Hadoop YARN

Department of Computer Science,

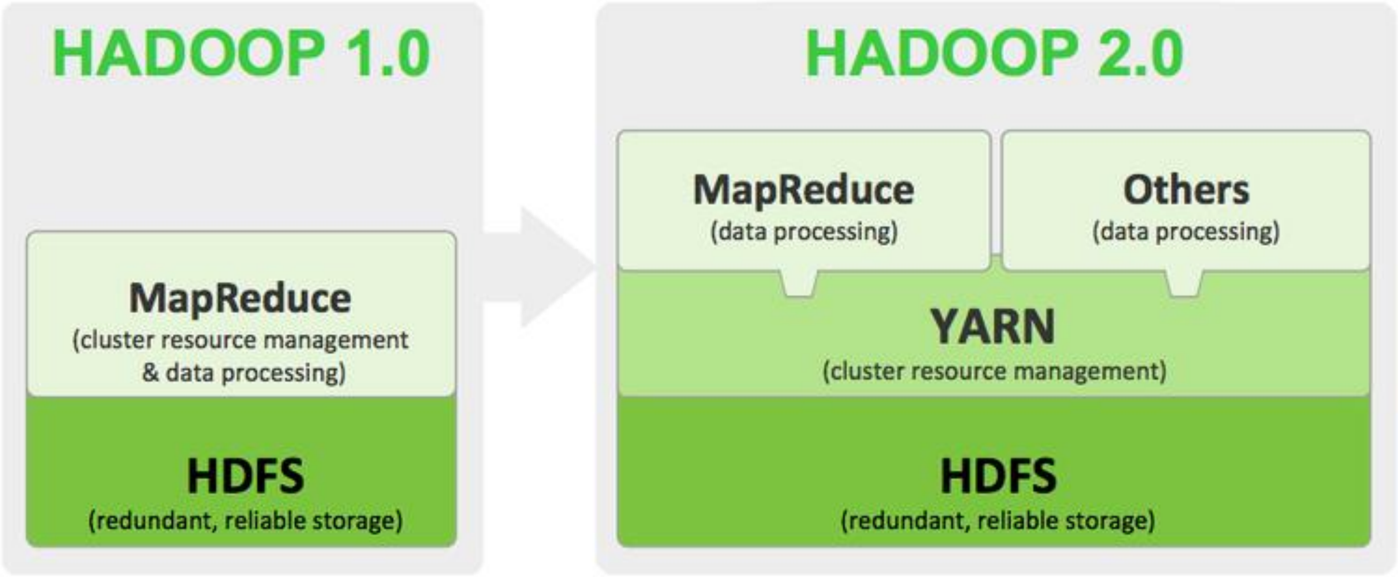
National University of Computer & Emerging Sciences,
Islamabad Campus

These slides are not ***always*** prepared by me and they do use content from different sources. Please do not share them ☺

Overview

- **YARN** was **introduced** in **Hadoop 2** initially, to **improve the MapReduce implementation**

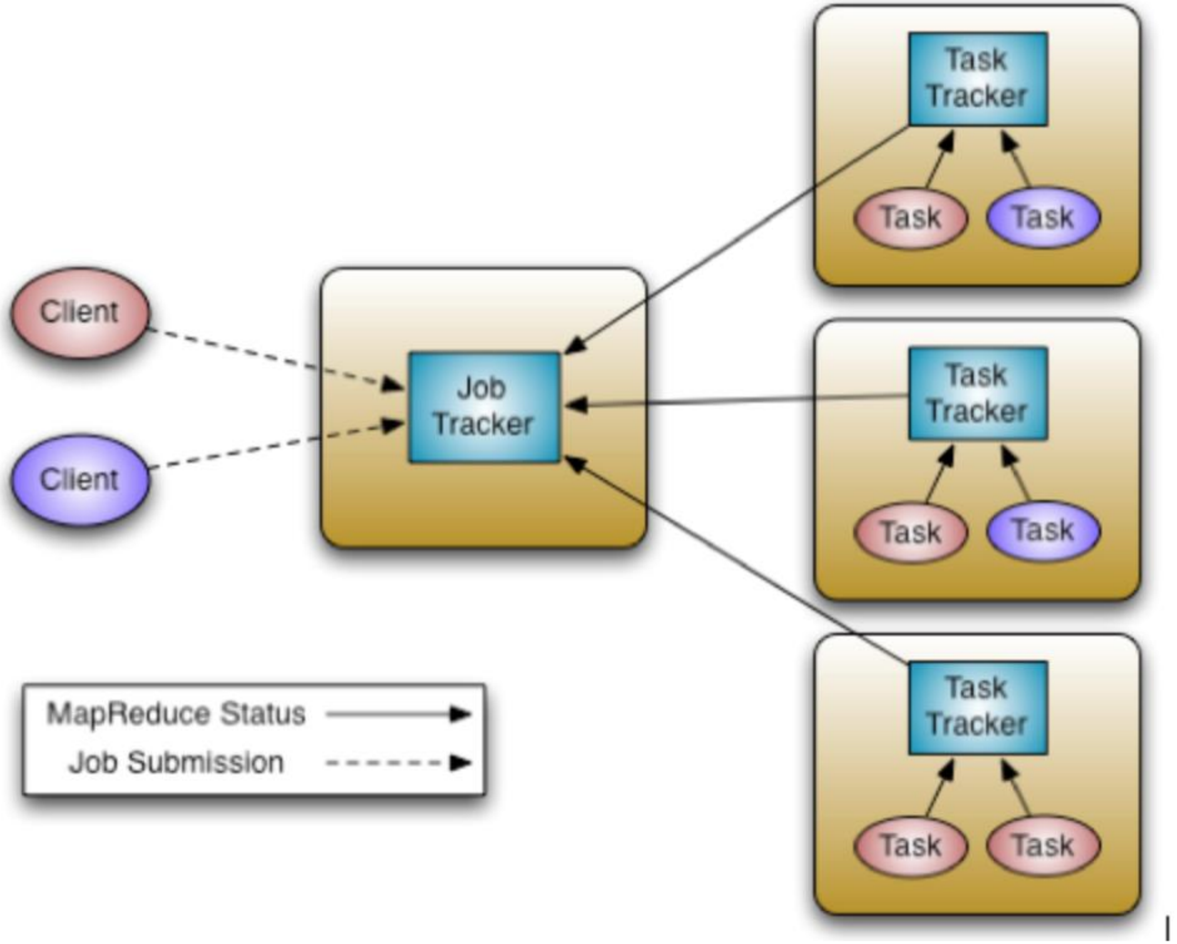
- it is general enough to **support other distributed computing paradigms as well**


HADOOP 1.0
MapReduce
(cluster resource management & data processing)
HDFS
(redundant, reliable storage)
HADOOP 2.0
MapReduce
(data processing)
Others
(data processing)
YARN
(cluster resource management)
HDFS
(redundant, reliable storage)

How does MapReduce1 work?

- In **MapReduce 1**, there are **two types** of **daemons** for **job execution process**:
  - The **jobtracker coordinates** all the **jobs** run on the system by **scheduling tasks** to **run** on **tasktrackers**.
  - **Tasktrackers run tasks** and **send progress reports** to the **jobtracker**
  - **If a task fails**, the **jobtracker** can **reschedule it** on a **different tasktracker**.

How does MapReduce 1 work?



