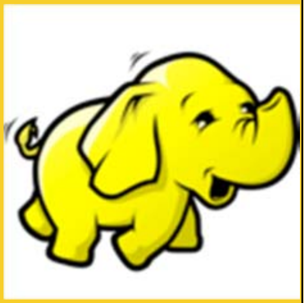
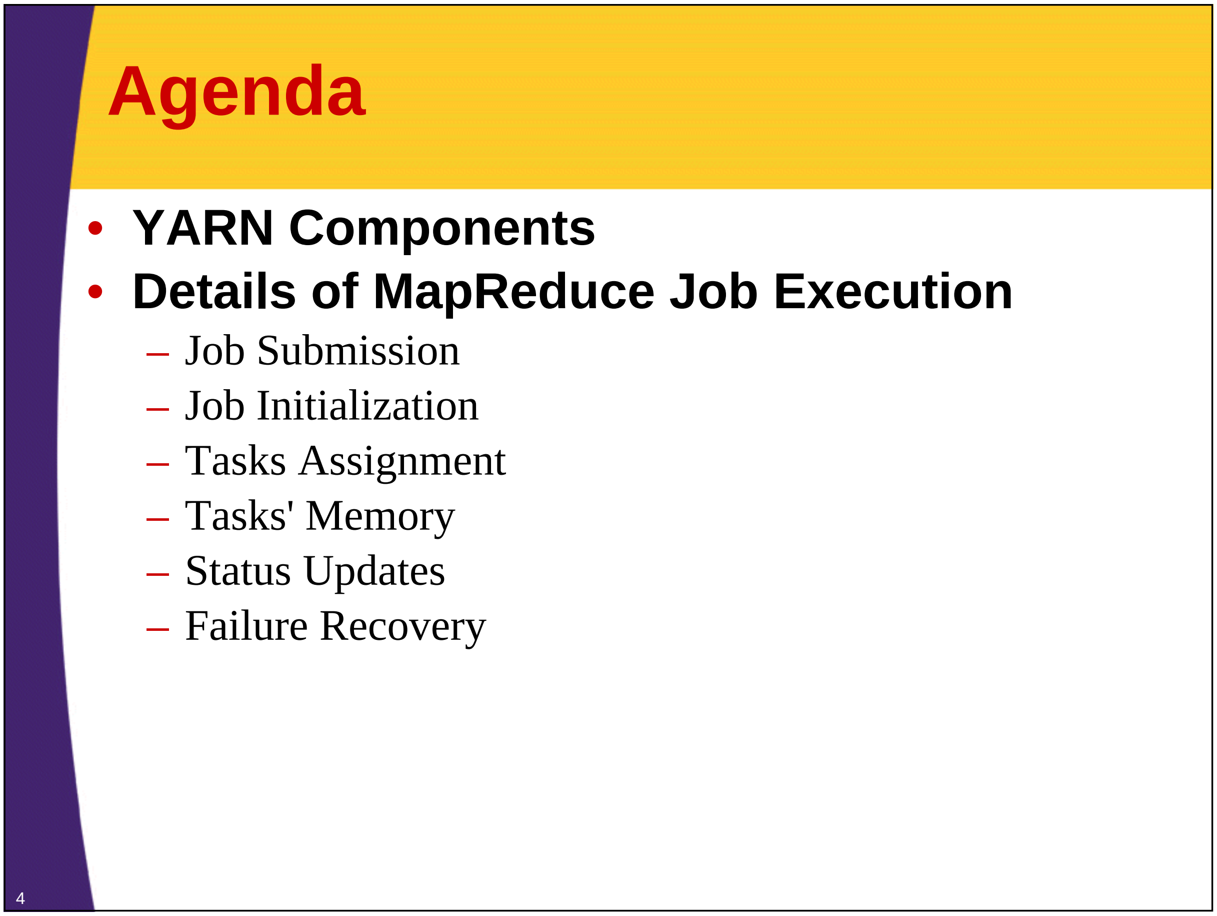
MapReduce on YARN Job Execution

Originals of Slides and Source Code for Examples:
http://www.coreservlets.com/hadoop-tutorial/

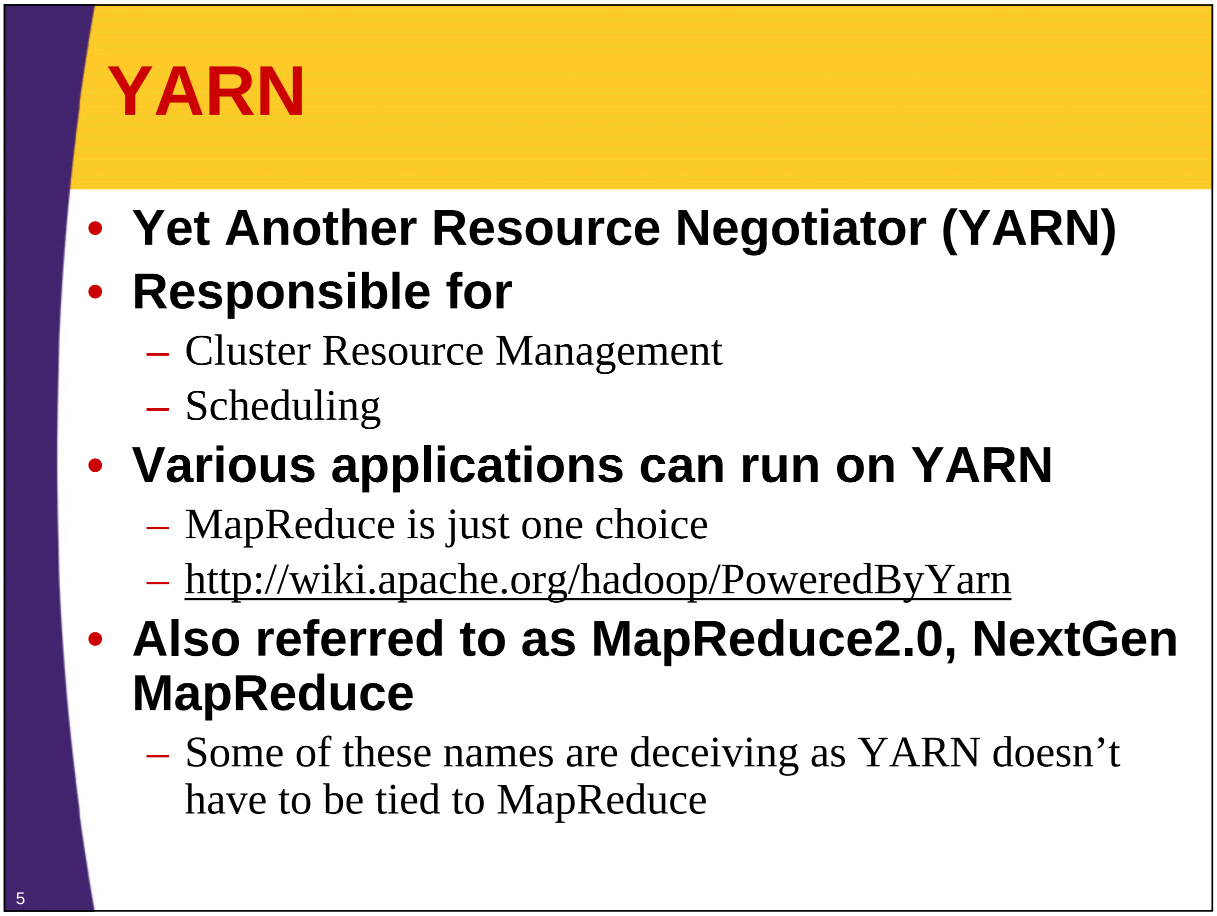
more SERVLETS and JAVASERVER PAGES
MARTY HALL
core SERVLETS and JAVASERVER PAGES
2ND EDITION
Volume 1: Core Technologies
MARTY HALL · LARRY BROWN





