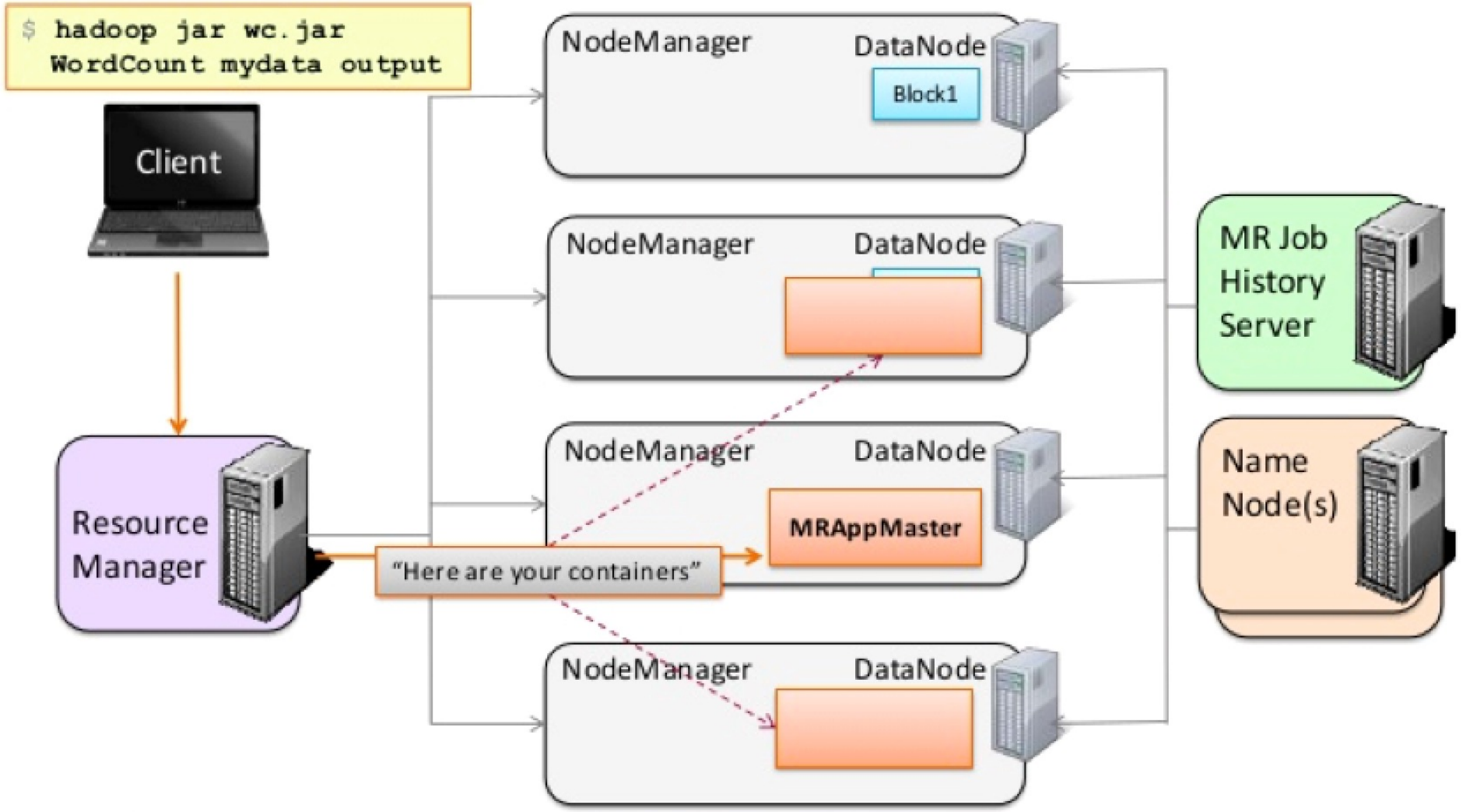
# Running a MR Job in YARN



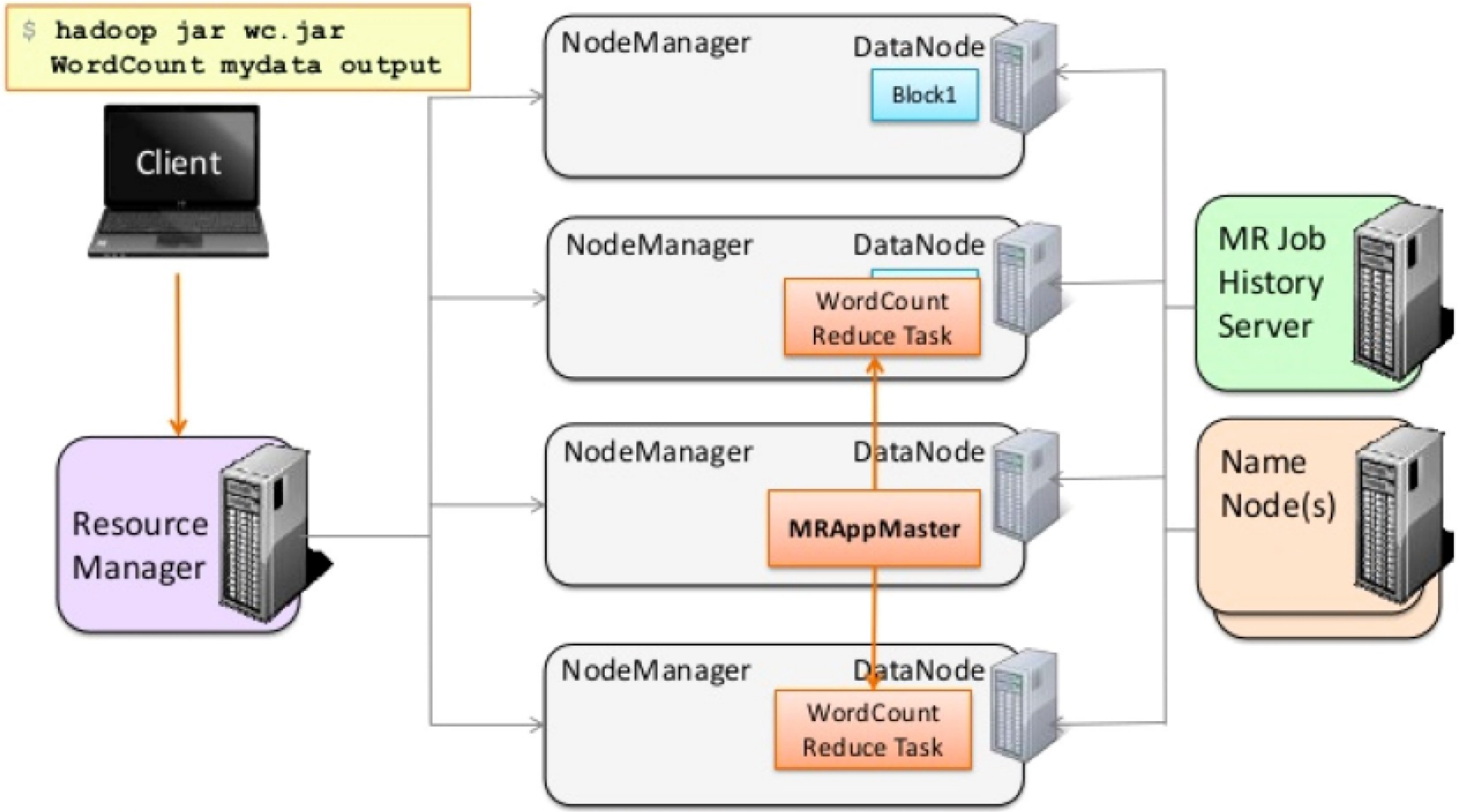
# Running a MR Job in YARN



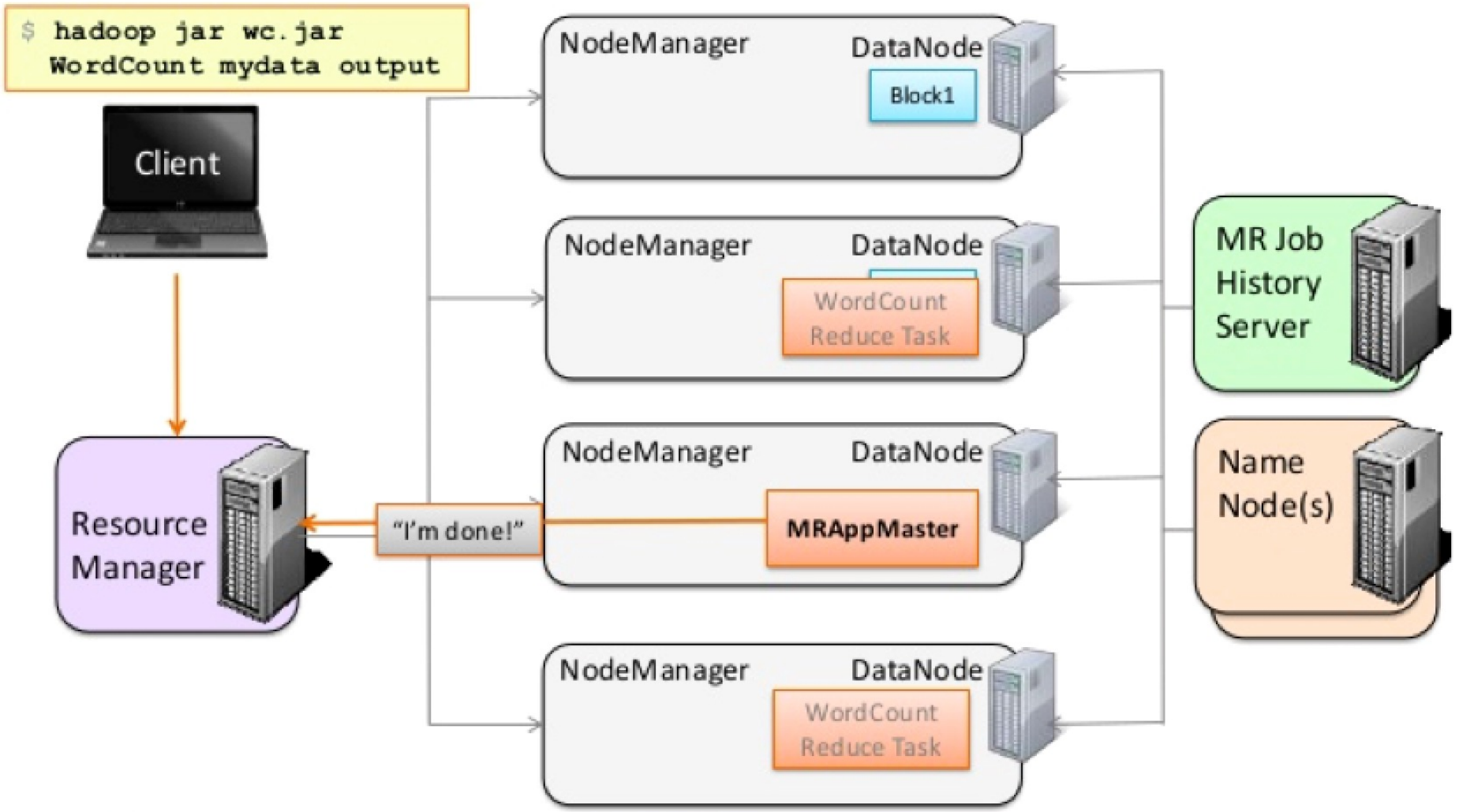
# Running a MR Job in YARN



# Running a MR