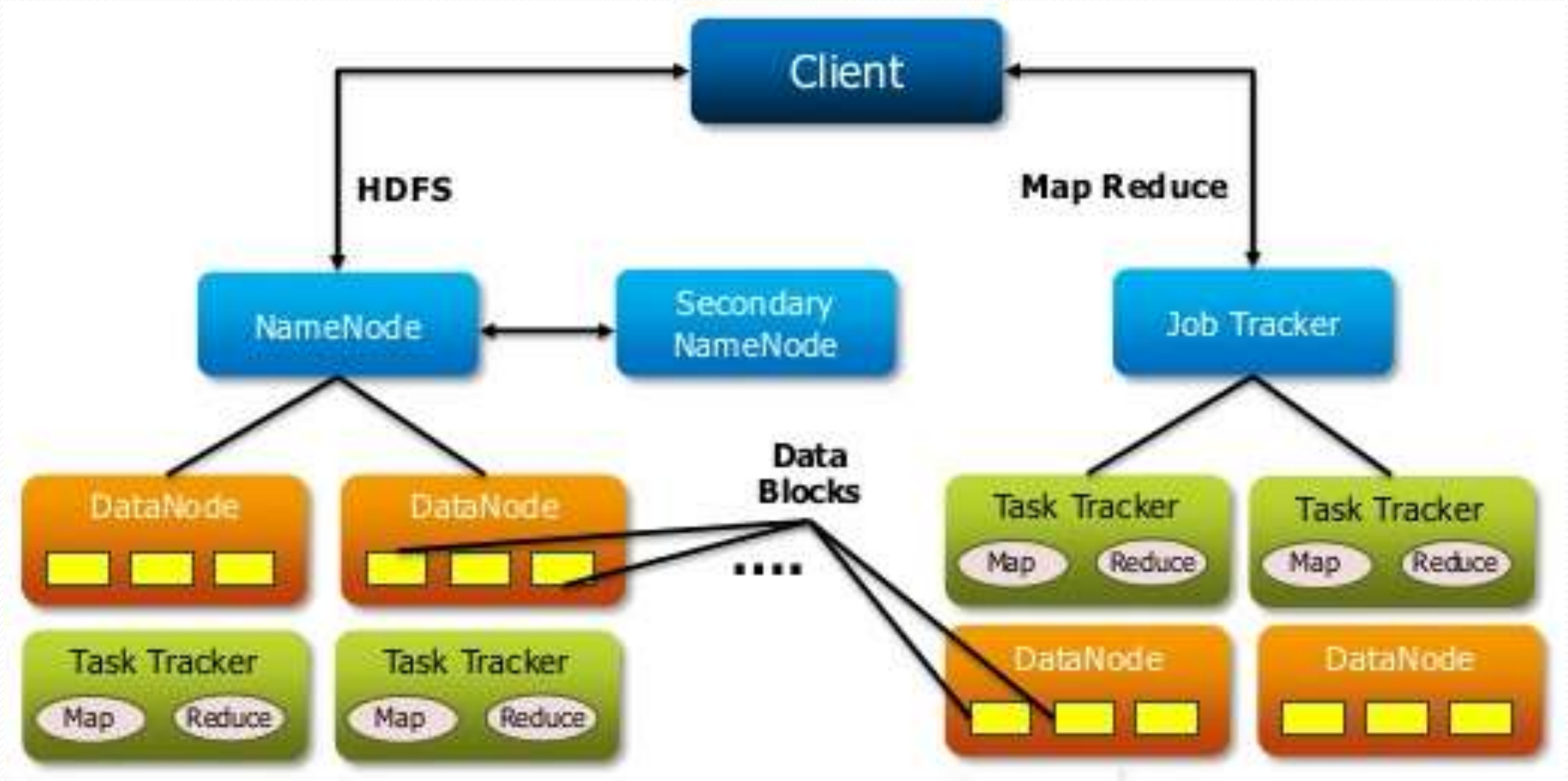
CS523 - BDT
Big Data Technologies

HDFS in depth & YARN
(Description of Hadoop daemons and need for Hadoop2)

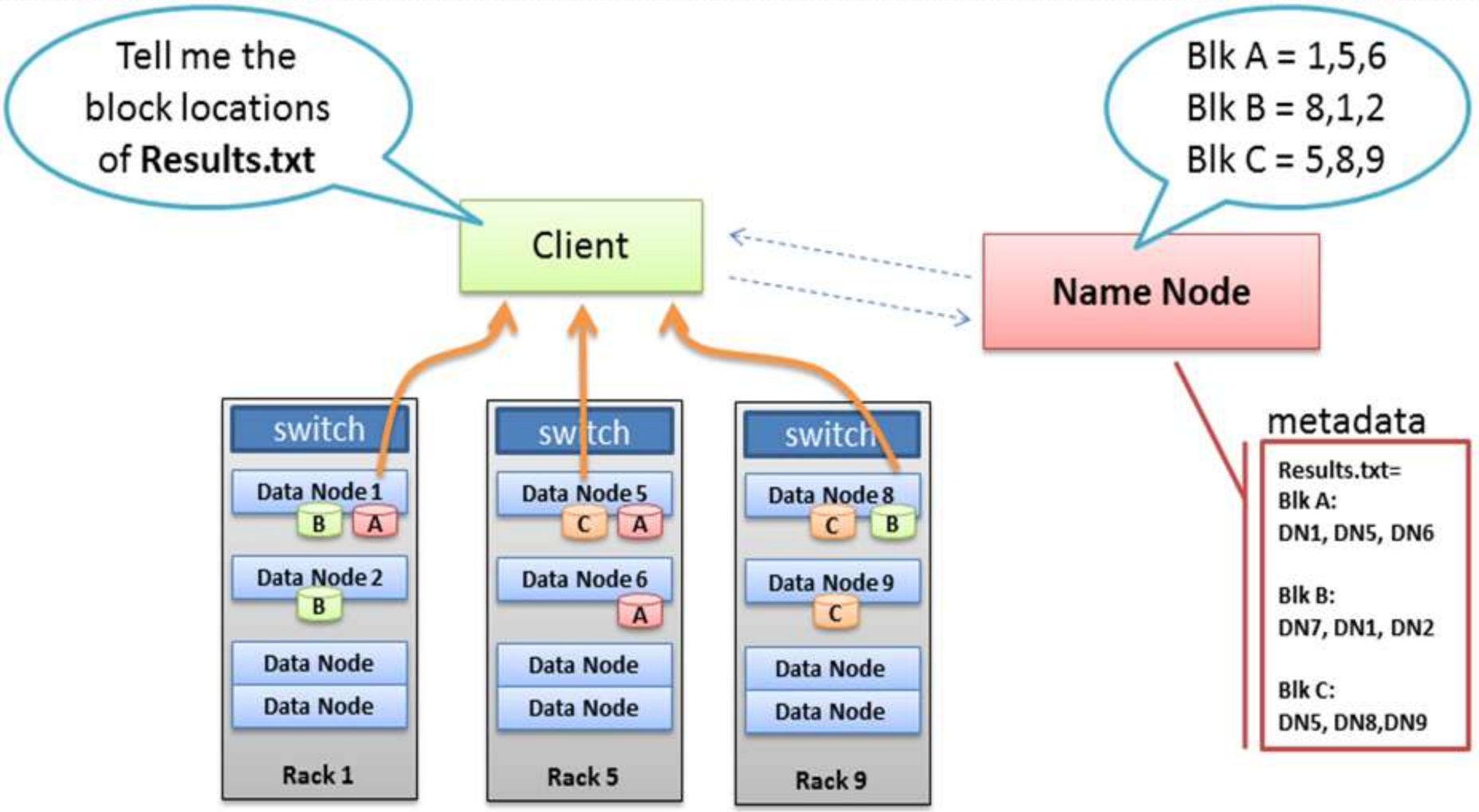
Hadoop 1.x Architecture



Data Flow

- It's good to know how client communicates with HDFS with the help of NameNode.
  - Reading a file from HDFS
  - Writing a file into HDFS