Job tracker manages all the Jobs (in a cluster), 3 main responsibilities:

Schedules tasks of **all jobs to nodes**

Keep monitoring the **tasks executions** etc.

Records history of the **executed jobs**

The TaskTracker

- The TaskTracker has **simple** responsibilities**:**
  - launch/teardown tasks on **orders** from the **JobTracker**
  - **provide** task-status information to the **JobTracker periodically**

The JobTracker

- **Responsible for resource management**
  - **Manages** the **worker nodes** i.e., **TaskTrackers**
  - **Tracks resource consumption/availability**
- **Performs job life-cycle management**
  - **Schedules** individual **tasks** of the job
  - **Tracks progress**
  - **Provides fault-tolerance** for **tasks** etc.

Limitations?

- The **JobTracker** was **over-burdened.**
  - **Resource management**
  - **Job and task scheduling and**
  - **Monitoring**
  - …

- It was **posing a limitation** in terms of
  - **Scalability**, **Availability**, **Resource Utilization** …

Limitations - Scalability

- **MapReduce 1 hits scalability bottlenecks** in the region of **4,000 nodes** and **40,000 tasks**.
- **YARN** is **designed** to **scale up to 10,000 nodes and 100,000 tasks**

Limitations - Non MapReduce Tasks

- **Job tracker** was **tightly integrated** with **MapReduce**

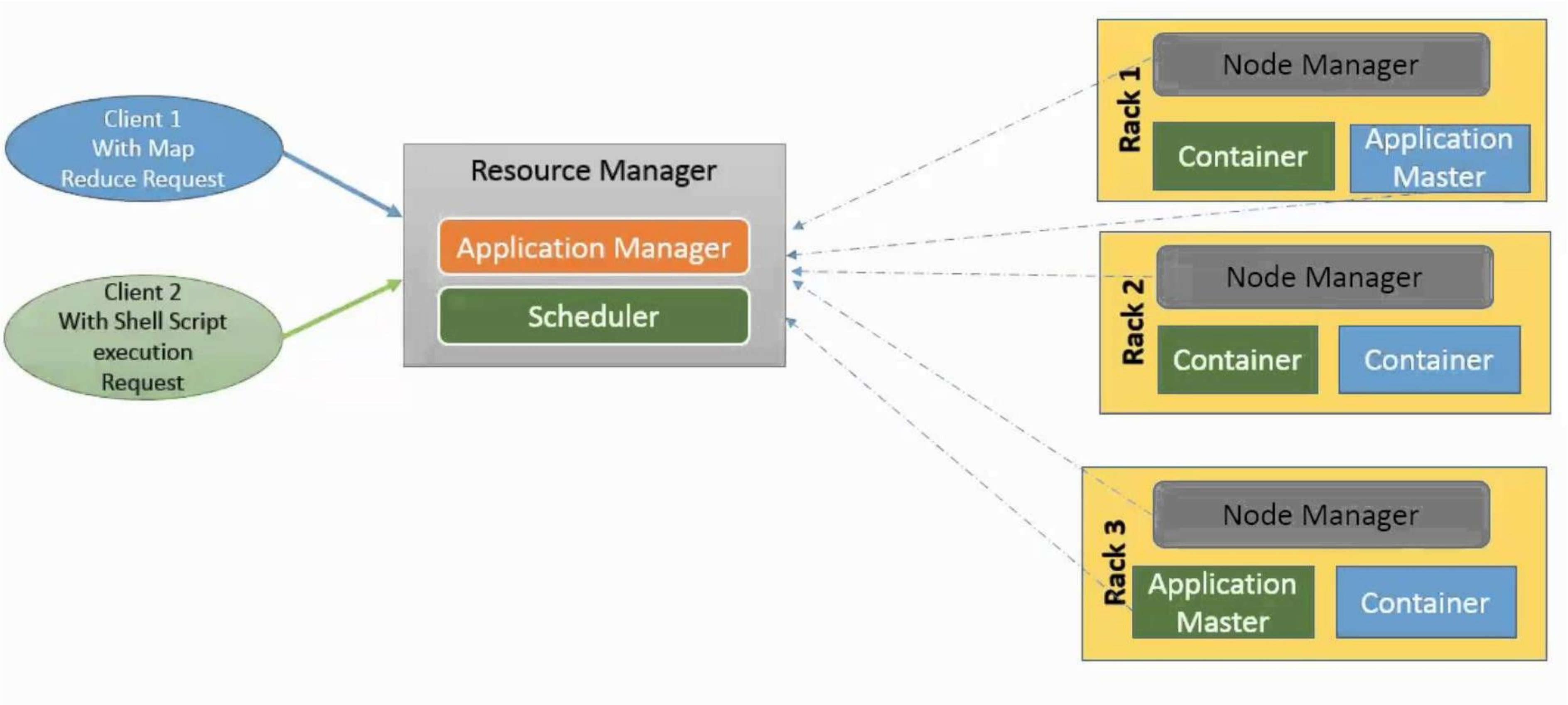
- **MapReduce works well** but **provides batch processing** and **lacks real-time analysis**

- It **should** thus be **possible to support other workloads** in **Hadoop**

Hadoop YARN

- **Fundamental idea** - **divide and conquer**

- **Split up** the **two major responsibilities** of the **JobTracker** i.e., **resource management** and **job scheduling/monitoring**, into **separate daemons**:
  - a **global** **ResourceManager** **(one per Cluster)**
  - a **number** of **NodeManagers** **(One per node)**
  - and **per-application** **ApplicationMaster** (**AM**).


Client 1
With Map
Reduce Request
Client 2
With Shell Script
execution
Request
Resource Manager
Application Manager
Scheduler
Rack 1
Node Manager
Container
Application
Master
Rack 2
Node Manager
Container
Container
Rack 3
Node Manager
Application
Master
Container

ResourceManager

- The **ResourceManager** is the **ultimate authority** that **arbitrates (i.e., allocates) resources** among all the **applications** in the **system**.

NodeManager

- The **NodeManager** is a **per node worker** **service** that is **responsible** for the **execution of containers** **based** on the **node capacity**

- The **NodeManager** service **sends a heartbeat signal** to the **ResourceManager** to **update its health status**

YARN's ApplicationMaster

- **Per application framework-specific library** that **manages** each **instance** of an **application** that **runs within YARN**.

- **Each application running** on the **Hadoop cluster** has its own, **dedicated Application Master instance**
  - MapReduce **ApplicationMaster** to **run mapreduce jobs**
  - ***Tez ApplicationsMaster***, ***SPARK ApplicationMaster*** …

- The **Application Master oversees** the **full lifecycle** of an **application**