Agenda

- **YARN Components**
- **Details of MapReduce Job Execution**
  - Job Submission
  - Job Initialization
  - Tasks Assignment
  - Tasks' Memory
  - Status Updates
  - Failure Recovery



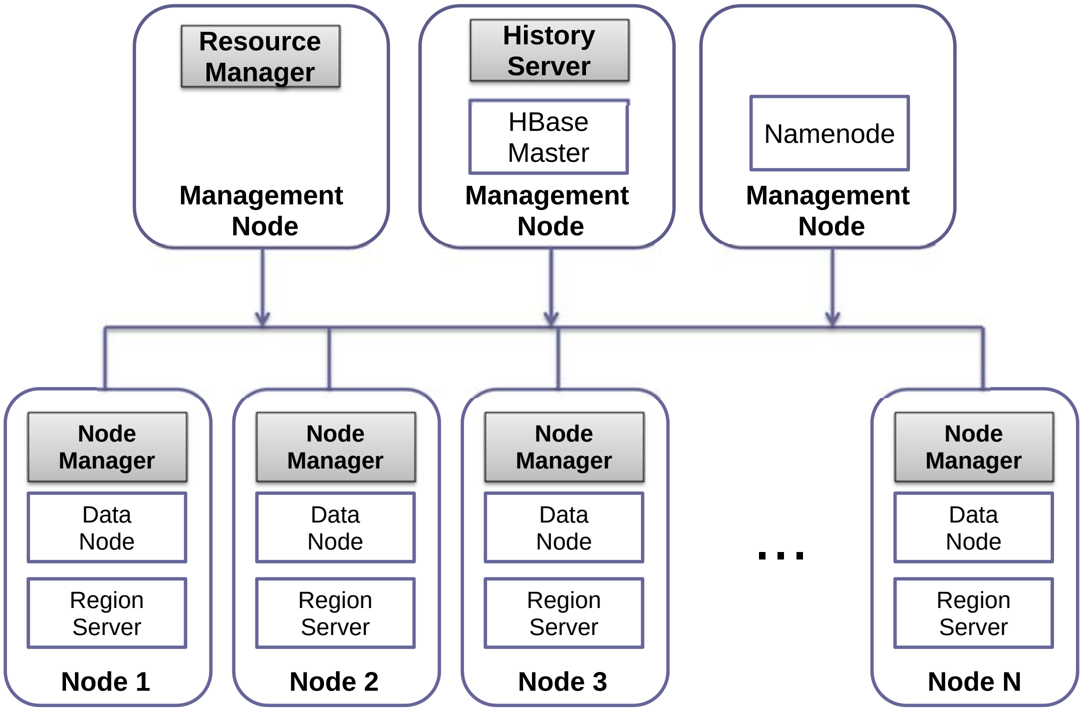
YARN

- **Yet Another Resource Negotiator (YARN)**
- **Responsible for**
  - Cluster Resource Management
  - Scheduling
- **Various applications can run on YARN**
  - MapReduce is just one choice
  - http://wiki.apache.org/hadoop/PoweredByYarn
- **Also referred to as MapReduce2.0, NextGen MapReduce**
  - Some of these names are deceiving as YARN doesn’t have to be tied to MapReduce

YARN vs. Old MapReduce

- **Prior to YARN Hadoop had JobTracker and TaskTracker daemons**
  - JobTracker is responsible for handling resources and tasks' progress monitoring/management
    - Dealing with failed tasks
    - Task Bookkeeping
- **JobTracker based approach had drawbacks**
  - Scalability Bottleneck – 4,000+ nodes
  - Cluster Resource sharing and allocation flexibility
    - Slot based approach (ex. 10 slots per machine no matter how small or big those tasks are)
- **In 2010 Yahoo! started designing next generation MapReduce => YARN**