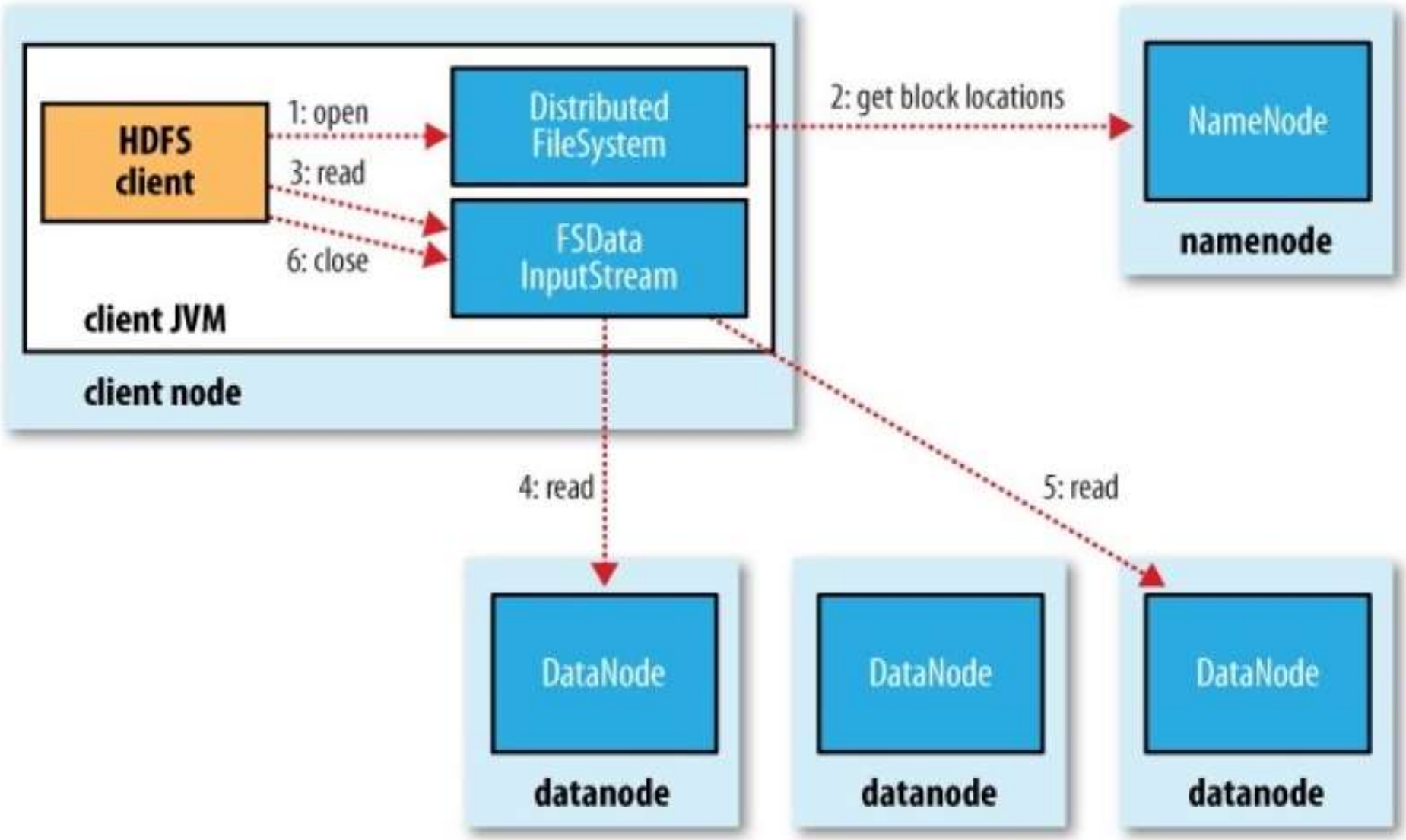
File Read Request



File Read Request

- In HDFS, an application client wishing to read a file must first contact the NameNode to determine where the actual data blocks of that file are stored.
- The NameNode returns the relevant block ids & the location where the block is held (i.e. DataNodes). This DataNode list is sorted according to the proximity to the client.
- The client contacts the closest DataNode to retrieve the data.
- After one block read is done, another closest DataNode from the list is contacted for 2nd block reading.
- An important feature of the design is that data is never moved through the NameNode. All data transfer occurs directly between clients and DataNodes; communications with the NameNode only involves transfer of metadata.

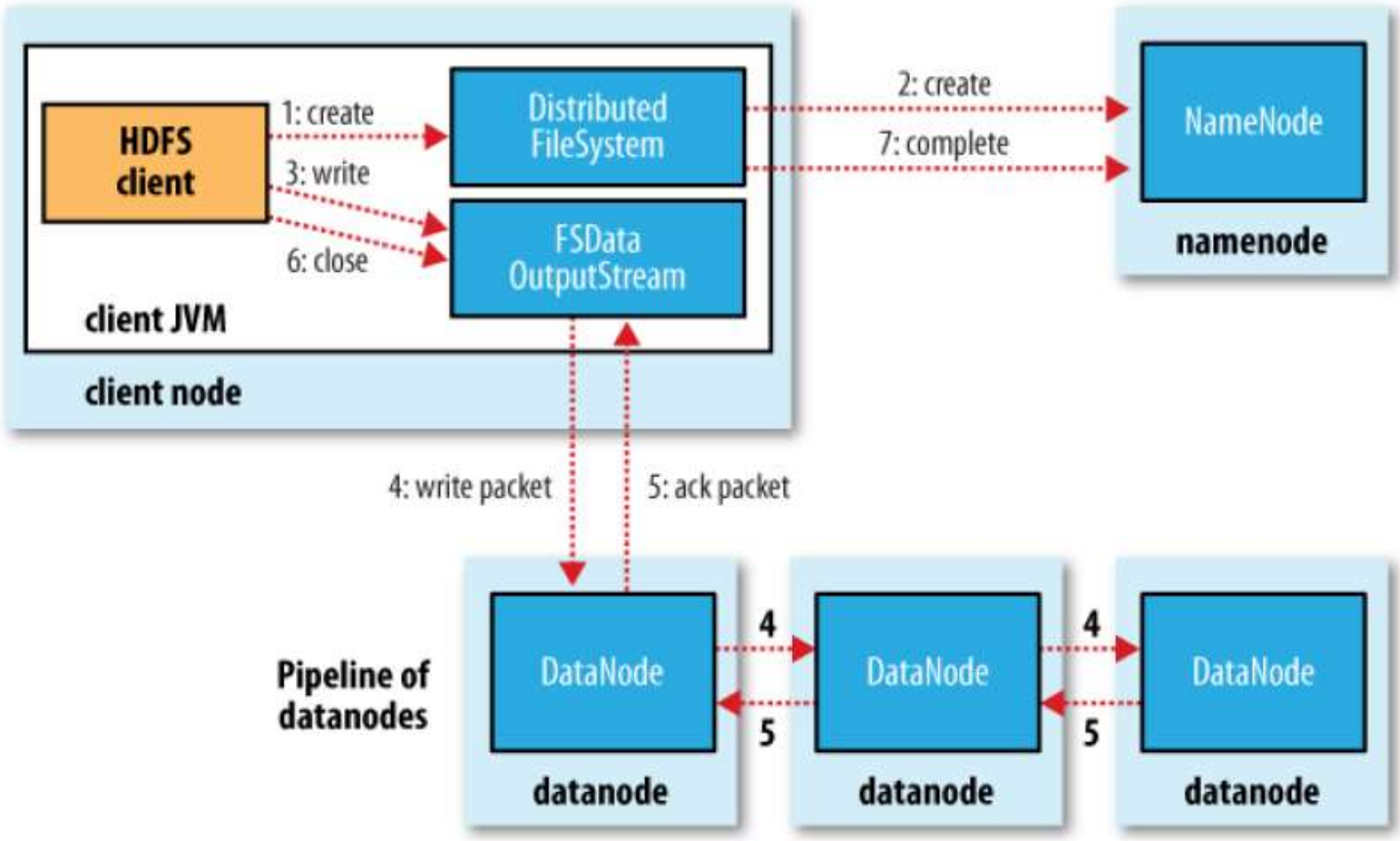
Anatomy of File Read



File Write Request

- To create a new file and write data to HDFS, the application client first contacts the NameNode, which updates the file namespace after checking permissions and making sure the file doesn't already exist.
- The NN allocates a new block on a suitable DN, and the application client is directed to stream data directly to that DN.
- From the initial DataNode, data is further propagated to other DataNodes for additional replicas.
- It's possible, but unlikely, for multiple DNs to fail while a block is being written. As long as *dfs.namenode.replication.min* replicas (default is 1) are written, the write will succeed, and the block will be asynchronously replicated across the cluster until its target replication factor is reached (*dfs.replication*, default is 3).
- Only file appends are supported.