MapReduce 1 Vs. YARN

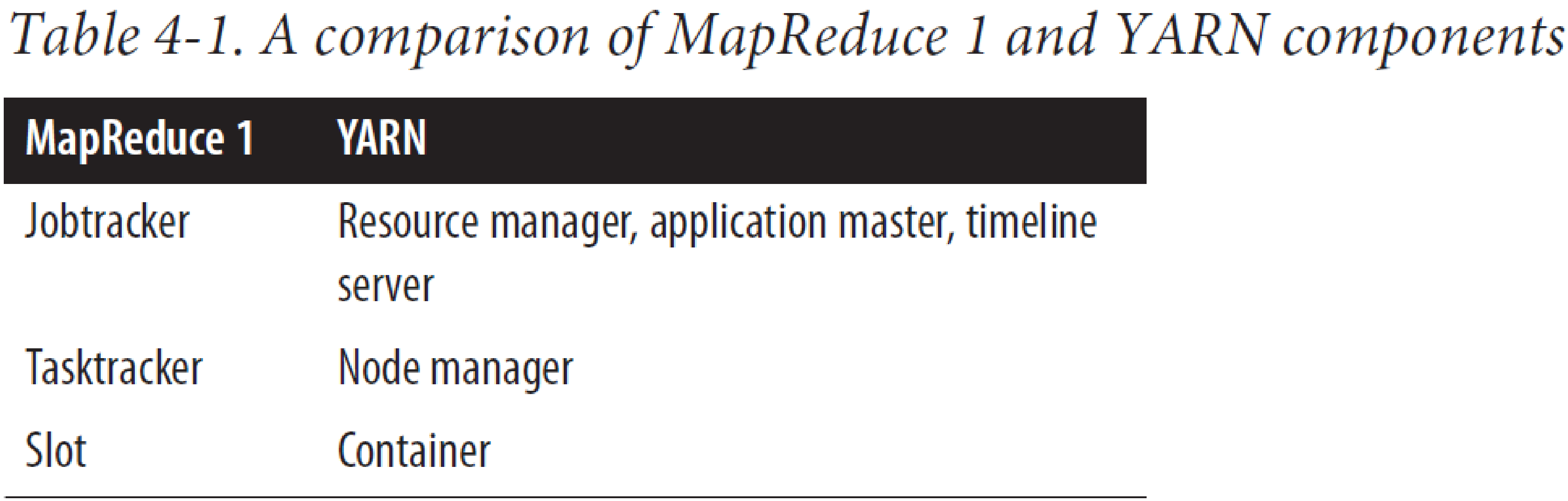
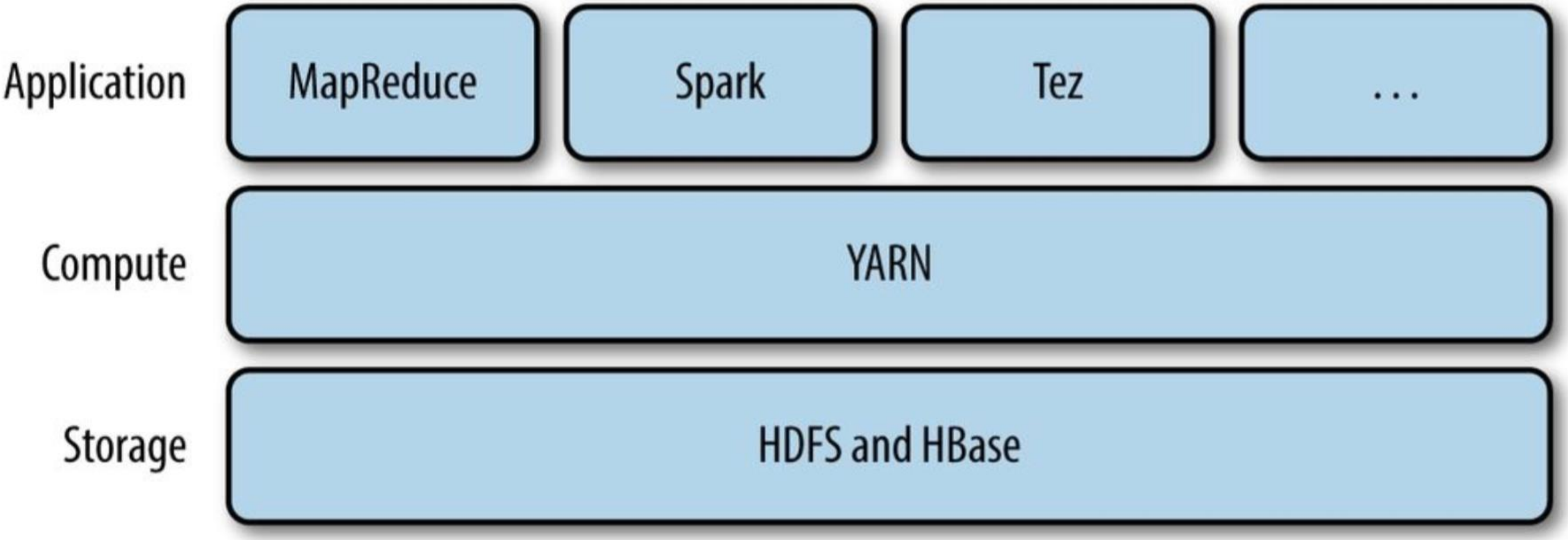


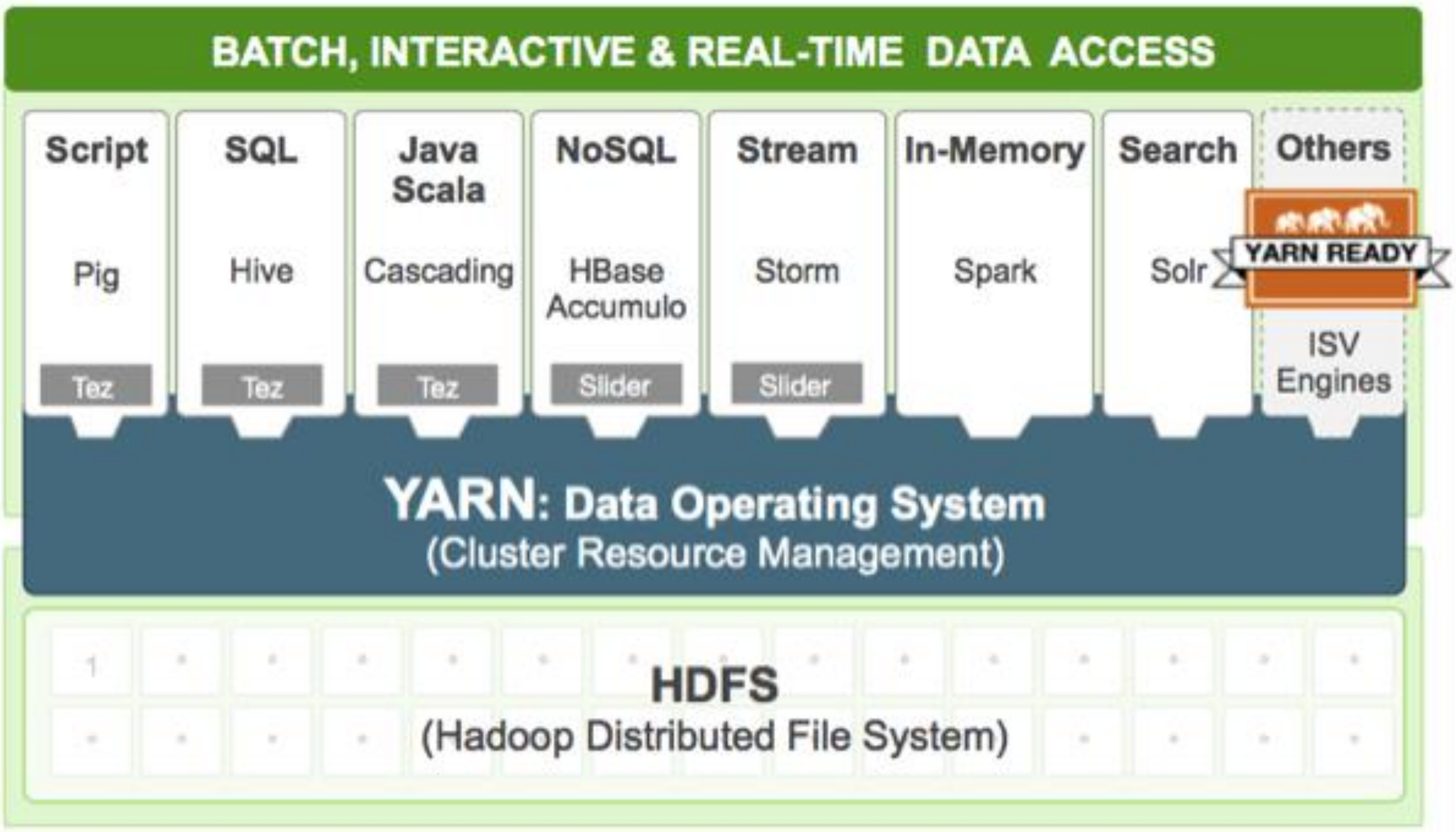
Table 4-1. A comparison of MapReduce 1 and YARN components