Sample YARN Daemons Deployments with HDFS and HBase



Resource Manager

Management Node

History Server

HBase Master

Management Node

Namenode

Management Node

Node Manager

Data Node

Region Server

Node 1

Node Manager

Data Node

Region Server

Node 2

Node Manager

Data Node

Region Server

Node 3

…

Node Manager

Data Node

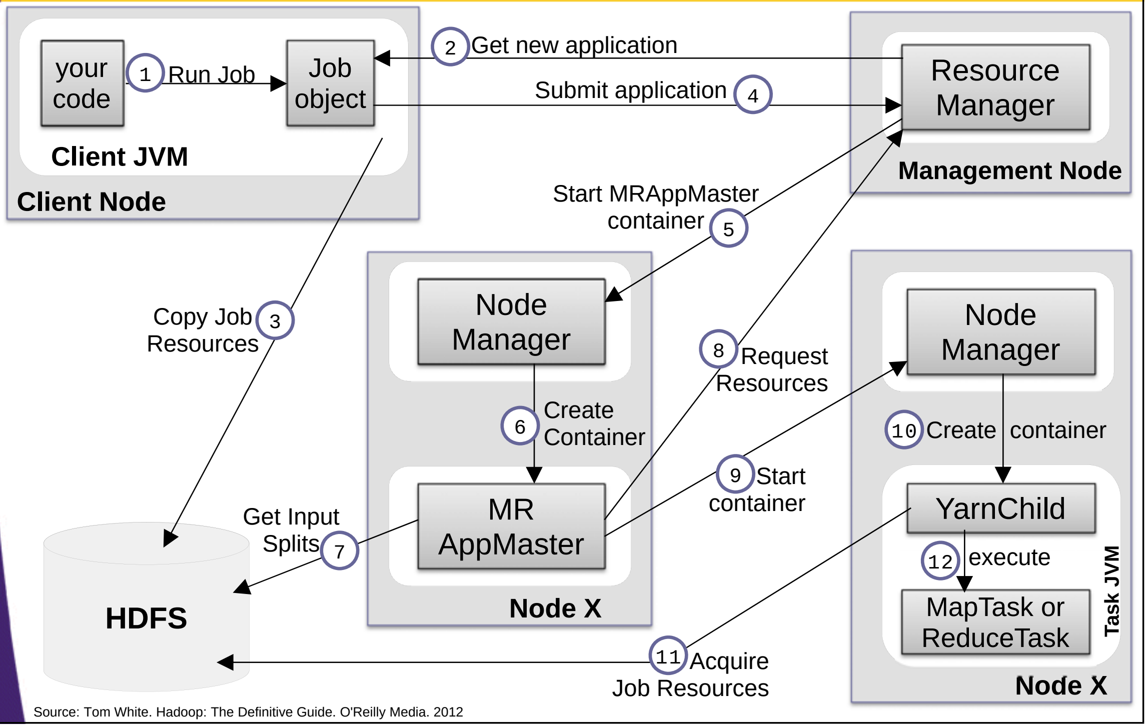
Region Server

Node N

MapReduce on YARN Components

- **Client – submits MapReduce Job**
- **Resource Manager – controls the use of resources across the Hadoop cluster**
- **Node Manager – runs on each node in the cluster; creates execution container, monitors container's usage**
- **MapReduce Application Master – Coordinates and manages MapReduce Jobs; negotiates with Resource Manager to schedule tasks; the tasks are started by NodeManager(s)**
- **HDFS – shares resources and jobs' artifacts between YARN components**

MapReduce Job Execution on YARN



your code

1 Run Job

Job object

Client JVM

Client Node

2 Get new application

Submit application 4

Resource Manager

Management Node

Start MRAppMaster container 5

Copy Job Resources 3

Node Manager

6 Create Container

MR AppMaster

Node X

8 Request Resources

9 Start container

Node Manager

10 Create container

YarnChild

12 execute

MapTask or ReduceTask

Task JVM

Node X

Get Input Splits 7

HDFS

11 Acquire Job Resources

Source: Tom White. Hadoop: The Definitive Guide. O'Reilly Media. 2012

MapReduce on YARN Job Execution

1. Client submits MapReduce job by interacting with Job objects; Client runs in it's own JVM
2. Job's code interacts with Resource Manager to acquire application meta-data, such as application id
3. Job's code moves all the job related resources to HDFS to make them available for the rest of the job
4. Job's code submits the application to Resource Manager
5. Resource Manager chooses a Node Manager with available resources and requests a container for MRAppMaster
6. Node Manager allocates container for MRAppMaster; MRAppMaster will execute and coordinate MapReduce job
7. MRAppMaster grabs required resource from HDFS, such as Input Splits; these resources were copied there in step 3

MapReduce Job Execution on YARN

8. MRAppMaster negotiates with Resource Manager for available resources; Resource Manager will select Node Manager that has the most resources
9. MRAppMaster tells selected NodeManager to start Map and Reduce tasks
10. NodeManager creates YarnChild containers that will coordinate and run tasks
11. YarnChild acquires job resources from HDFS that will be required to execute Map and Reduce tasks
12. YarnChild executes Map and Reduce tasks

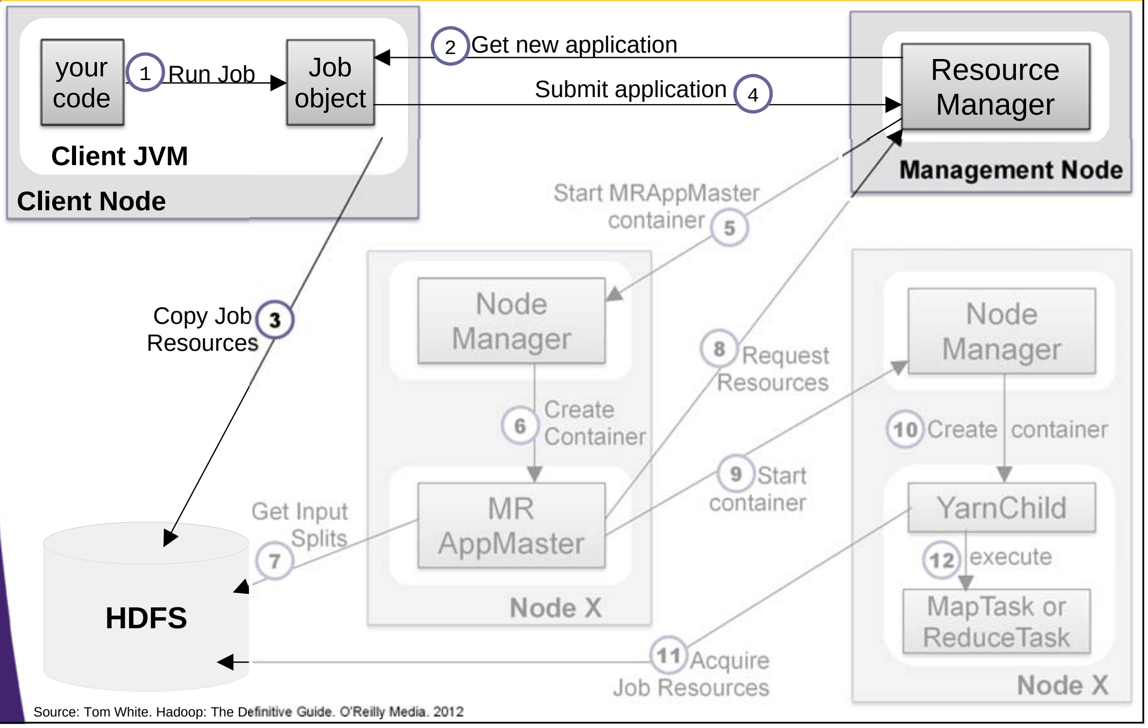
MapReduce Job Submission

- Use org.apache.hadoop.mapreduce.Job class to configure the job
- Submit the job to the cluster and wait for it to finish.
  - job.waitForCompletion(true)
- The YARN protocol is activated when mapreduce.framework.name property in mapred-site.xml is set to yarn
- Client code in client JVM

MapReduce Job Submission Steps

1. Application Id is retrieved from Resource Manager
2. Job Client verifies output specification of the job
   - Delegates to OutputFormat - you may have seen annoying messages that output directory already exists
3. Computes Input Splits
   - Can optionally generate in the cluster – good use case for jobs with many splits (yarn.app.mapreduce.am.compute-splits-in-cluster property)
4. Copy Job Resources to HDFS
   - Jar files, configurations, input splits
5. Job Submission