Job in YARN

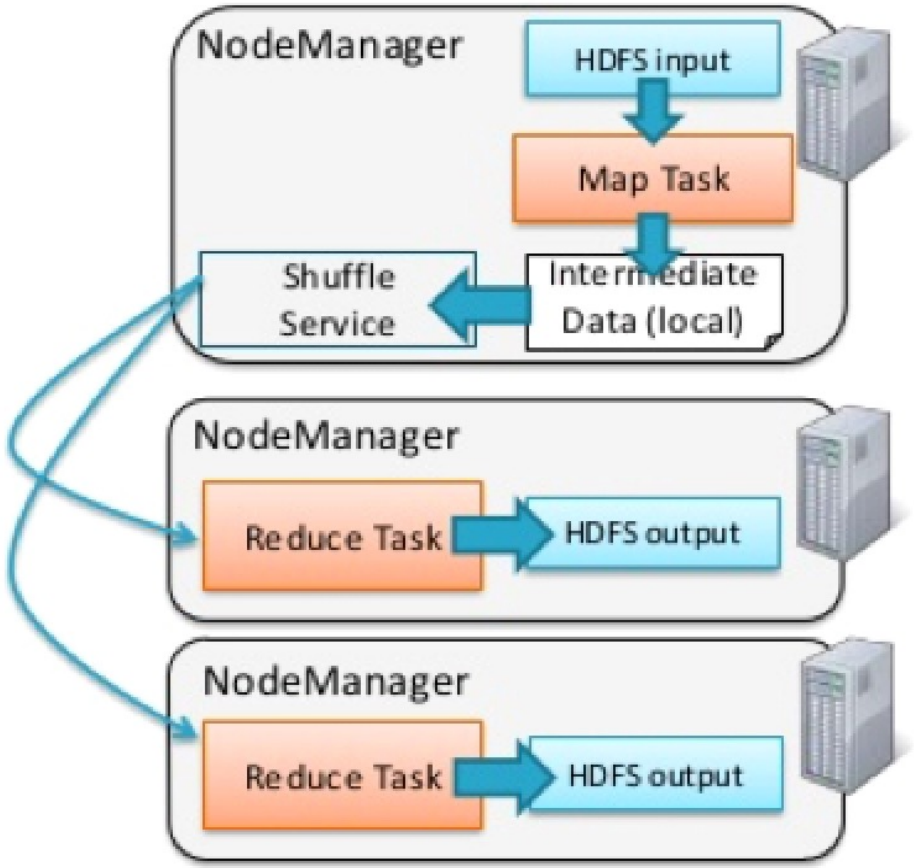


# Running a MR Job in YARN

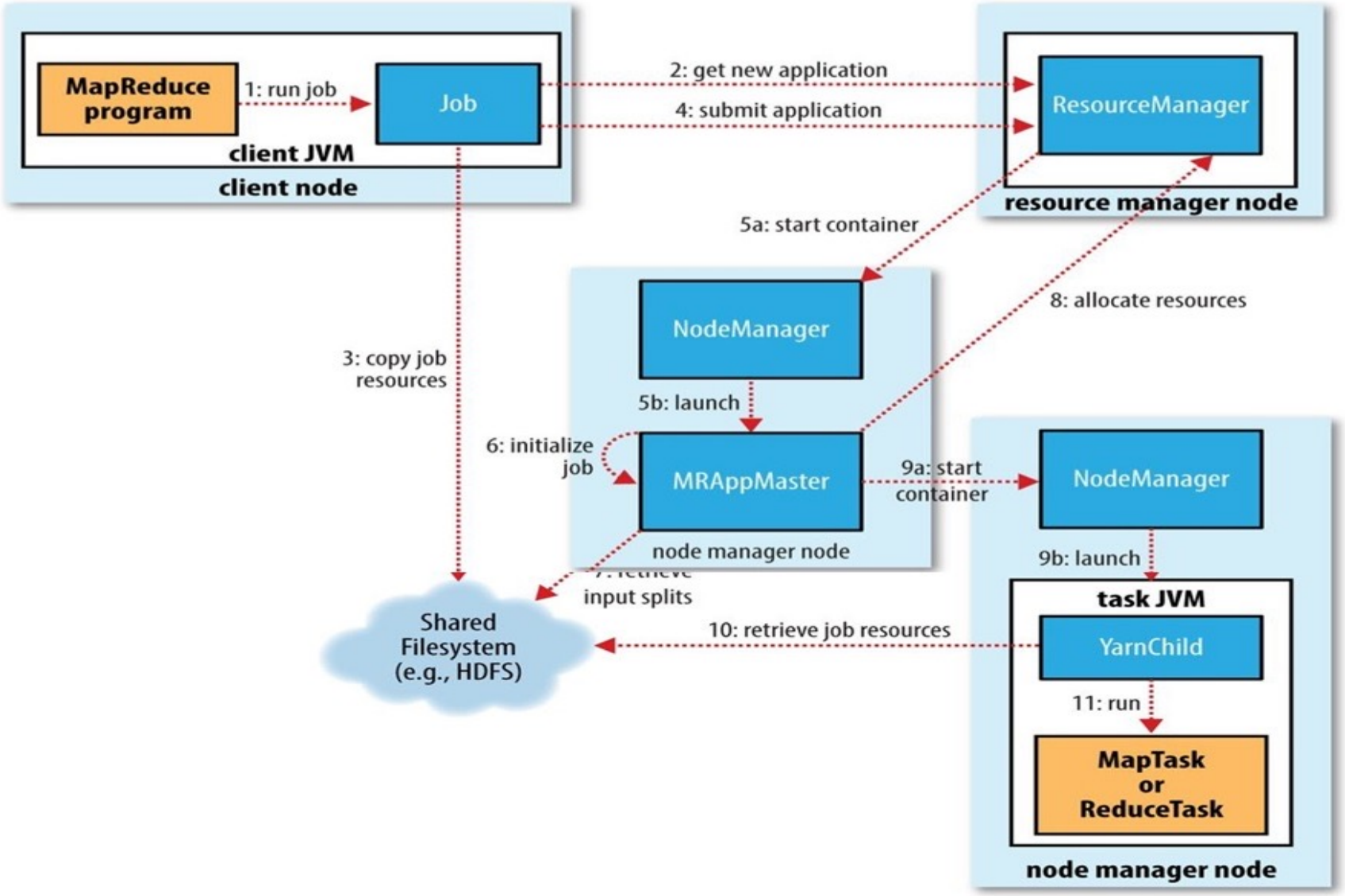


# Shuffle in YARN

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



# Running a MR Job in YARN



# Job Submission

The submit() method on Job creates an internal Job Submitter instance and calls submit Job Internal() on it (step 1 ). Having submitted the job, wait For Completion() polls the job's progress once per second and reports the progress to the console if it has changed since the last report. When the job completes successfully,
the job counters are displayed. Otherwise, the error that caused the job to fail is logged to the console.