MapReduce 1	YARN
Jobtracker	Resource manager, application master, timeline server
Tasktracker	Node manager
Slot	Container

YARN Applications



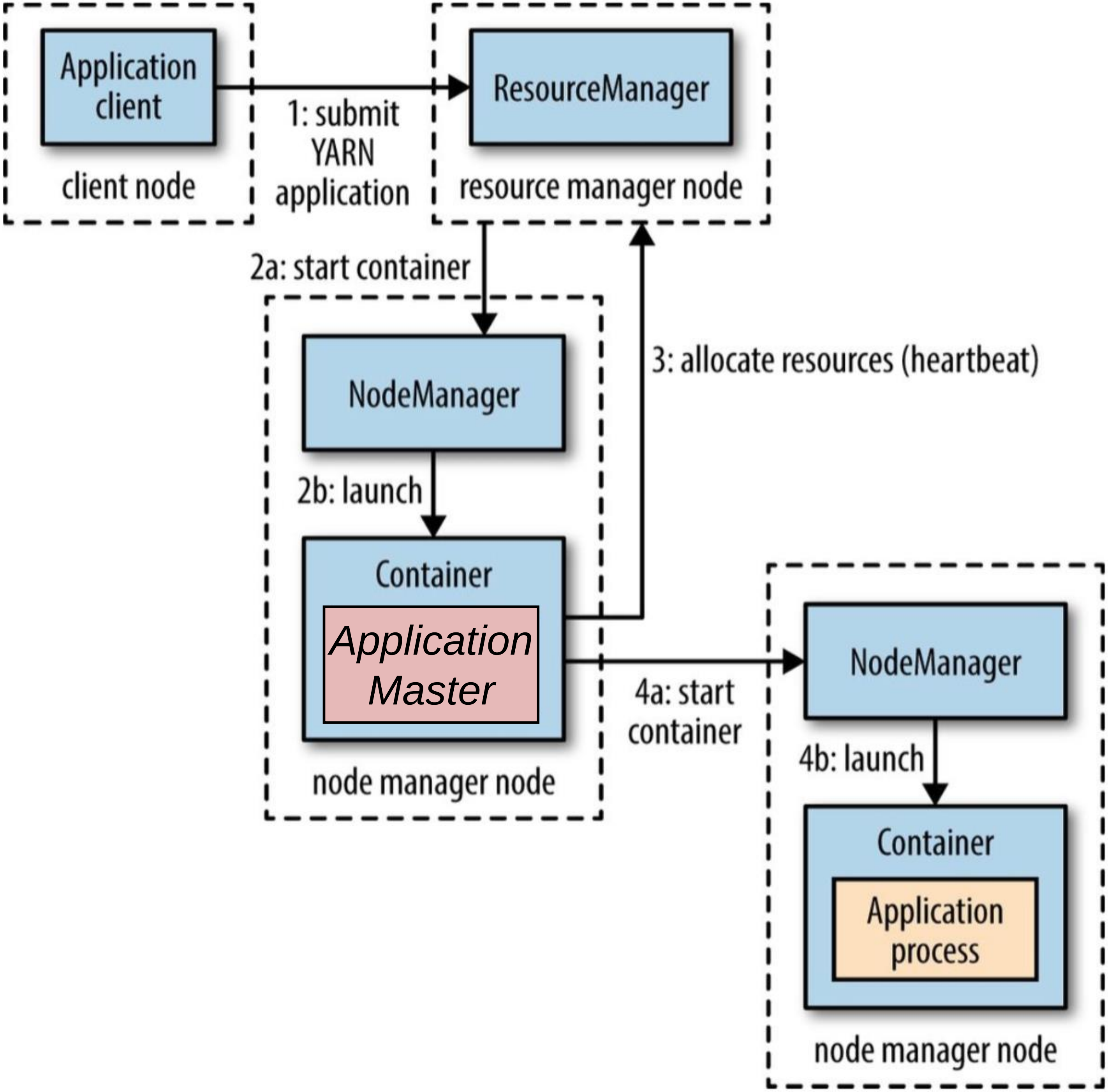
YARN Applications



Anatomy of a YARN Application Run

- To **run an application on YARN**, a **client contacts** the **resource manager** and asks it to **run an application** ***master process***.

- The **resource manager** then **finds** a **node manager** that can **launch** the **application master** in a **container** (steps 2a and 2b).


Application client
client node
1: submit YARN application
ResourceManager
resource manager node
2a: start container
3: allocate resources (heartbeat)
NodeManager
2b: launch
Container
Application Master
node manager node
4a: start container
NodeManager
4b: launch
Container
Application process
node manager node

Anatomy of a YARN Application Run

- **ApplicationMaster** is **framework specific** and can simply **run** a **computation** in the **container it is running** in

- It could **request more containers** from the **resource managers** **(step 3)**, and **use them** to **run a distributed computation** **(steps 4a and 4b)**.

Resource Requests

- A **request** for a **set of containers** can **express** the **amount** of **computer resources required** (***memory*** and ***CPU***), as well as **locality constraints**.