Anatomy of File Write



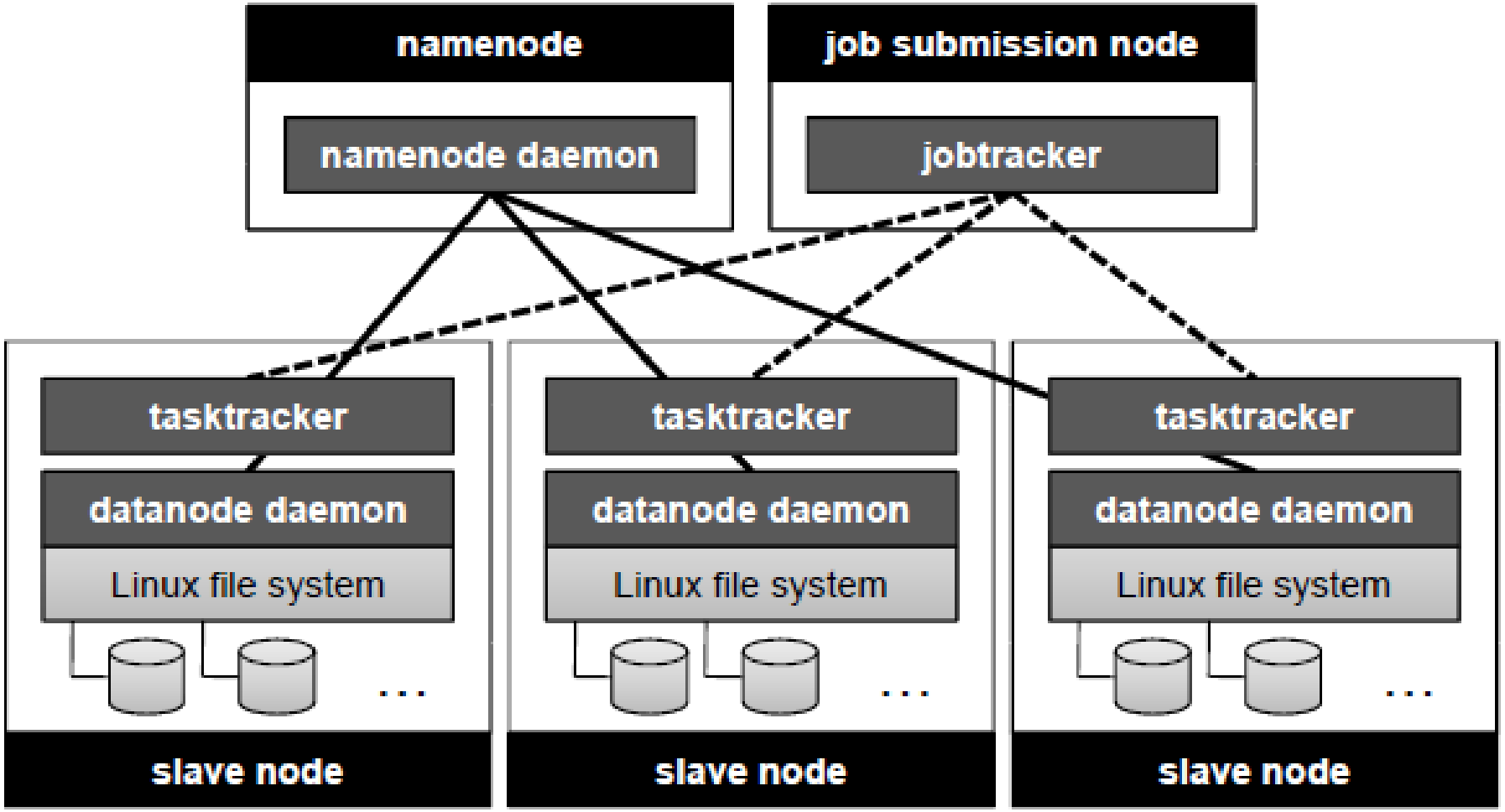
Hadoop 1.x Daemons

- **NameNode** – This daemon stores and maintains the metadata for HDFS.
- **Secondary NameNode** – Performs housekeeping functions for the NameNode.
- **JobTracker** – Manages MapReduce jobs, distributes individual tasks to machines running the Task Tracker.
- **DataNode** – Stores actual HDFS data blocks.
- **TaskTracker** – Responsible for instantiating and monitoring individual Map and Reduce tasks.

Daemon is a service or process that runs in the background in unix environment.

Each of these daemons run in its own JVM.

Architecture of Hadoop Cluster in Hadoop 1.x



NameNode

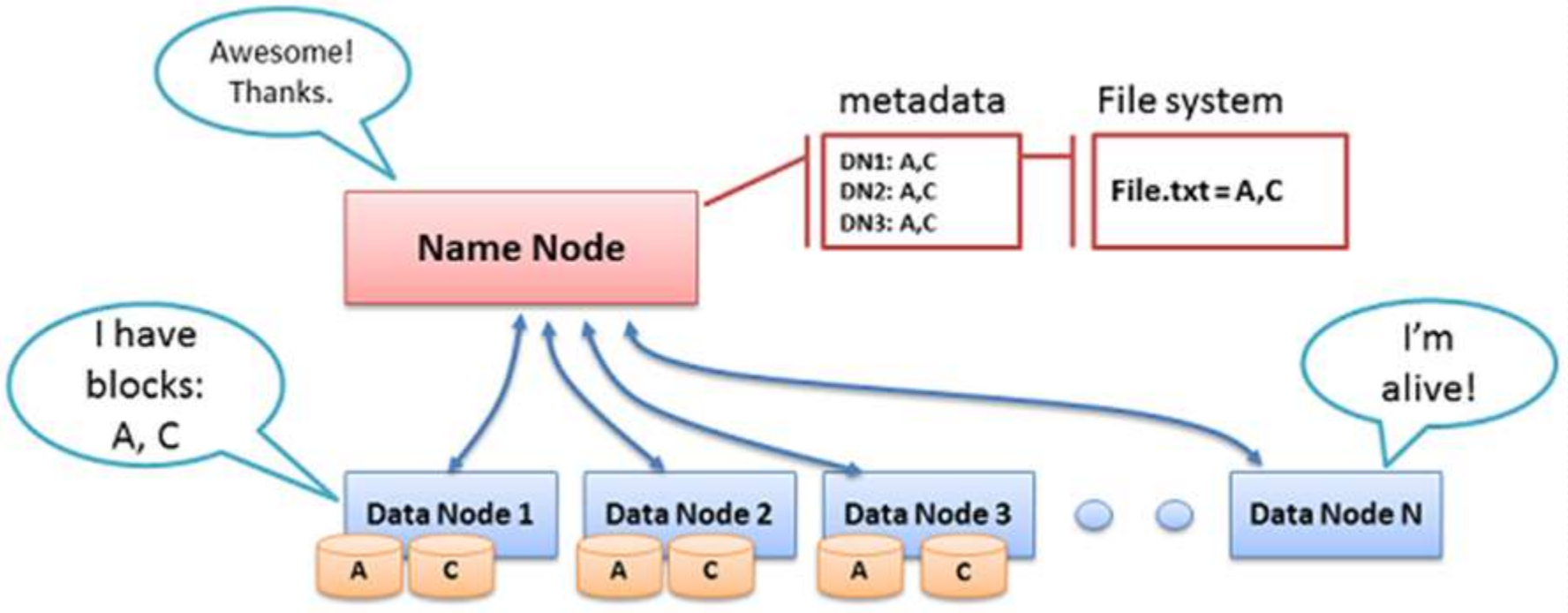
- HDFS cluster consists of a **single NameNode**, a master server that manages the file system namespace and regulates access to files by clients.
- Without the NameNode, the file system cannot be used!
- Maintains and manages the blocks that are present on DataNodes
- Very expensive hardware
- Namenode is simply a server and never initiates a request; it only responds to RPC requests issued by DataNodes or clients.
- Single point of failure