MapReduce Job Submission Components



your code

1 Run Job

Job object

Client JVM

Client Node

2 Get new application

Submit application 4

Resource Manager

Management Node

Start MRAppMaster container 5

Copy Job Resources 3

Node Manager

6 Create Container

MR AppMaster

Node X

8 Request Resources

9 Start container

Get Input Splits 7

HDFS

Node Manager

10 Create container

YarnChild

12 execute

MapTask or ReduceTask

Node X

11 Acquire Job Resources

Source: Tom White. Hadoop: The Definitive Guide. O'Reilly Media. 2012

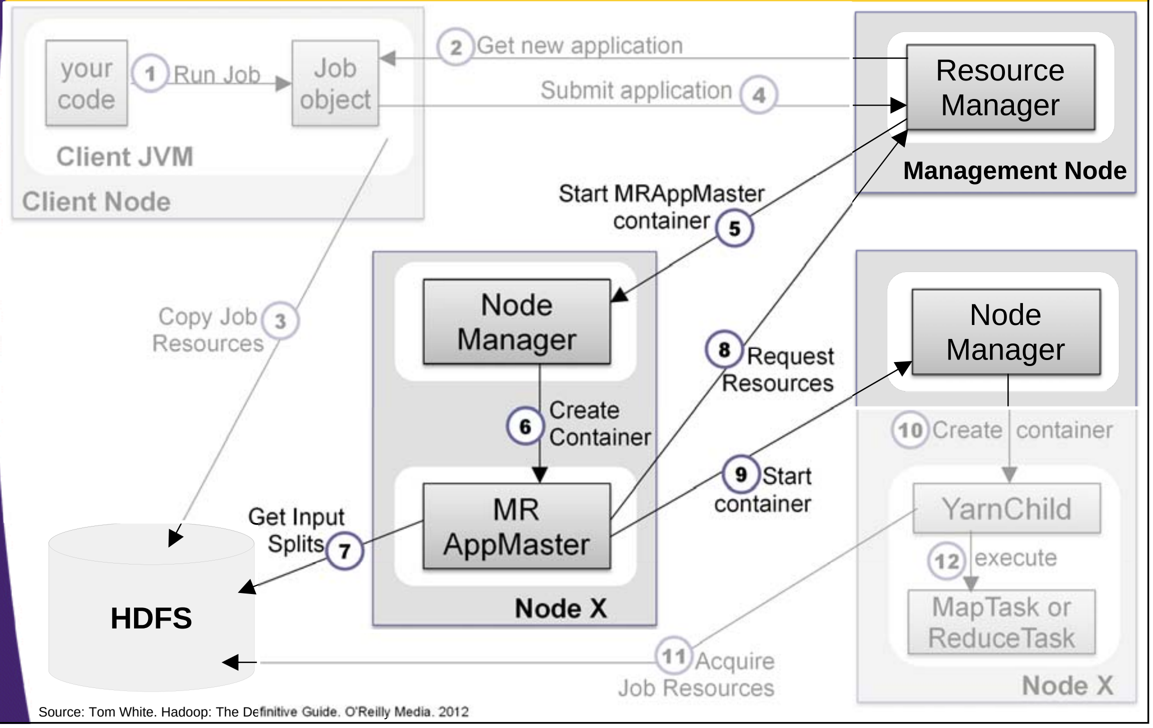
Job Initialization Steps

1. **Resource Manager receives request for a new application**
2. **Resource Manager delegates to its internal component – Scheduler**
   – There are various schedulers available
3. **Scheduler requests a container for an Application Master process**
   – MapReduce’s Application Master is MRAppMaster
4. **MRAppMaster initializes its internal objects and executes a job by negotiating with Resource Manager**

MRAppMaster Initialization Steps

1. **Creates internal bookkeeping objects to monitor progress**
2. **Retrieves Input Splits**
   - These were created by the client and copied onto HDFS
3. **Creates tasks**
   - Map tasks per split
   - Reduce tasks based on mapreduce.job.reduces property
4. **Decides how to run the tasks**
   - In case of a small job, it will run all tasks in MRAppMaster's JVM; this job is called "uberized" or "uber"
   - Execute tasks on Node Manager
5. **Execute the tasks**

MapReduce Job Initialization Components



Source: Tom White. Hadoop: The Definitive Guide. O'Reilly Media. 2012

MRAppMaster and Uber Job

- **If a job is too small the MRAppMaster will execute map and reduce tasks within the same JVM**
  - The idea that distributed task allocation and management overhead exceeds the benefits of executing tasks in parallel
- **Will Uberize if all of these conditions are met:**
  1. Less than 10 mappers (mapreduce.job.ubertask.maxmaps property)
  2. A single Reducer (mapreduce.job.ubertask.maxreduces property)
  3. Input size less than 1 HDFS block (mapreduce.job.ubertask.maxbytes property)
- **Execution of Jobs as Uber can be disabled**
  - Set mapreduce.job.ubertask.enable property to false

Task Assignment

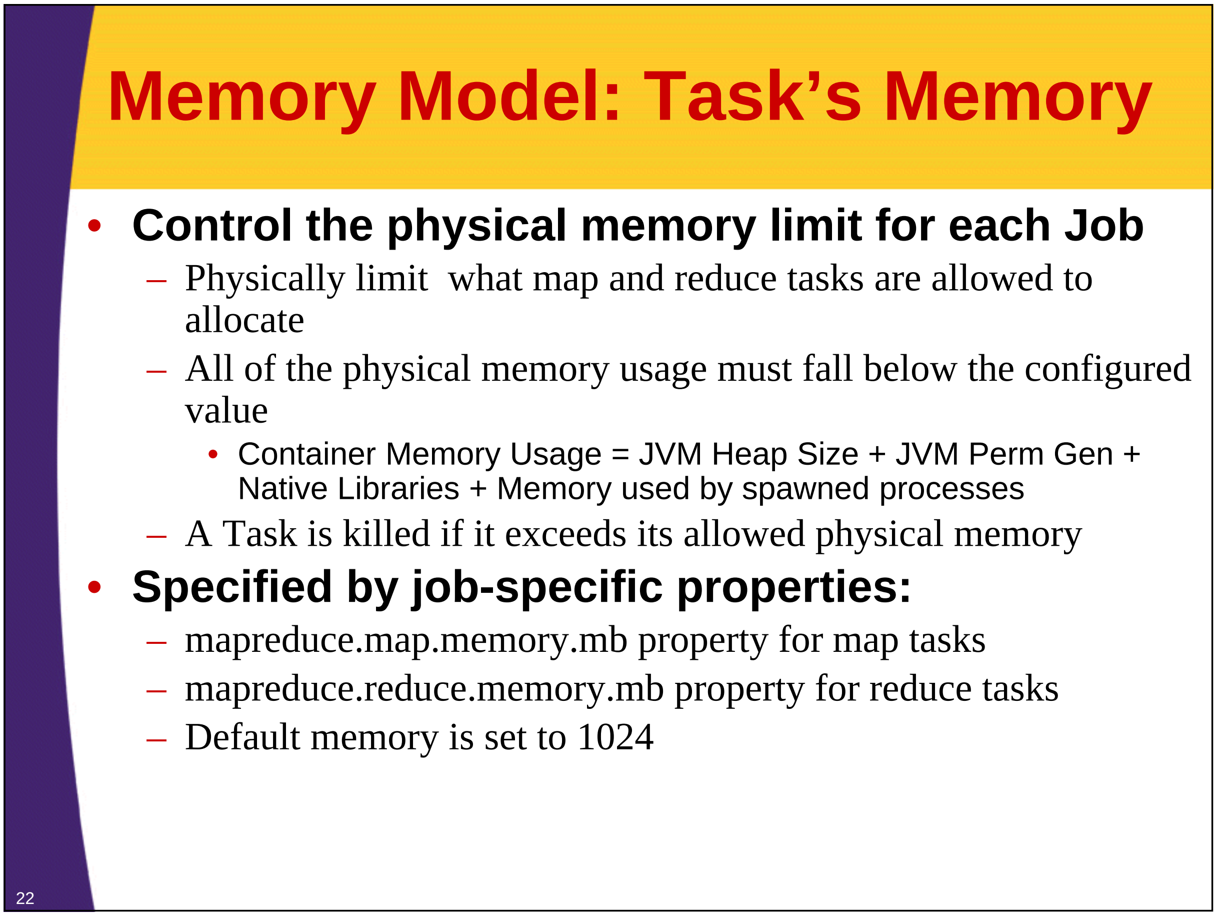
- **Only applicable to non-Uber Jobs**
- **MRAppMaster negotiates container for map and reduce tasks from Resource Manager, request carry:**
  - data locality information (hosts & racks) which was computed by InputFormat and stored inside InputSplits
  - Memory requirements for a task
- **Scheduler on Resource Manager utilizes provided information to make a decision on where to place these tasks**
  - Attempts locality: Ideally placing tasks on the same nodes where the data to process resides. Plan B is to place within the same rack