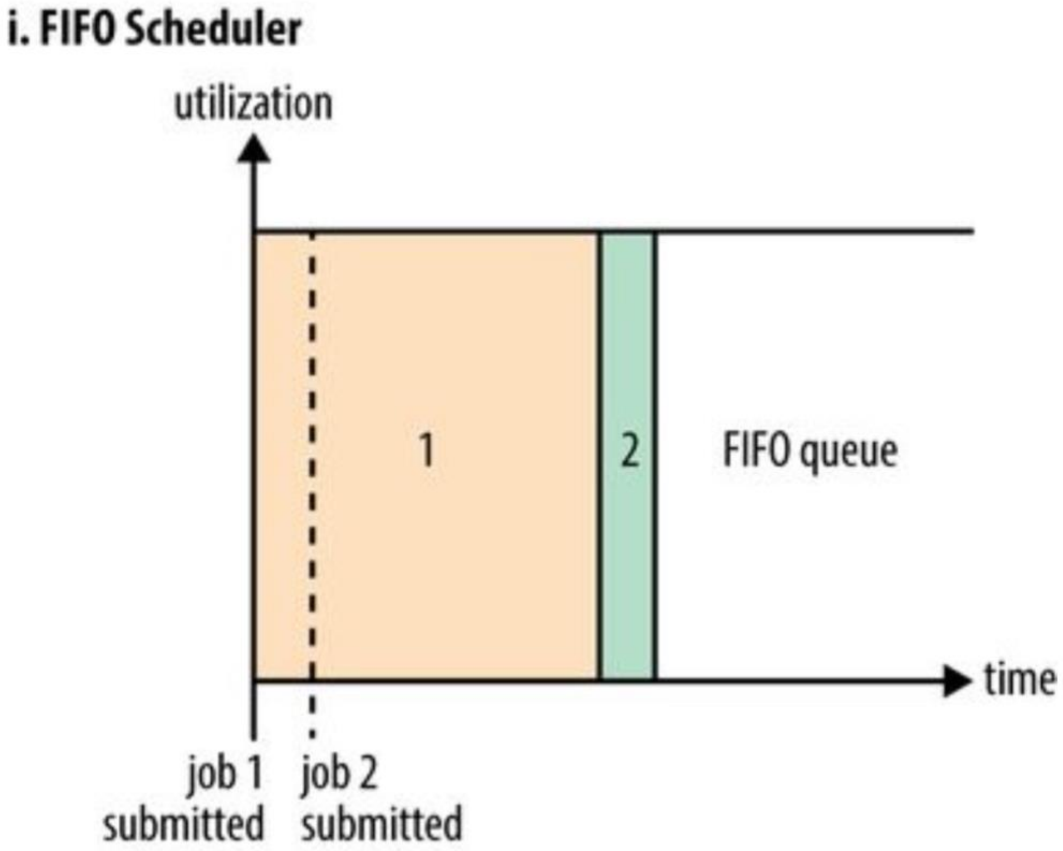
- **Locality constraints** can be used to **request a container** on a **specific node** or **rack**, or **anywhere on the cluster** (***off-rack***).

Scheduling in YARN

- **It is** the **job** of the **YARN scheduler** to **allocate resources to applications according** to some **defined policy**.

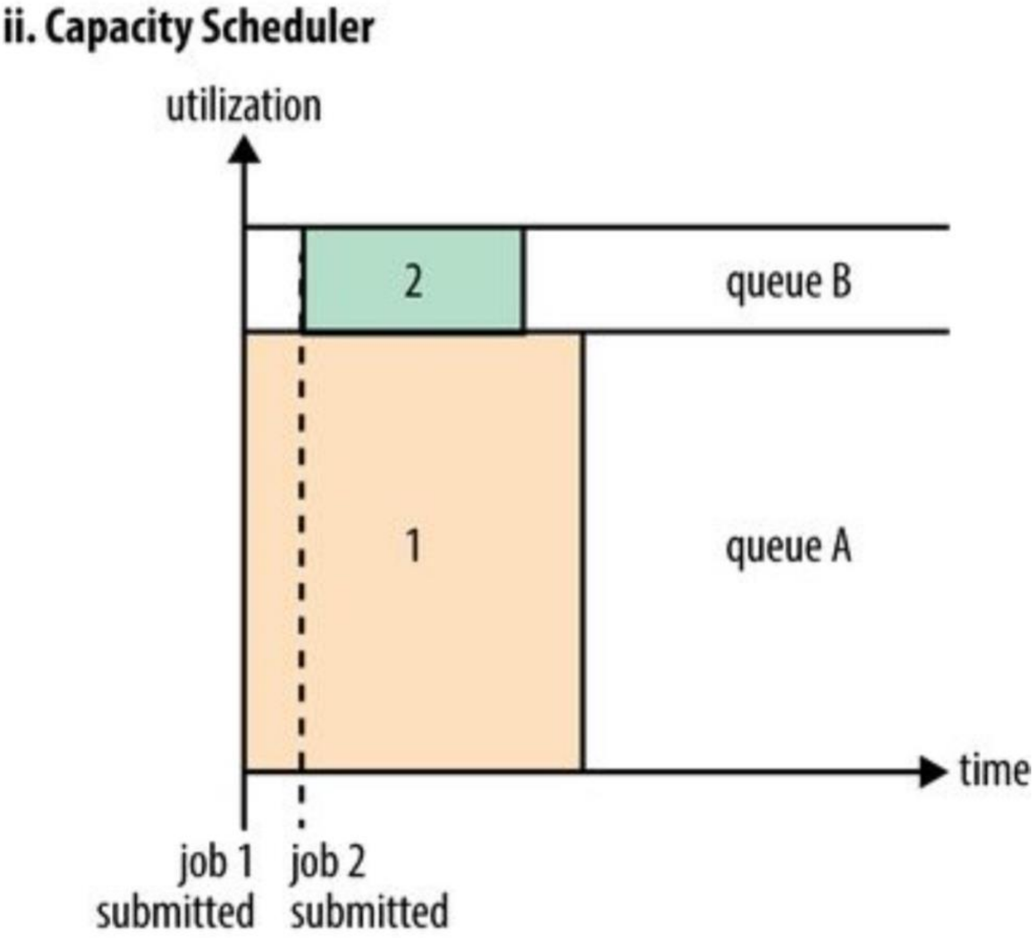
- **Three schedulers** are **available** in **YARN**: the ***FIFO***, ***Capacity***, and ***Fair Schedulers***.

YARN FIFO Scheduler



- One job execution at a time (first come first server)
- Small jobs have to wait for larger jobs to finish
- Single job utilizes full cluster resources