DataNode

- Slaves (work horses) which are deployed on each machine in the cluster and provide the actual storage.
- Stores data in files in its local file system.
- Store and retrieve blocks when they are told to (by clients or the NameNode).
- It stores each block of HDFS data in a separate file.
- DataNode has no knowledge about what it is storing or processing. Hiding the details of blocks from DNs is called as block abstraction.
- When the filesystem starts up, each DataNode generates a list of all HDFS blocks that are stored locally and sends this report to NameNode: BlockReport.

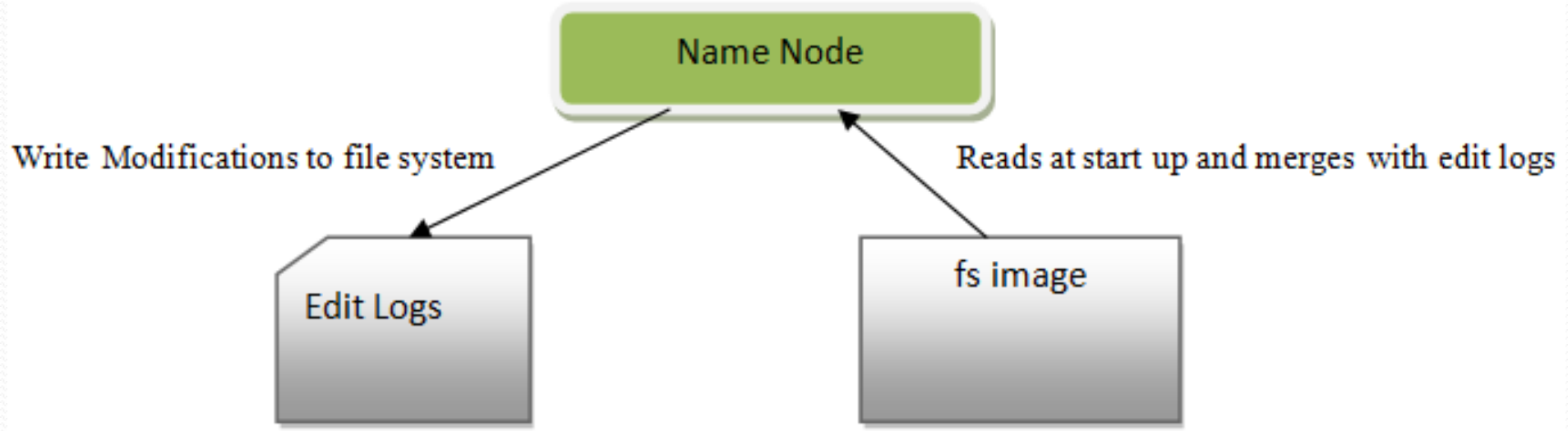
NN – DN Communication

- DataNodes send **Heartbeats** (every 3 secs)
- Every 10th heartbeat is a BlockReport
- NameNode builds metadata from BlockReports



Filesystem Metadata

- The NameNode manages the filesystem namespace. It maintains the filesystem tree and the metadata (list of files, list of blocks for each file, list of DataNodes for each block, file permissions, modification, access times, replication factor, etc.) for all the files and directories in the tree.
- This information (except the block locations info) is stored persistently on the NameNode's local disk in the form of two files: the namespace image (**FsImage**) and the **edit journal (edit logs)**.



FsImage & Edits Log

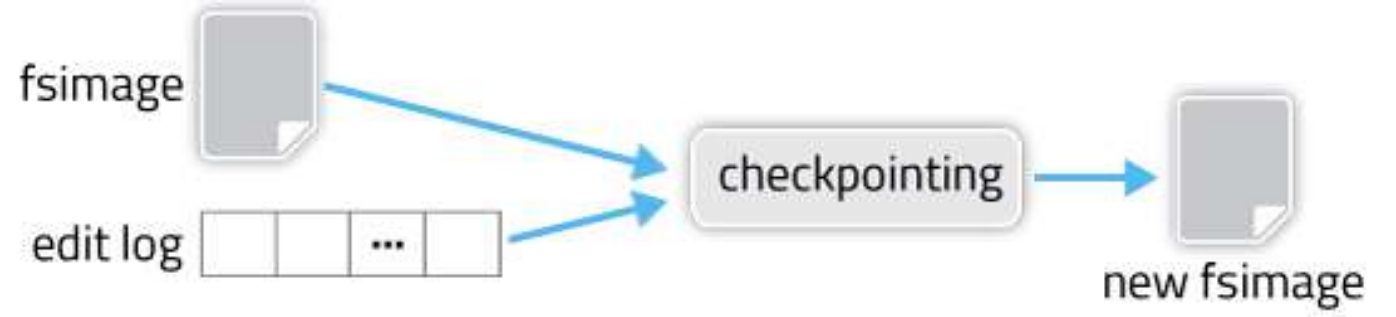
- **FsImage** is a file that represents a point-in-time snapshot of the filesystem's metadata.
- However, while the FsImage file format is very efficient to read, it's unsuitable for making small changes like renaming a single file.
- Thus, rather than writing a new FsImage every time the namespace is modified, the NameNode instead records the modifying operation in the **edits journal** for durability.
- This way, if the NameNode crashes, it can restore its state by first loading the FsImage then replaying all the operations (also called edits or transactions) in the edits journal to catch up to the most recent state of the filesystem.

Filesystem Metadata

- The NameNode also knows the DataNodes on which all the blocks for a given file are located; however, it does not store block locations persistently, because this information is reconstructed from DataNodes when the system starts.
- So information of block locations is in NameNode's memory.
- In RAM - file to block and block to data node mapping.
- In persistent storage (includes both edits journal and FsImage) - file related metadata (permissions, name, replication factor, access times and so on)

Checkpointing

- Since NameNode merges FsImage and editlogs files only during start up, the editlogs file could get very large over time on a busy cluster. Another side effect of a larger editlogs file is that next restart of NameNode takes longer.
- Checkpointing is a process that takes an FsImage and edits log and compacts them into a new FsImage by applying all the changes in the edits log to FsImage.
- This is a far more efficient operation and reduces NameNode startup time.