Fine-Grained Memory Model for Tasks

- **In YARN, administrators and developers have a lot of control over memory management**
  - NodeManager
    - Typically there is one NodeManager per machine
  - Task Containers that run Map and Reduce tasks
    - Multiple tasks can execute on a single NodeManager
  - Scheduler
    - Minimum and maximum allocations controls
  - JVM Heap
    - Allocate memory for your code
  - Virtual Memory
    - Prevent tasks from monopolizing machines

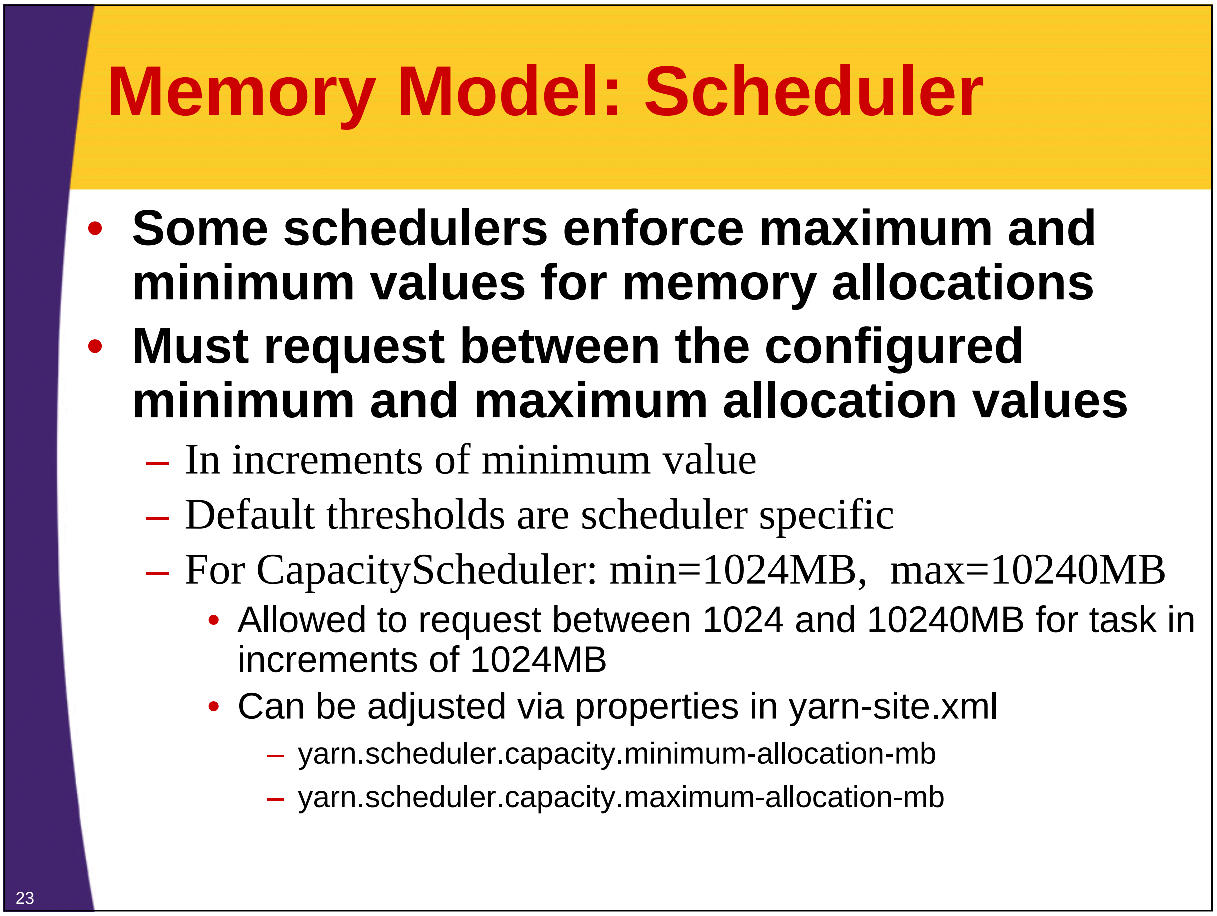
Memory Model: Node Manager

- **Node Manager allocates containers that run Map and Reduce tasks**
  - The sum of all the containers can not exceed configured threshold
  - Node Manager will not allocate a container if there isn’t enough available memory
- **The memory allocation limit is configurable per Node Manager**
  - Set yarn.nodemanager.resource.memory-mb property in yarn-default.xml
    - Default is 8,192MB
  - Configured once at start-up



Memory Model: Task's Memory

- **Control the physical memory limit for each Job**
  - Physically limit what map and reduce tasks are allowed to allocate
  - All of the physical memory usage must fall below the configured value
    - Container Memory Usage = JVM Heap Size + JVM Perm Gen + Native Libraries + Memory used by spawned processes
  - A Task is killed if it exceeds its allowed physical memory
- **Specified by job-specific properties:**
  - mapreduce.map.memory.mb property for map tasks
  - mapreduce.reduce.memory.mb property for reduce tasks
  - Default memory is set to 1024