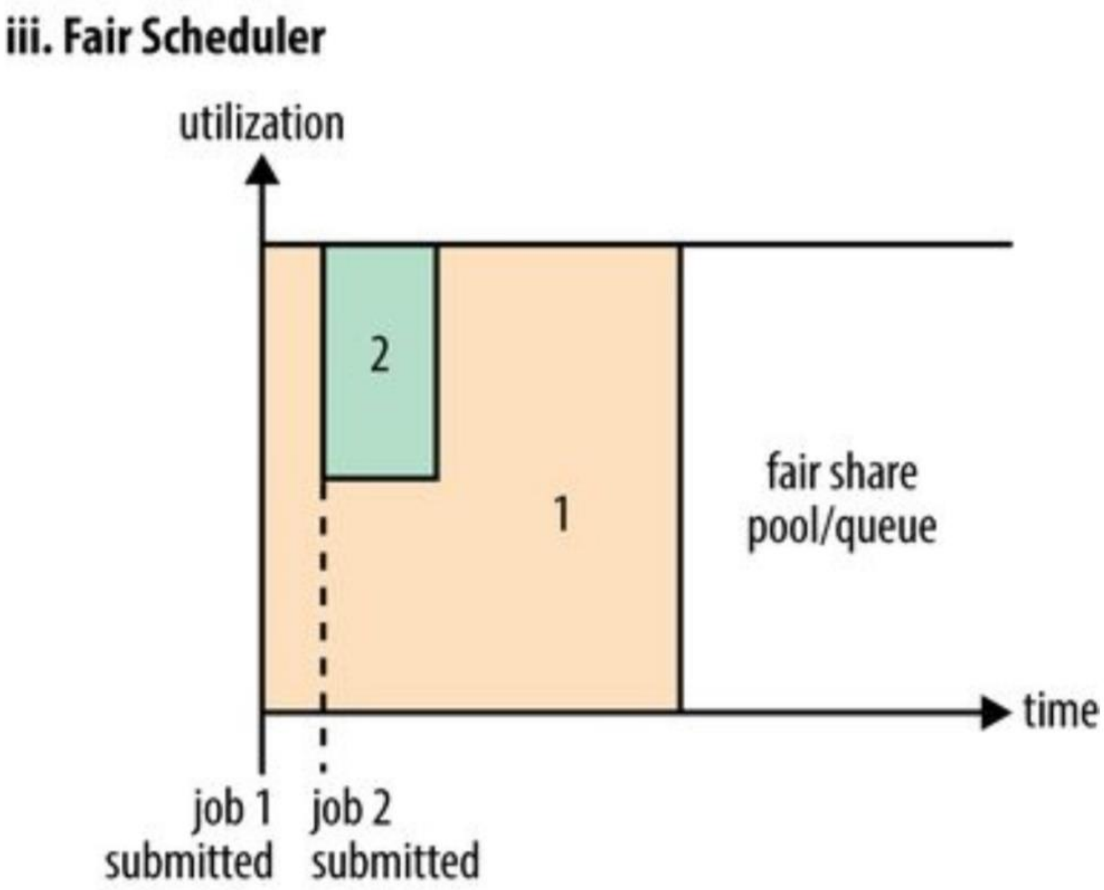
YARN Capacity Scheduler



ii. Capacity Scheduler

- Capacities for different queues are reserved (e.g., Queue for small, medium, or large jobs etc.)
- Simultaneous execution of the jobs

YARN Fair Scheduler



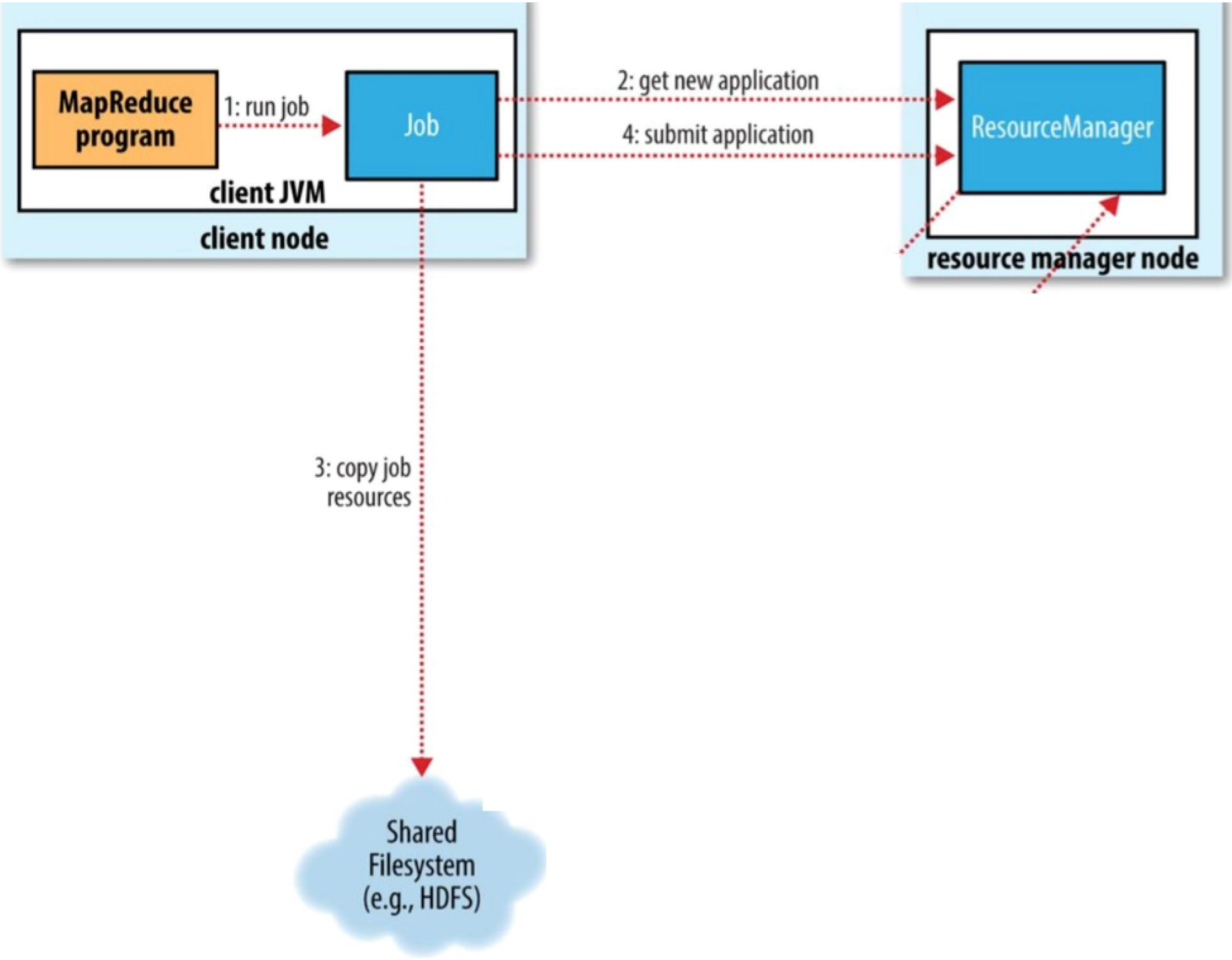
- Simultaneous execution of the jobs
- Dynamic resource allocation to jobs (no reserved capacity)

Anatomy of a MapReduce Job Run

- You can **run** a **MapReduce job** with **waitForCompletion( )**, which **submits the job** and ***then waits for it to finish***.

Anatomy of a MapReduce Job Run

- **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 **distributed filesystem**, normally **HDFS**


MapReduce program
1: run job
Job
client JVM
client node
2: get new application
4: submit application
ResourceManager
resource manager node
3: copy job resources
Shared Filesystem (e.g., HDFS)