Checkpointing

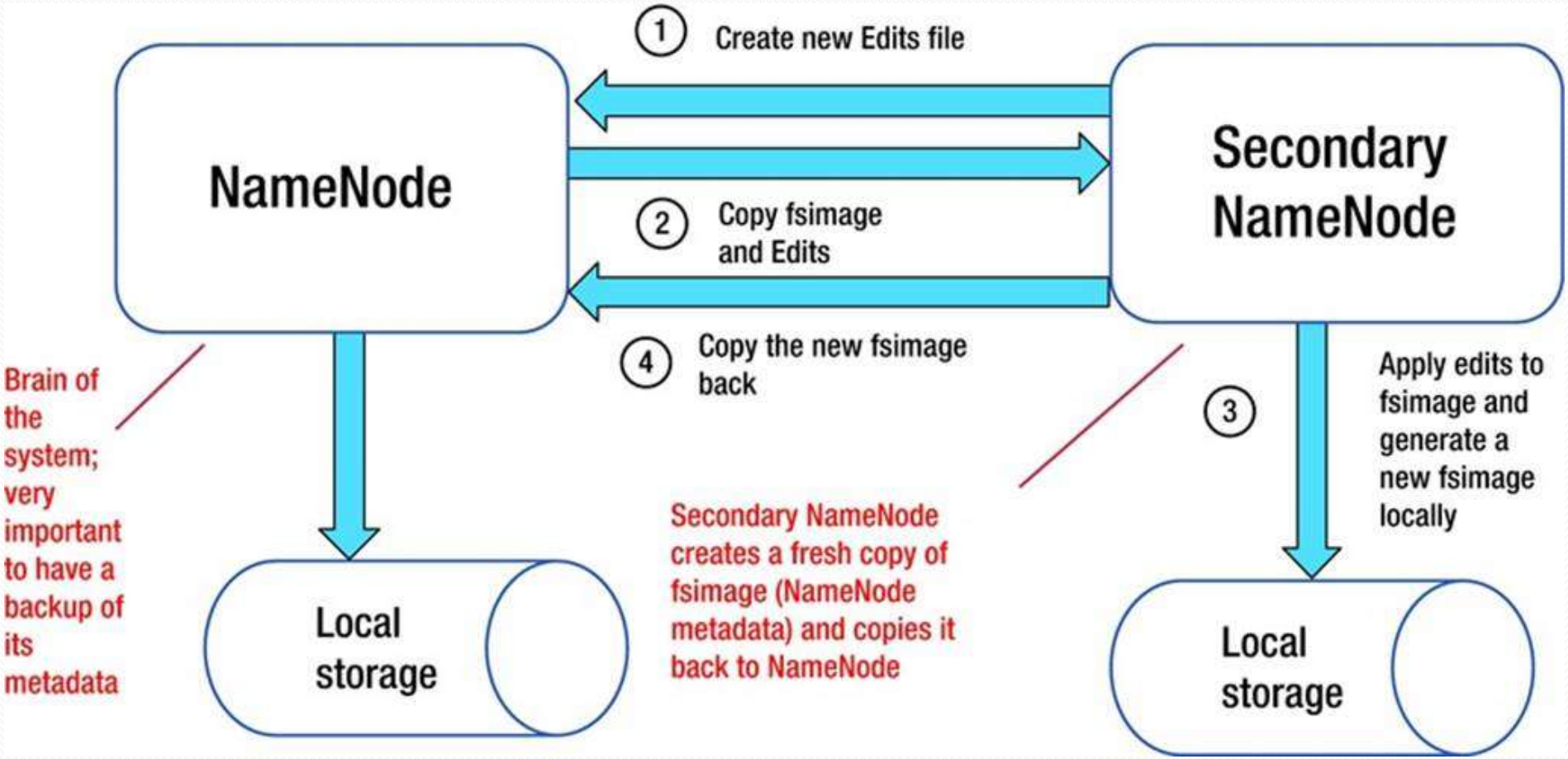
- However, creating a new FsImage is an I/O- and CPU-intensive operation, sometimes taking minutes to perform.
- During a checkpoint, the namesystem also needs to restrict concurrent access from other users.
- So, rather than pausing the active NameNode to perform a checkpoint, HDFS defers it to the SecondaryNameNode.

Secondary NameNode

- The Secondary NameNode merges the FsImage and the editlogs files periodically (default is every hour) and keeps editlogs size within a limit.
- This copy of the merged namespace image can be used in the event of the NameNode failing.
- Secondary NameNode is usually run on a different machine than the primary NameNode since its memory requirements are on the same order as the primary NameNode.

Secondary NameNode

Secondary NameNode is not a substitute for the NameNode. If the NameNode fails, the Secondary NameNode **will not** become the NameNode.



Safemode Startup of NameNode

- On startup NameNode enters Safemode.
- It reads file system metadata from the FsImage file from its local file system.
- Reads edits journal and apply logged operations to FsImage.
- Write a new checkpoint (a new FsImage consisting of the prior FsImage plus the application of all operations from the edits journal).
- Periodic checkpointing is done. So that the system can recover back to the last checkpointed state in case of a crash.
- Each DataNode checks in with Heartbeat and BlockReport.
- NameNode verifies that each block has acceptable number of replicas.
- Replication of data blocks do not occur in Safemode.
- After a configurable percentage of safely replicated blocks check in with the NameNode, NameNode exits Safemode.
- It then makes the list of blocks that need to be replicated.
- NameNode then proceeds to replicate these blocks to other DataNodes.
- Safemode for the NameNode is essentially a read-only mode for the HDFS cluster, where it does not allow any modifications to file system or blocks.

Possible Failures

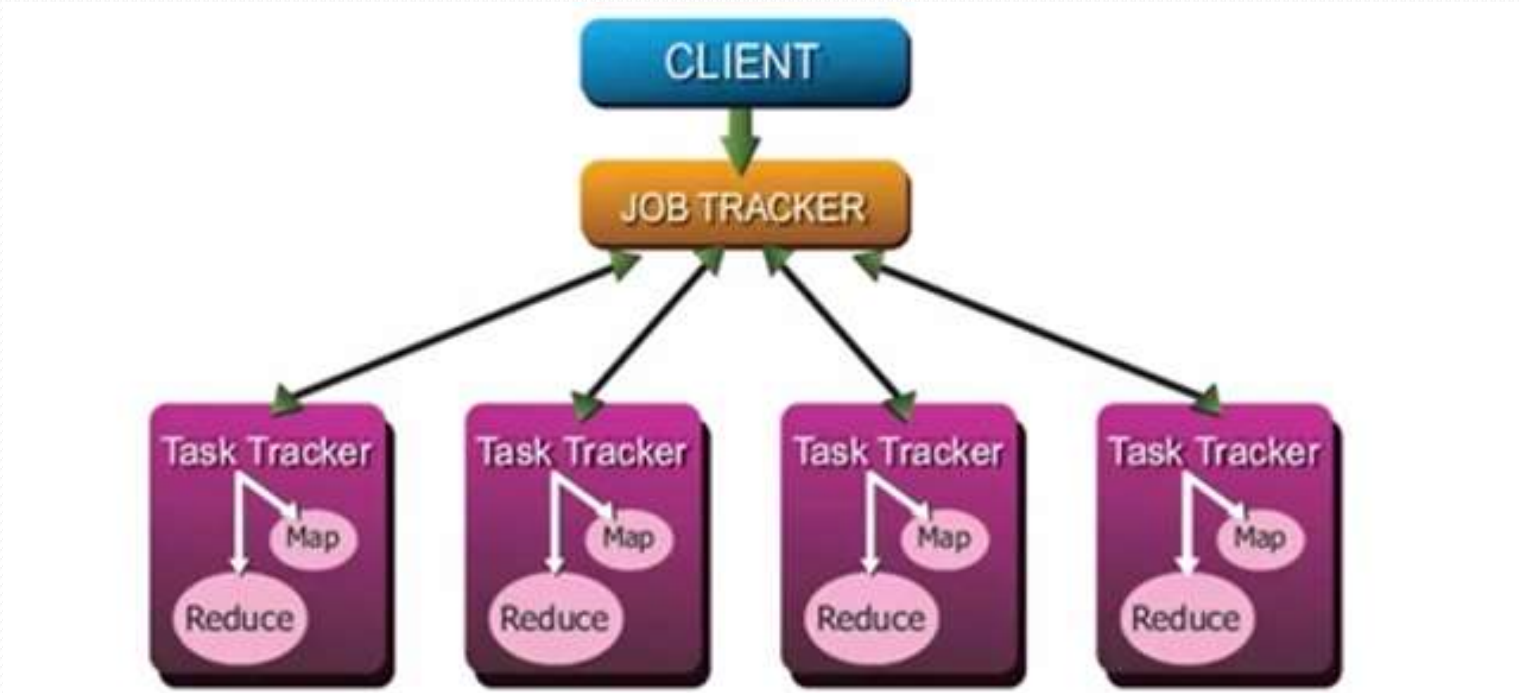
- Primary objective of HDFS is to store data reliably in the presence of failures making HDFS fault tolerant.
- Three common failures are: Namenode failure, Datanode failure and network partition.
- NN failure will make the Hadoop cluster unusable!