Memory Model: Scheduler

- **Some schedulers enforce maximum and minimum values for memory allocations**
- **Must request between the configured minimum and maximum allocation values**
  - In increments of minimum value
  - Default thresholds are scheduler specific
  - For CapacityScheduler: min=1024MB, max=10240MB
    - Allowed to request between 1024 and 10240MB for task in increments of 1024MB
    - Can be adjusted via properties in yarn-site.xml
      - yarn.scheduler.capacity.minimum-allocation-mb
      - yarn.scheduler.capacity.maximum-allocation-mb

Memory Model: JVM Heap

- **Recall that mapreduce.map.memory.mb and mapreduce.reduce.memory.mb properties set the limit for map and reduce containers**

Container's Memory Usage = JVM Heap Size + JVM Perm Gen + Native Libraries + Memory used by spawned processes

- **JVM Heap size can be specified by job specific properties:**
  - mapreduce.reduce.java.opts
  - mapreduce.map.java.opts

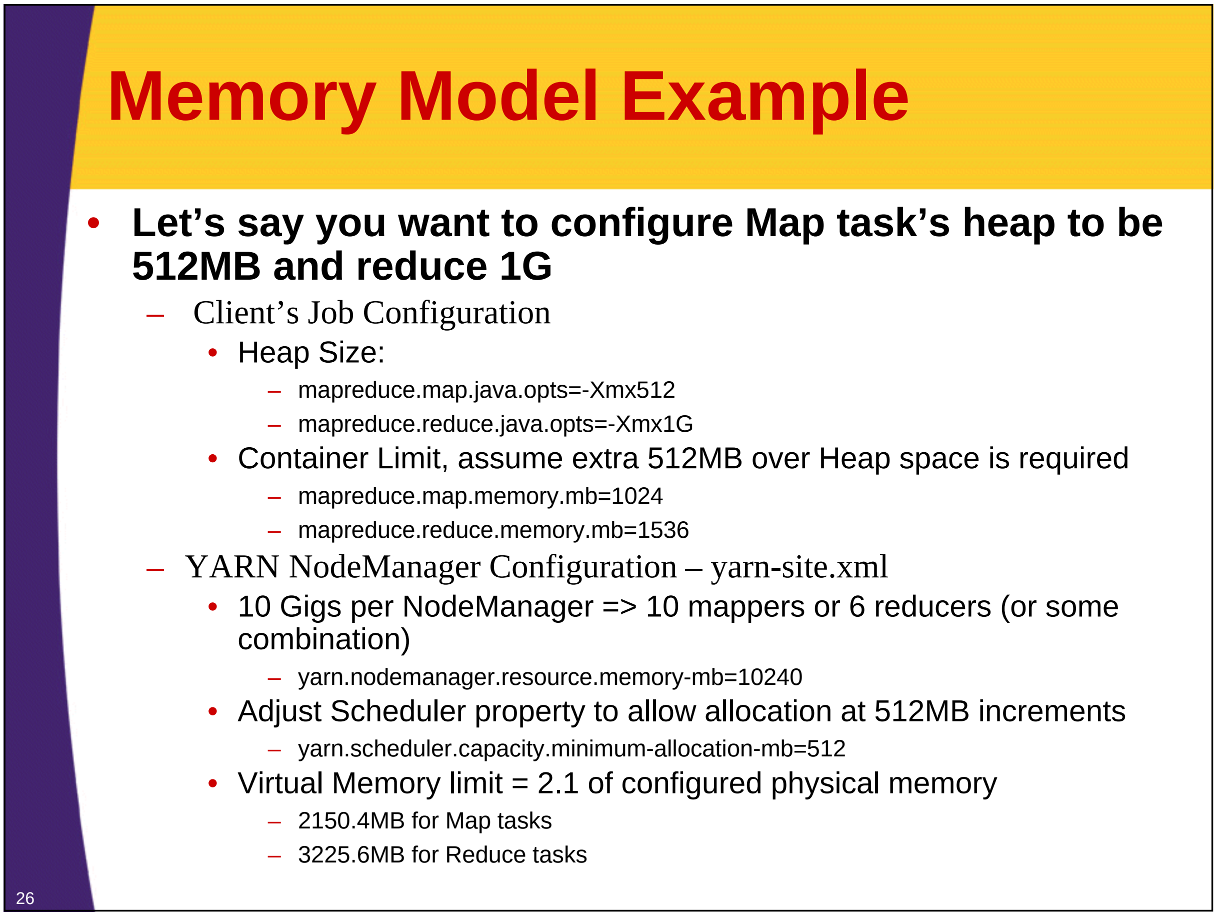
```
Example: mapreduce.map.java.opts=-Xmx2G
```

Memory Model: Virtual Memory

- **From `top` command documentation, Virtual Memory**

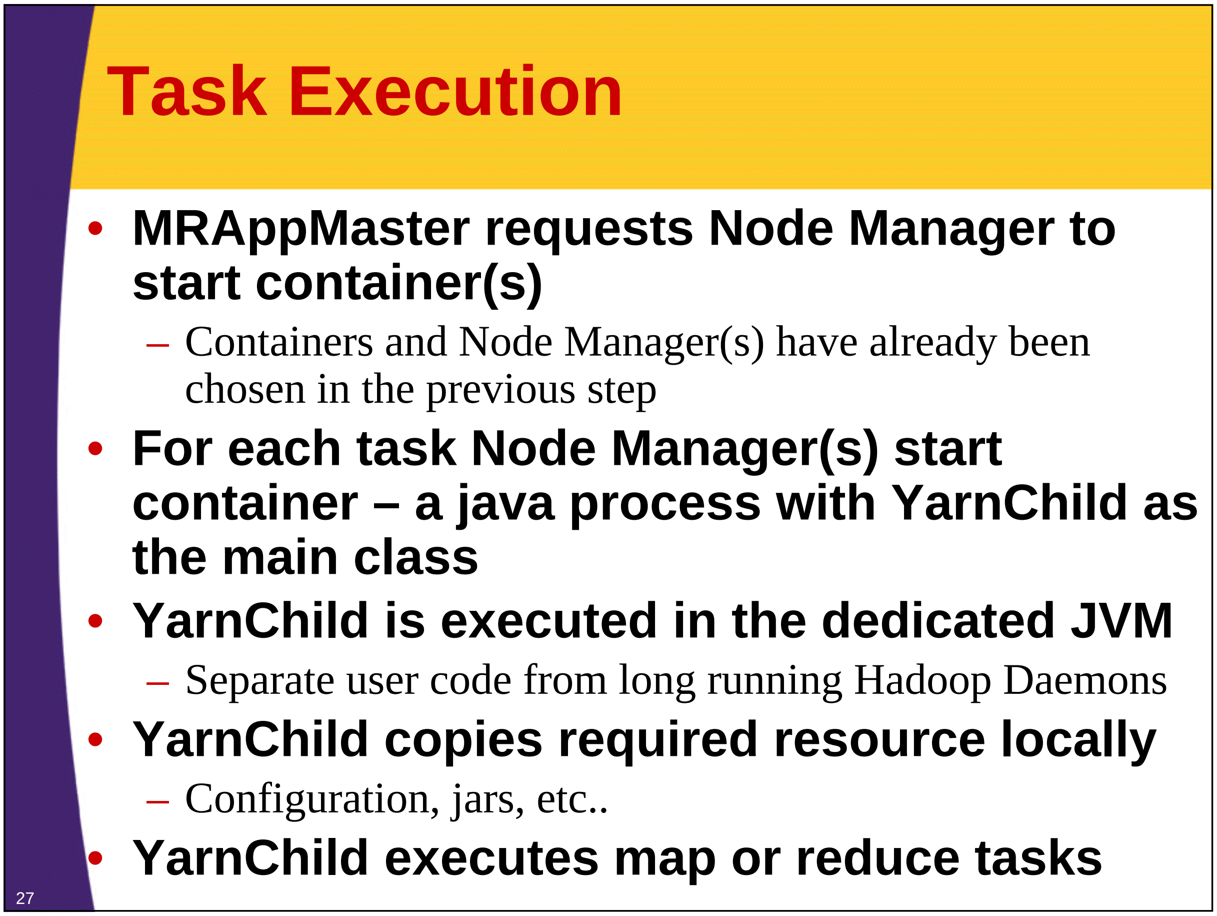
  "includes all code, data and shared libraries plus pages that have been swapped out"