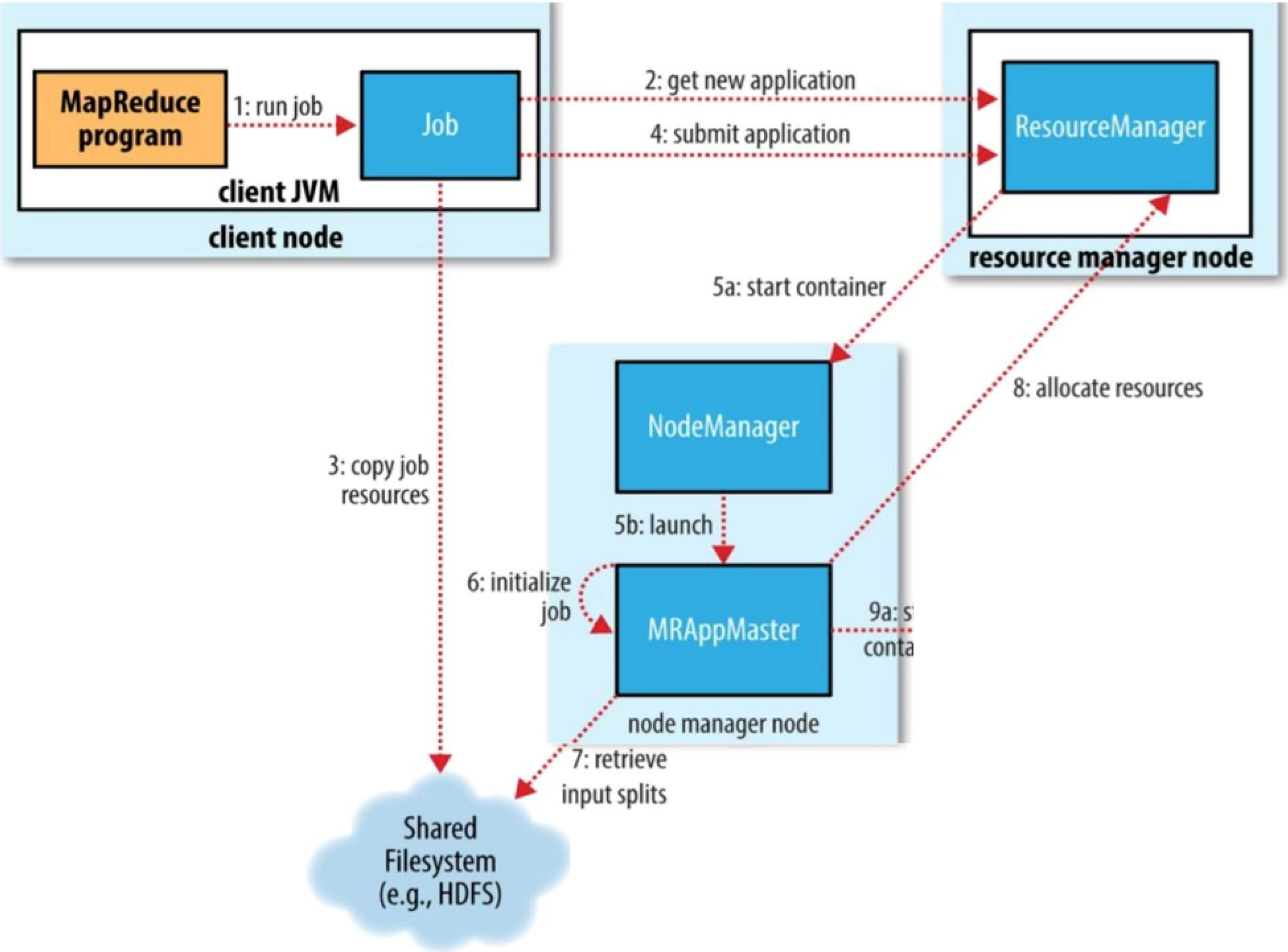
Job Submission

- The **job submission process** **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**.
  - **Check input** and **Computes** the **input splits** for the **job**.
  - **Copies** the **resources needed** to **run** the **job**, to the **shared filesystem** in a directory **named after the job ID** (**step 3**)
  - **Submits** the **job** by **calling submitApplication( )** on the **resource manager** (**step 4**).
  - **waitForCompletion( ) polls** the **job's progress**

Job Initialization

- **When** the **resource manager** **receives** a **call** to its **submitApplication( ) method**, it **hands off** the **request** to the **YARN scheduler**.

- The **YARN 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**).


MapReduce program
1: run job
Job
client JVM
client node
2: get new application
4: submit application
ResourceManager
resource manager node
5a: start container
NodeManager
8: allocate resources
3: copy job resources
5b: launch
6: initialize job
MRAppMaster
9a: s
conta
node manager node
7: retrieve input splits
Shared Filesystem (e.g., HDFS)

Job Initialization

- The **application master** for **MapReduce** jobs **initializes** the **job** by **creating** a **number of bookkeeping objects** to **keep track** of the **job's progress** **(step 6)**

- Next, it **retrieves** the **input splits** **computed** in the **client** from the **shared filesystem** (**step 7**).

Job Initialization

- It then **creates** a **map task object** for **each split**, as well as a **number** of **reduce task objects**.

- **Tasks** are **given IDs** at this point.

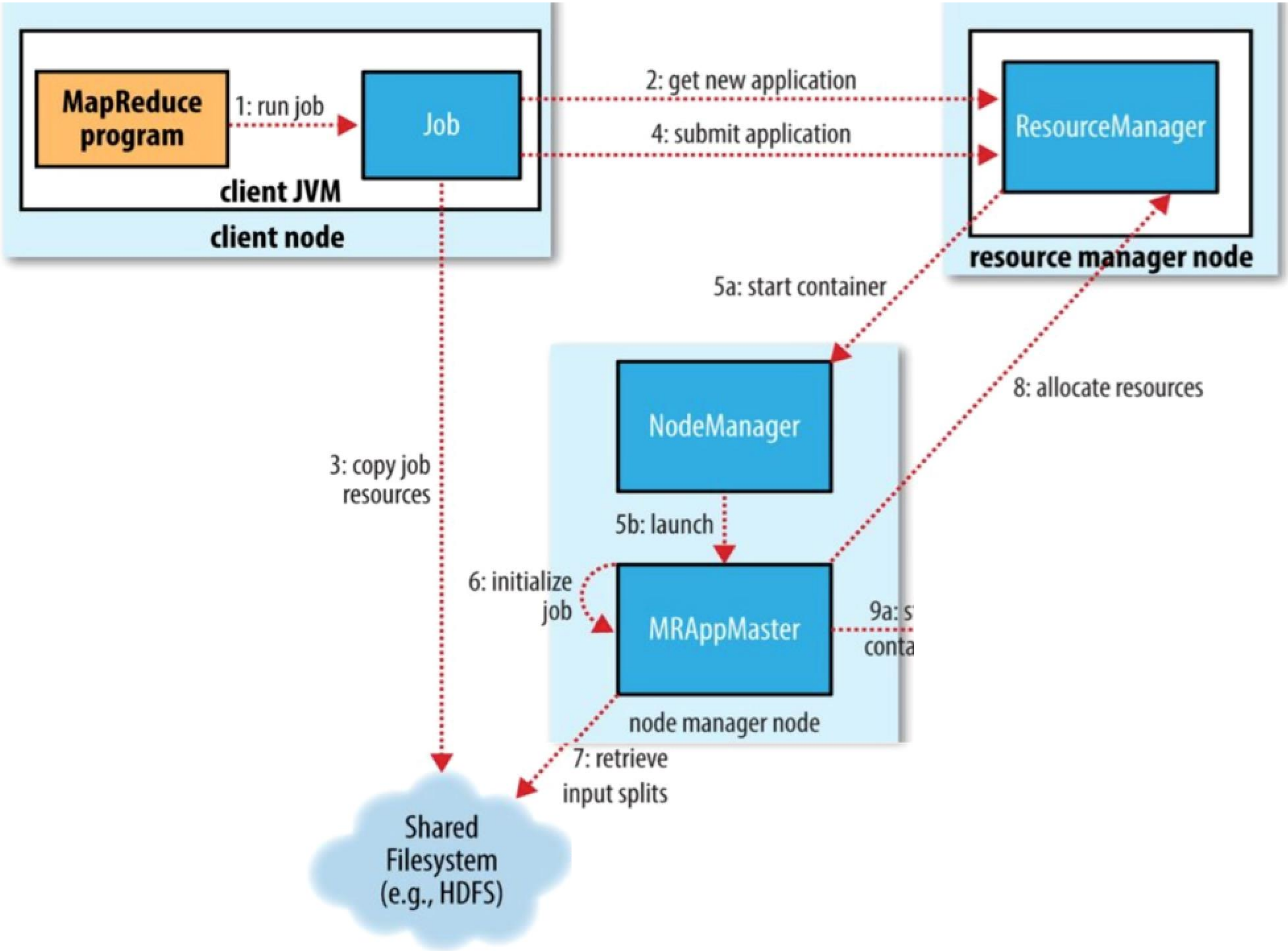
Job Initialization

- The application master must **decide** ***how to run the tasks*** that **make up** the **MapReduce job**.