**The job submission process implemented by Job Submitter does the following:**

- Asks the resource manager for a new application ID, used for the MapReduce job ID (step 2).
- Checks the output specification of the job. For example, if the output directory has not been specified or it already exists, the job is not submitted and an error is thrown to the MapReduce program.
- Computes the input splits for the job. If the splits cannot be computed (because the input paths don't exist, for example), the job is not submitted and an error is thrown to the MapReduce program.
- Copies the resources needed to run the job, including the job JAR file, the configuration file, and the computed input splits, to the shared file system in a directory named after the job ID (step 3). The job JAR is copied with a high replication factor (controlled by the mapreduce.client.submit.file.replication property, which defaults to 10) so that there are lots of copies across the cluster for the node managers to access when they run tasks for the job.
- Submits the job by calling submit Application() on the resource manager (step 4).

# Job Initialization

- When the resource manager receives a call to its submit Application() method, it hands off the request to the YARN scheduler.
- The scheduler allocates a container, and the resource manager then launches the application master's process there, under the node manager's management (steps 5a and 5b).
- The application master for MapReduce jobs is a Java application whose main class is MRAppMaster.
- It initializes the job by creating a number of bookkeeping objects to keep track of the job's progress, as it will receive progress and completion reports from the tasks (step 6).
- Next, it retrieves the input splits computed in the client from the shared filesystem (step 7).
- It then creates a map task object for each split, as well as a number of reduce task objects determined by the mapreduce.job.reduces property (set by the set Num  Reduce Tasks() method on Job). Tasks are given IDs at this point.

# Task Assignment

- The application master requests containers for all the map and reduce tasks in the job from the resource manager (step 8).
- Requests for map tasks are made first and with a higher priority than those for reduce tasks, since all the map tasks must complete before the sort phase of the reduce can start.
- Requests for reduce tasks are not made until 5% of map tasks have completed.
- Reduce tasks can run anywhere in the cluster, but requests for map tasks have data locality constraints that the scheduler tries to honor.
- In the optimal case, the task is data local — that is, running on the same node that the split resides on.
- Alternatively, the task may be rack local: on the same rack, but not the same node, as the split.
- Some tasks are neither data local nor rack local and retrieve their data from a different rack than the one they are running on. For a particular job run, you can determine the number of tasks that ran at each locality level by looking at the job's counters.
- Requests also specify memory requirements and CPUs for tasks. By default, each map and reduce task is allocated 1,024 MB of memory and one virtual core.

# Task Execution

- Once a task has been assigned resources for a container on a particular node by the resource manager's scheduler, the application master starts the container by contacting the node manager (steps 9a and 9b). The task is executed by a Java application whose main class is Yan Child
- Before it can run the task, it localizes the resources that the task needs, including the job configuration and JAR file, and any files from the distributed cache.
- Finally, it runs the map or reduce task (step 11).
- The YarnChild runs in a dedicated JVM, so that any bugs in the user-defined map and reduce functions (or even in YarnChild) don't affect the node manager — by causing it to crash or hang, for example.
- Each task can perform setup and commit actions, which are run in the same JVM as the task itself.

# Progress a MapReduce

**All of the following operations constitute progress:**

- Reading an input record (in a mapper or reducer)
- Writing an output record (in a mapper or reducer)
- Setting the status description on a reporter (using Reporter's setStatus() method)
- Incrementing a counter (using Reporter's incrCounter() method)
- Calling Reporter's progress() method

# Job Completion

- When the application master receives a notification that the last task for a job is complete, it changes the status for the job to “successful.” Then, when the Job polls for status, it learns that the job has completed successfully, so it prints a message to tell the user and then returns from the wait for completion() method. Job statistics and counters are printed to the console at this point.
- The application master also sends an HTTP job notification if it is configured to do so. This can be configured by clients wishing to receive callbacks, via the mapreduce.job end-notification.url property.
- Finally, on job completion, the application master and the task containers clean up their working state (so intermediate output is deleted), and the Output  Committer's commit Job() method is called.
- Job information is archived by the job history server to enable later interrogation by users if desired.

# Next Session

Sqoop Architecture