- **Virtual Memory allocation is limited by Node Manager**
  - A Task is killed if it exceeds it's allowed Virtual Memory
- **Configured in multiples of physical memory**
  - Be default, it's 2.1 of container's physical memory
    - Ex: If a container is configured to 1G of physical memory then it will not be able to exceed 2.1G of Virtual Memory
  - Adjust via yarn.nodemanager.vmem-pmem-ratio property in yarn-site.xml
  - Configured once at start-up



Memory Model Example

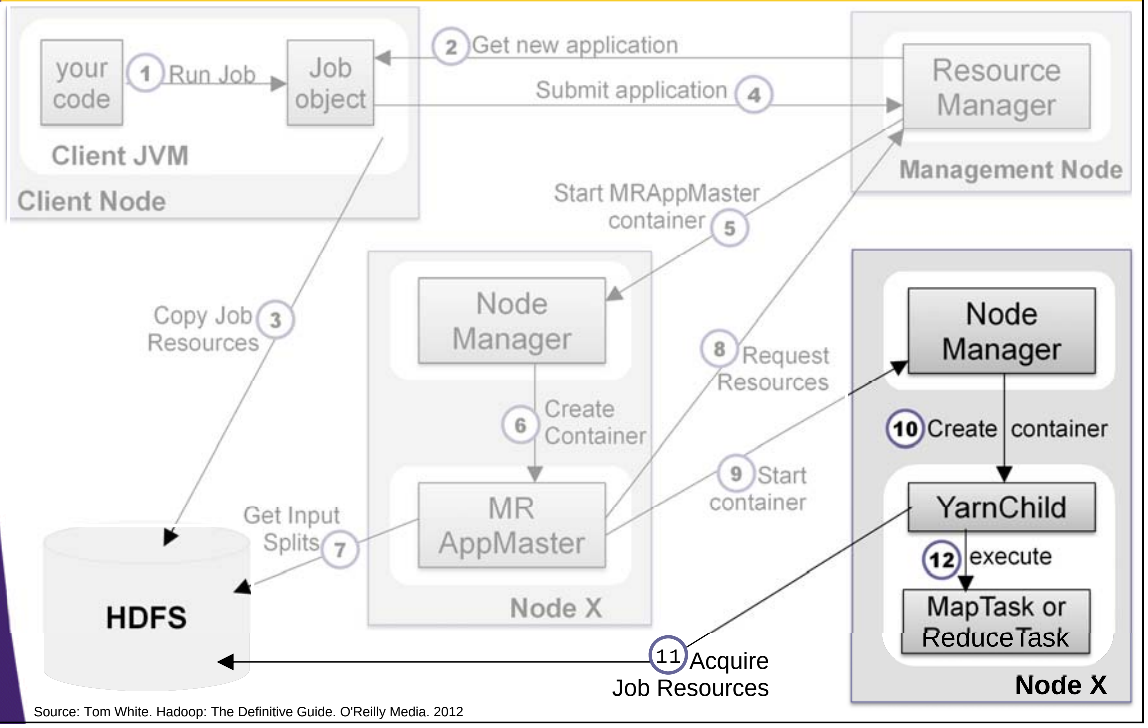
- **Let's say you want to configure Map task's heap to be 512MB and reduce 1G**
  - Client's Job Configuration
    - Heap Size:
      - mapreduce.map.java.opts=-Xmx512
      - mapreduce.reduce.java.opts=-Xmx1G
    - Container Limit, assume extra 512MB over Heap space is required
      - mapreduce.map.memory.mb=1024
      - mapreduce.reduce.memory.mb=1536
  - YARN NodeManager Configuration – yarn-site.xml
    - 10 Gigs per NodeManager => 10 mappers or 6 reducers (or some combination)
      - yarn.nodemanager.resource.memory-mb=10240
    - Adjust Scheduler property to allow allocation at 512MB increments
      - yarn.scheduler.capacity.minimum-allocation-mb=512
    - Virtual Memory limit = 2.1 of configured physical memory
      - 2150.4MB for Map tasks
      - 3225.6MB for Reduce tasks



Task Execution

- **MRAppMaster requests Node Manager to start container(s)**
  - Containers and Node Manager(s) have already been chosen in the previous step
- **For each task Node Manager(s) start container – a java process with YarnChild as the main class**
- **YarnChild is executed in the dedicated JVM**
  - Separate user code from long running Hadoop Daemons
- **YarnChild copies required resource locally**
  - Configuration, jars, etc..
- **YarnChild executes map or reduce tasks**