- **If** the **job** is **small**, the **application master** may **choose** to **run** the **tasks** in the **same JVM** as **itself**.

- **Such** a **job** is **said** to be **uberized**, or run as an uber task.

- The **application master** will also **creates** the **final output directory** the **temporary space** for the **task output**.


MapReduce program
1: run job
Job
client JVM
client node
2: get new application
4: submit application
ResourceManager
resource manager node
3: copy job resources
5a: start container
NodeManager
8: allocate resources
5b: launch
6: initialize job
MRAppMaster
9a: s
conta
node manager node
7: retrieve input splits
Shared Filesystem (e.g., HDFS)

Task Assignment

- **If** the **job** **does not run as** an ***uber task***, then 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**.

- **Reduce tasks can run anywhere** in the **cluster**, but **requests** for **map tasks** have **data locality constraints** that the scheduler ***tries to honor***

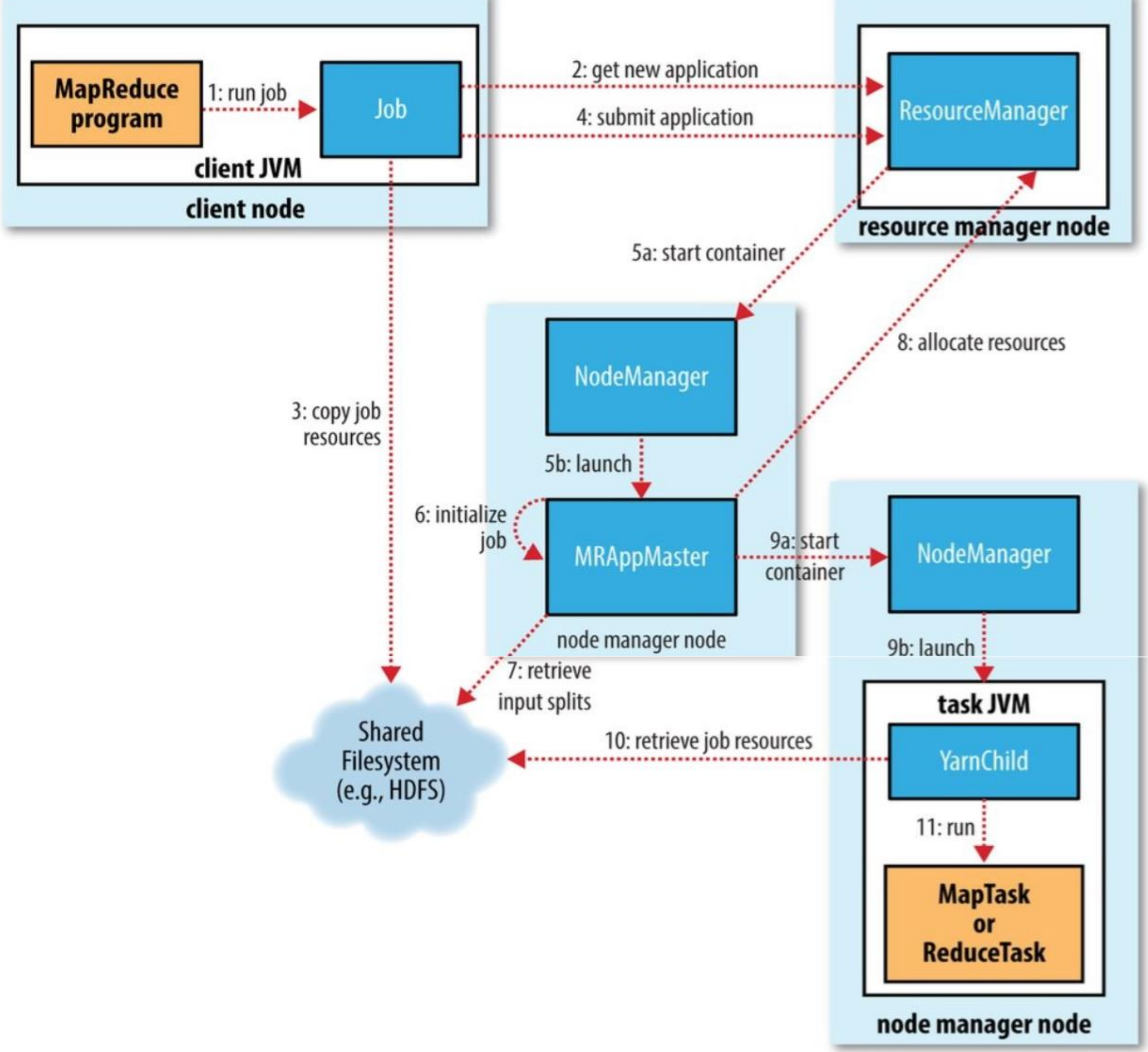
Task Execution

- On **resources assignment**, 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 **YarnChild**.

Task Execution

- 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***.


MapReduce program
1: run job
Job
client JVM
client node
2: get new application
4: submit application
ResourceManager
resource manager node
5a: start container
NodeManager
8: allocate resources
3: copy job resources
5b: launch
6: initialize job
MRAppMaster
9a: start container
NodeManager
node manager node
9b: launch
7: retrieve input splits
task JVM
Shared Filesystem (e.g., HDFS)
10: retrieve job resources
YarnChild
11: run
MapTask or ReduceTask
node manager node

Hadoop Installation

- **Standalone (or local) mode,** there are no daemons running and everything runs in a single process.
- **Pseudo-distributed mode**, Hadoop daemons run on the local machine, thus simulating a cluster on a small scale.
- **Fully distributed mode,** the Hadoop daemons run on a cluster of machines.

Hadoop Installation

- **Standalone (or local) mode,** there are no daemons running and everything runs in a single process.
- **Pseudo-distributed mode**, Hadoop daemons run on the local machine, thus simulating a cluster on a small scale.
- **Fully distributed mode,** the Hadoop daemons run on a cluster of machines.

YARN on a Single Node

- ***Assuming you have already configured HDFS, follow the steps …***

YARN on a Single Node

- **Step 1: Configure parameters as follows:**

etc/hadoop/mapred-site.xml:

```
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
    <property>
        <name>mapreduce.application.classpath</name>
<value>$HADOOP_MAPRED_HOME/share/hadoop/mapreduce/*:$HADOOP_MAPRED_HOME/share/hadoop/mapreduce/lib/*</value>
    </property>
</configuration>
```

https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/SingleCluster.html#YARN_on_a_Single_Node

YARN on a Single Node

- **Step 1: Configure parameters as follows:**

etc/hadoop/yarn-site.xml:

```
<configuration>
    <property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
    </property>
    <property>
        <name>yarn.nodemanager.env-whitelist</name>

<value>JAVA_HOME,HADOOP_COMMON_HOME,HADOOP_HDFS_HOME,HADOOP_CONF_
DIR,CLASSPATH_PREPEND_DISTCACHE,HADOOP_YARN_HOME,HADOOP_MAPRED_HO
ME</value>
    </property>
</configuration>
```

YARN on a Single Node

- **Step 2**: **Start ResourceManager daemon and NodeManager daemon:**

```
$ start-yarn.sh
```

Browse the web interface *for the* ***Re****sourceManager; by default it is available at:* ***http://localhost:8088/***

Any Questions ?