DataNode Failure

- A network partition can cause a subset of DataNodes to lose connectivity with the NameNode.
- NameNode detects this condition by the absence of a Heartbeat message.
- NameNode marks DataNodes without Heartbeat and does not send any IO requests to them.
- Death of a DataNode may cause replication factor of some of the blocks to fall below their specified value and so a need for re-replication arises.
- Any data registered to the failed DataNode needs to be made available on some other DataNode.

JobTracker & TaskTracker

- In MapReduce 1, there are two types of daemons that control the job execution process: a JobTracker and one or more TaskTrackers.
- 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, which keeps a record of the overall progress of each job.



JobTracker

- There's one JobTracker for a cluster. It keeps the record of overall progress of each job.
- Client application is sent to the JobTracker.
- JobTracker talks to the NameNode, locates the TaskTracker near the data.
- JobTracker moves the work (jar of the program) to the chosen TT node.
- TaskTracker monitors the execution of the task and updates the JobTracker through heartbeat. Any failure of a task is detected through missing heartbeat.
- JobTracker can reschedule a failed task on a different TaskTracker.
- Intermediate merging on the nodes are also taken care of by the JT.
- The JT is also responsible for storing job history for completed jobs, although it is possible to run a job history server as a separate daemon to take the load off the JT (MapReduce job history server).

Tasktracker

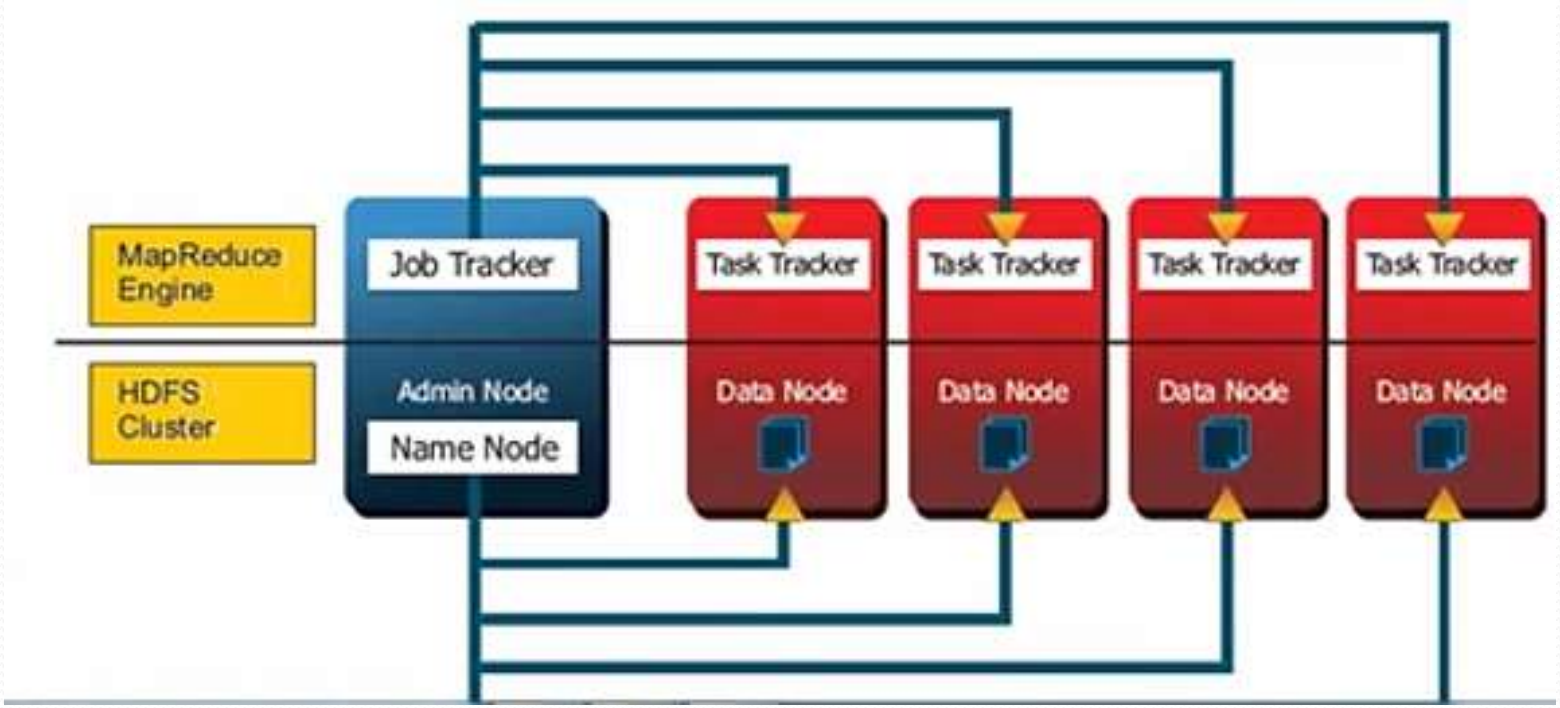
- Grass root level execution of a task.
- It accepts tasks (Map, Reduce, Shuffle, etc.) from JobTracker and executes them and sends progress report back to the JobTracker.
- There is only one TaskTracker process that runs on one DataNode and it runs in its own JVM.
- Each TaskTracker has a number of slots for the tasks; these are execution slots available on the machine.
- It indicates the number of available slots through the heartbeat message to the JobTracker.
- The TaskTracker starts a separate JVM process to do the actual work (called as Task Instance) this is to ensure that any process failure does not take down the complete TaskTracker.
- When the Task Instances finish, successfully or not, the TaskTracker notifies the JobTracker.

Scalability Issue

- HDFS is based on an architecture where the namespace is decoupled from the data.
- The namespace forms the file system metadata, which is maintained by a dedicated server (NameNode). The data itself resides on other servers called DataNodes.
- HDFS has one NameNode for each cluster.
- The NameNode keeps the entire namespace in RAM. This architecture has a natural limiting factor: the memory size; that is, the number of namespace objects (files and blocks) the single namespace server can handle.
- On very large clusters, increasing average size of files stored in HDFS helps with increasing cluster size without increasing memory requirements on NameNode.

Hadoop 1.x

- HDFS (Hadoop Distributed File System) for storage
- MapReduce for processing



Hadoop 1 Drawbacks

1. **Too much dependency on NameNode** – Single point of failure (SPOF). If NN fails, needs manual intervention to overcome.
   - SNN is not a hot standby for the NameNode.
   - Only one NameNode and one namespace per cluster is supported.
2. **Single JobTracker** – as the processing becomes huge, JobTracker gets overloaded - Single point of failure.
   - Single listener thread to communicate with thousands of Map & Reduce tasks.
   - Performs many activities like Resource Management, Job Scheduling, Job Monitoring, Re-scheduling Jobs etc.
3. **No multitenancy** – Only MapReduce jobs can be run.
   - Only suitable for Batch Processing of Big Data.
   - Cannot be used for other processing such as Real-time, Streaming, Graph analysis, machine learning (iterative algorithms) etc.