MapReduce Task Execution Components

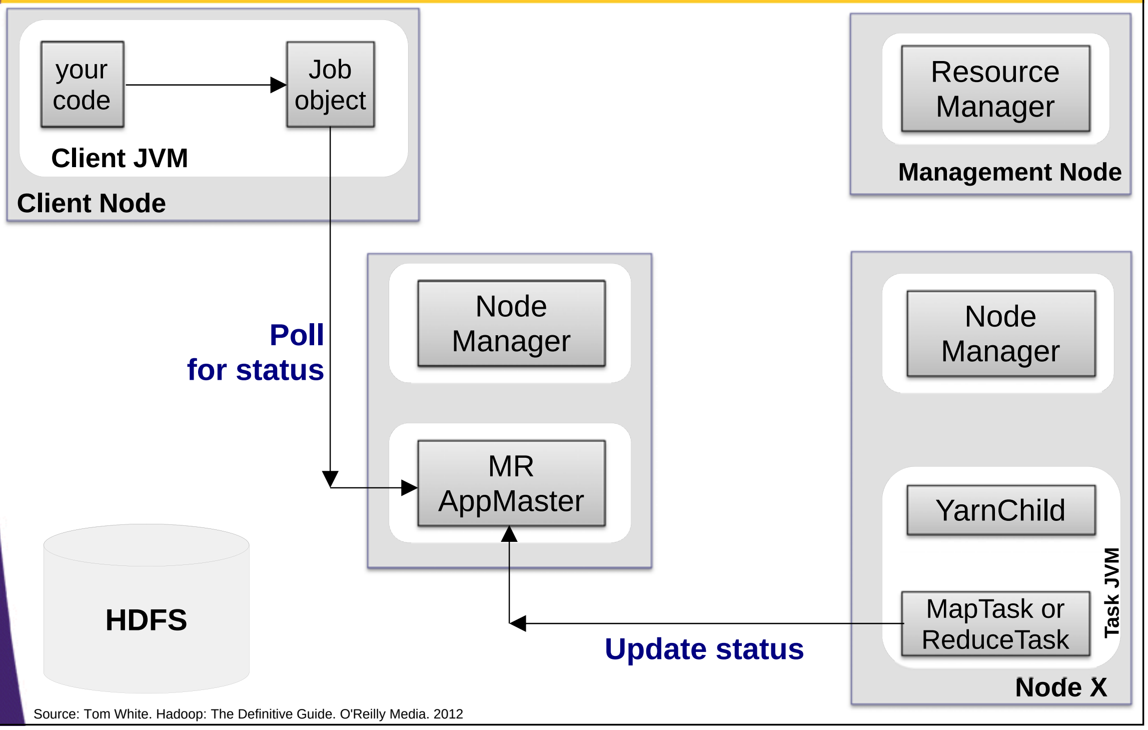


Source: Tom White. Hadoop: The Definitive Guide. O'Reilly Media. 2012

Status Updates

- **Tasks report status to MRAppMaster**
  - Maintain umbilical interface
  - Poll every 3 seconds
- **MRAppMaster accumulates and aggregates the information to assemble current status of the job**
  - Determines if the job is done
- **Client (Job object) polls MRAppMaster for status updates**
  - Every second by default, Configure via property mapreduce.client.progressmonitor.pollinterval
- **Resource Manager Web UI displays all the running YARN applications where each one is a link to Web UI of Application Master**
  - In this case MRAppMaster Web UI

MapReduce Status Updates



your code

Job object

Client JVM

Client Node

Resource Manager

Management Node

Poll for status

Node Manager

MR AppMaster

HDFS

Node Manager

YarnChild

MapTask or ReduceTask

Task JVM

Update status

Node X

Source: Tom White. Hadoop: The Definitive Guide. O'Reilly Media. 2012

Failures

- **Failures can occur in**
  - Tasks
  - Application Master – MRAppMaster
  - Node Manager
  - Resource Manager

Task Failures

- **Most likely offender and easiest to handle**
- **Task's exceptions and JVM crashes are propagated to MRAppMaster**
  - Attempt (not a task) is marked as 'failed'
- **Hanging tasks will be noticed, killed**
  - Attempt is marked as failed
  - Control via mapreduce.task.timeout property
- **Task is considered to be failed after 4 attempts**
  - Set for map tasks via mapreduce.map.maxattempts
  - Set for reduce tasks via mapreduce.reduce.maxattempts

Application Master Failures - MRAppMaster

- **MRAppMaster Application can be re-tried**
  - By default will not re-try and will fail after a single application failure
- **Enable re-try by increasing yarn.resourcemanager.am.max-retries property**
- **Resource Manager recieves heartbeats from MRAppMaster and can restart in case of failure(s)**
- **Restarted MRAppMaster can recover latest state of the tasks**
  - Completed tasks will not need to be re-run
  - To enable set yarn.app.mapreduce.am.job.recovery.enable property to true

Node Manager Failure

- **Failed Node Manager will not send heartbeat messages to Resource Manager**
- **Resource Manager will black list a Node Manager that hasn't reported within 10 minutes**
  - Configure via property:
    - yarn.resourcemanager.nm.liveness-monitor.expiry-interval-ms
  - Usually there is no need to change this setting
- **Tasks on a failed Node Manager are recovered and placed on healthy Node Managers**

Node Manager Blacklisting by MRAppMaster

- **MRAppMaster may blacklist Node Managers if the number of failures is high on that node**
- **MRAppMaster will attempt to reschedule tasks on a blacklisted Node Manager onto Healthy Nodes**
- **Blacklisting is per Application/Job therefore doesn't affect other Jobs**
- **By default blacklisting happens after 3 failures on the same node**
  - Adjust default via mapreduce.job.maxtaskfailures.per.tracker

Resource Manager Failures

- **The most serious failure = downtime**
  - Jobs or Tasks can not be launched
- **Resource Manager was designed to automatically recover**
  - Incomplete implementation at this point
  - Saves state into persistent store by configuring yarn.resourcemanager.store.class property
  - The only stable option for now is in-memory store
    - org.apache.hadoop.yarn.server.resourcemanager.recovery.MemStore
  - Zookeeper based implementation is coming
    - You can track progress via https://issues.apache.org/jira/browse/MAPREDUCE-4345

Job Scheduling

- **By default FIFO scheduler is used**
  - First In First Out
  - Supports basic priority model: VERY_LOW, LOW, NORMAL, HIGH, and VERY_HIGH
  - Two ways to specify priority
    - mapreduce.job.priority property
    - job.setPriority(JobPriority.HIGH)
- **Specify scheduler via yarn.resourcemanager.scheduler.class property**
  - CapacityScheduler
  - FairScheduler

Job Completion

- **After MapReduce application completes (or fails)**
  - MRAppMaster and YARN Child JVMs are shut down
  - Management and metrics information is sent from MRAppMaster to the MapReduce History Server
- **History server has a similar Web UI to YARN Resource Manager and MRAppMaster**
  - By default runs on port 19888
    - http://localhost:19888/jobhistory
  - Resource Manager UI auto-magically proxies to the proper location, MRAppMaster if an application is running and History Server after application's completion
    - May get odd behavior (blank pages) if you access an application as it's moving its management from MRAppMaster to History Server

Wrap-Up

Summary

- **We learned about**
  - YARN Components
- **We discussed MapReduce Job Execution**
  - Job Submission
  - Job Initialization
  - Tasks Assignment
  - Tasks' Memory
  - Status Updates
  - Failure Recovery

Questions?