Hadoop 1 Drawbacks

4. **Scalability issue** – Scales to only
   - ~ 4000 nodes cluster
   - ~40,000 concurrent tasks
5. **Use of static slots** - It has static Map and Reduce Slots for allocating Resources (Memory, CPU). That means once it assigns resources to Map/Reduce jobs, it cannot re-use them even though some slots are idle.
   - E.g. Suppose, 10 Map and 10 Reduce tasks are running with 10 + 10 Slots to perform a computation. All Map slots are running their tasks but all Reduce slots are idle. We cannot use these Idle slots for other purposes.
   - A map slot can only be used to run a map task, and a reduce slot can only be used for a reduce task.

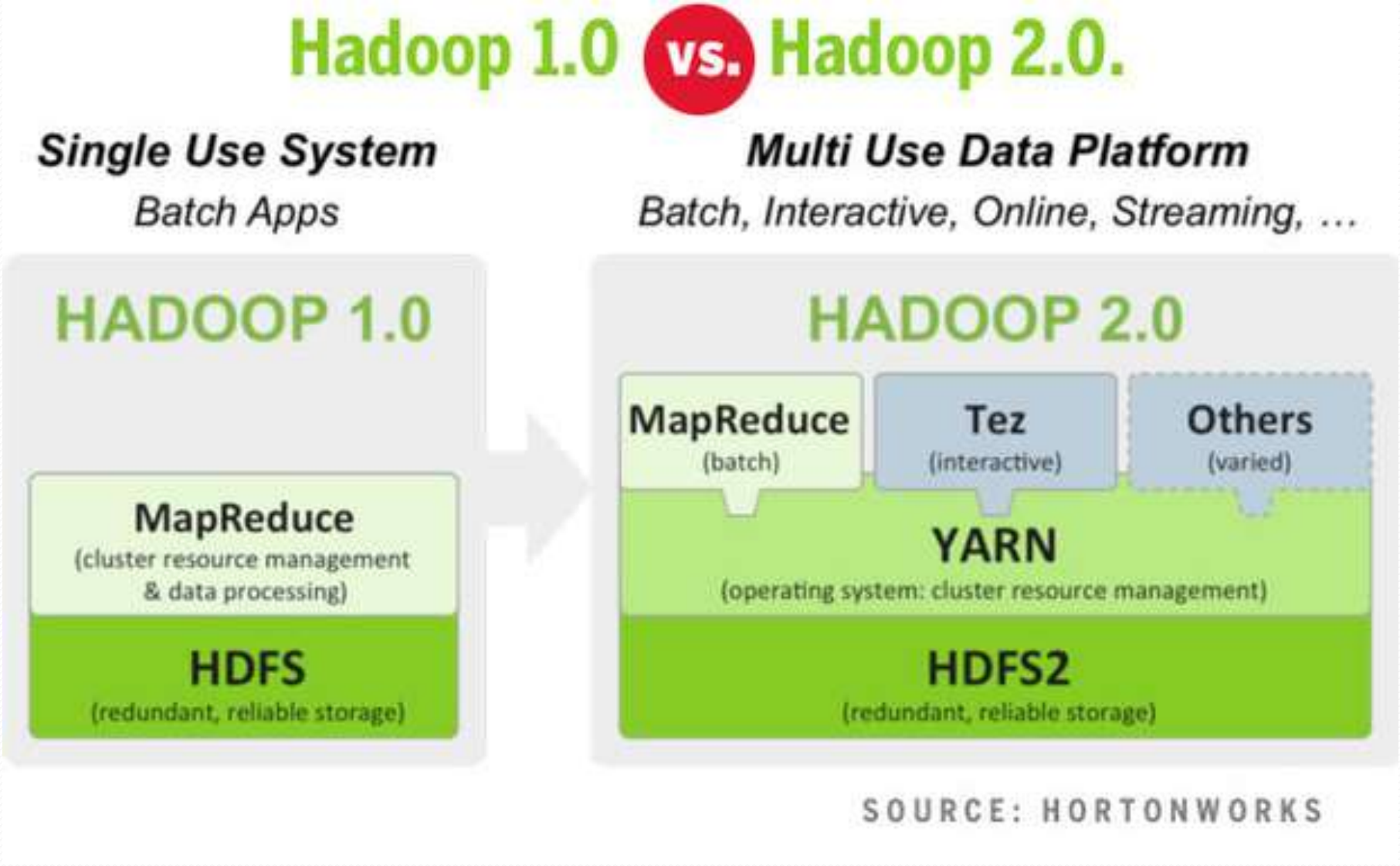
- **In Summary, Hadoop 1.x System is a Single Purpose System. We can use it only for MapReduce Based Applications.**

Introducing Hadoop 2

Hadoop 1.x is re-architected and introduced new components and concepts to solve Hadoop 1.x limitations.

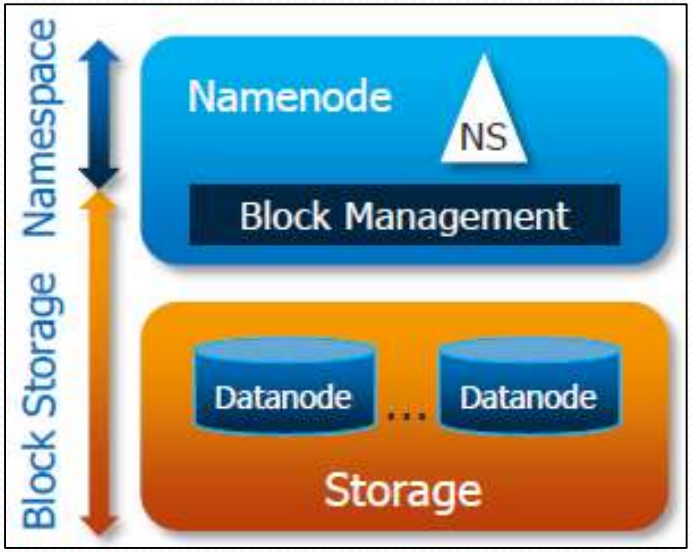
- **High Availability** – Taking care of NameNode SPOF problem
- **YARN -** Taking care of JobTracker SPOF and added support for non-mapreduce type of processing (multitenancy) making MapReduce as a user library, or one of the applications residing in Hadoop.
- **HDFS Federation** - Added support for multiple namespaces with multiple NNs.
- **High Cluster Utilization** - Use of variable-sized Containers instead of fixed-size Slots mechanism
- **Improved Scalability** - Hadoop 2.x supports more than 10,000 nodes per cluster.
- **MRv2** (simply MRv1 rewritten to run on top of YARN) – no need to rewrite existing MapReduce jobs.
- **Beyond Java**

Architecture Comparison



Motivation for HDFS Federation

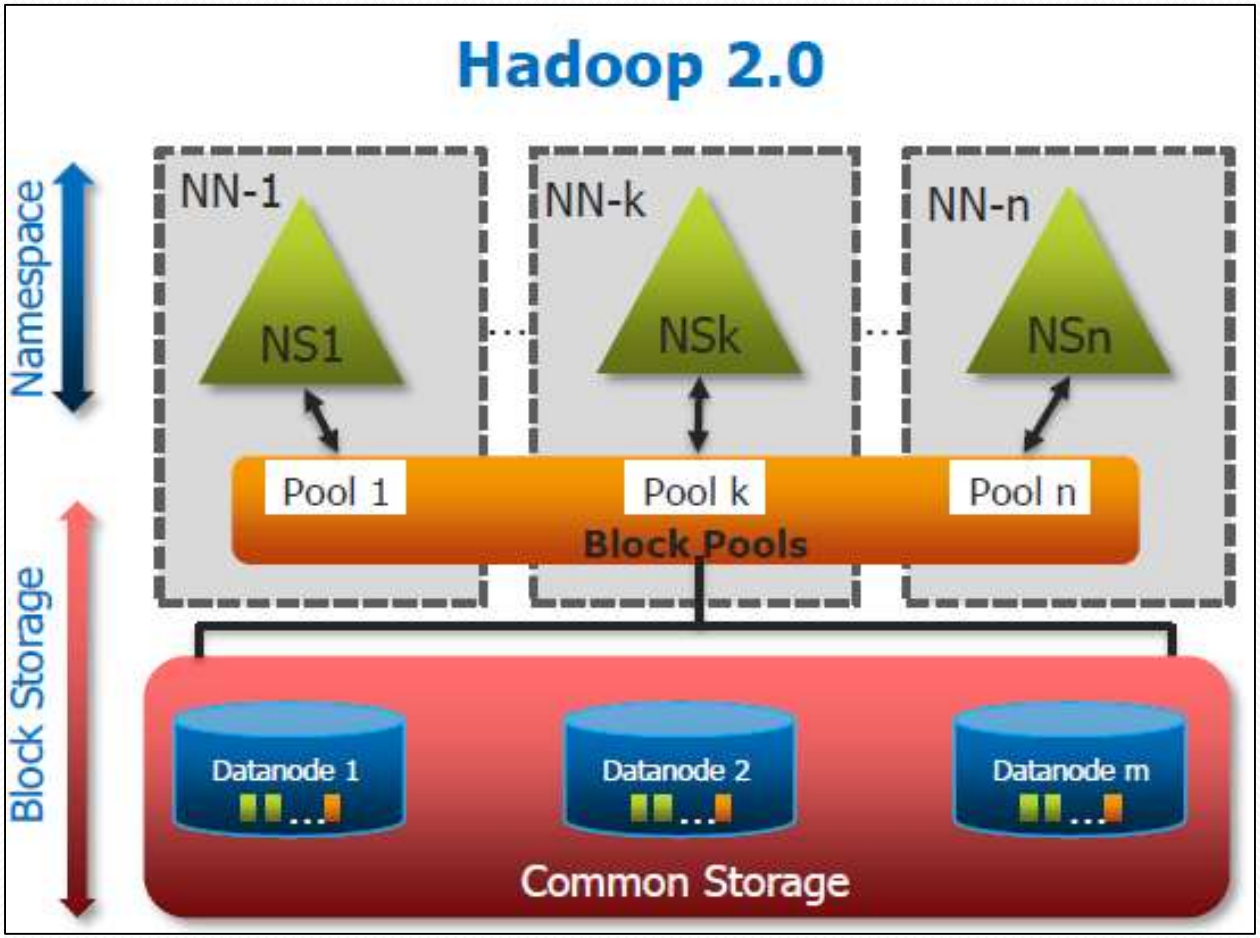
- **Limited Namespace Scalability**
  - The NameNode keeps a reference to every file and blocks in the filesystem in memory, which means that on very large clusters with many files, memory becomes the limiting factor for scaling.
- **Lack of isolation**
  - Experimental applications can overload the NameNode and slow down production jobs.
  - Production applications with different requirements can benefit from "own" NameNode.



HDFS Federation

- HDFS Federation partitions the filesystem namespace over multiple separated NameNodes each of which manages a portion of the filesystem namespace.
- This concept allows a cluster to scale by adding NameNodes.
- For example, one NameNode might manage all the files rooted under */user*, and a second NameNode might handle files under */share*.
- Removes tight coupling of Block storage and Namespace.

HDFS Federation



NNs doesn't talk to each other

NNs manages only slice of namespace

DNs can store blocks managed by any NN

HDFS Federation

- Under federation, each NameNode manages a *namespace volume*, which is made up of the metadata for the namespace, and a *block pool* containing all the blocks for the files in the namespace.
- Namespace volumes are independent of each other, which means NameNodes do not communicate with one another, and furthermore the failure of one NameNode does not affect the availability of the namespaces managed by other NameNodes.
- Block pool storage is not partitioned however, so DataNodes register with each NameNode in the cluster and store blocks from multiple block pools.

Motivation for NameNode High Availability (HA)

- Prior to Hadoop 2, the master HDFS process could only run on a single node, resulting in single point of failure (SPOF).
  - Secondary NameNode and HDFS Federation do not change that!
  - Recovery from failed NameNode may take even tens of minutes!

- Two types of downtimes
  - Unexpected failure (infrequent)
  - Planned maintenance (common)

NameNode High Availability

- The High Availability (HA) feature in Hadoop 2 addresses the NameNode SPOF problem by providing the option of running two redundant NameNodes in the same cluster in an Active/Passive configuration with a hot standby.
- This allows a fast failover to a new NameNode in the case that a machine crashes, or a graceful administrator-initiated failover for the purpose of planned maintenance.
- In the event that the active NameNode fails, the standby NameNode will take over as the active NameNode.
- This failover can be configured to be automatic, negating the need for human intervention. The fact that a NameNode failover occurred is transparent to Hadoop clients.

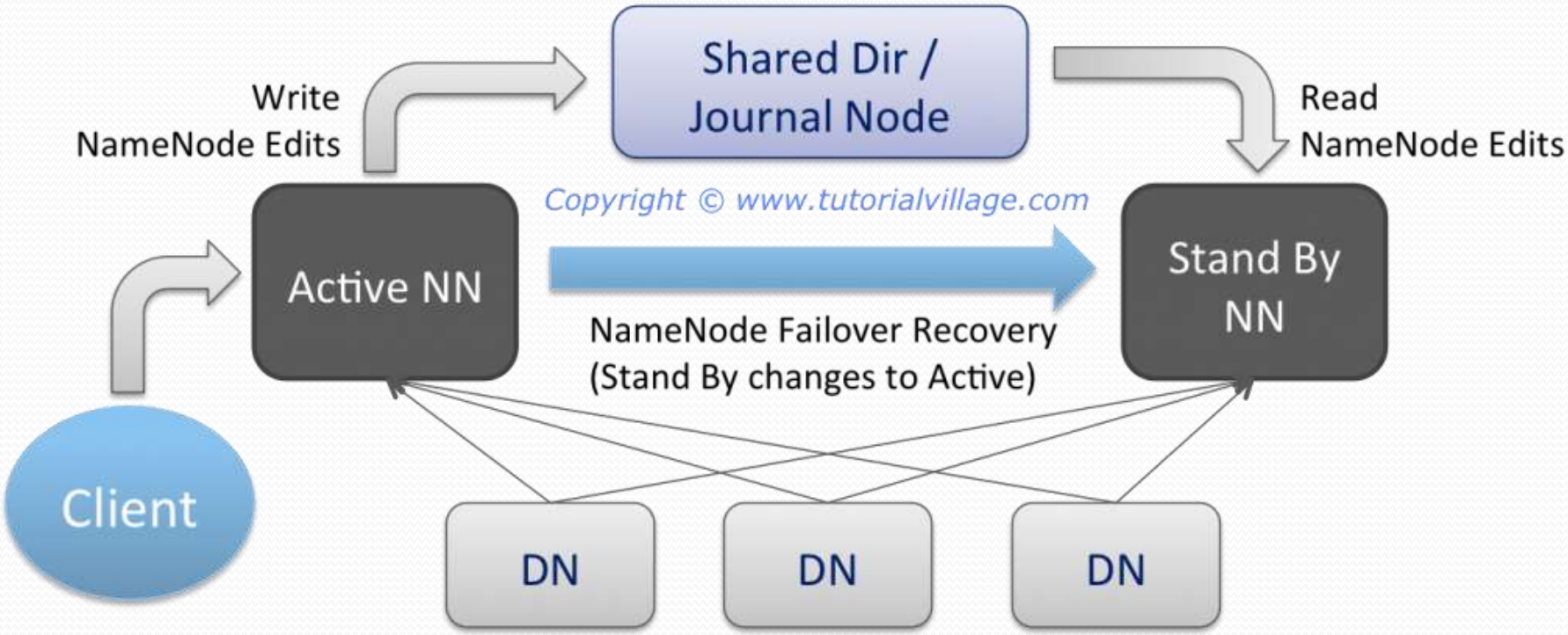
NameNode High Availability

- In a typical HA cluster, two separate machines are configured as NameNodes. At any point in time, exactly one of the NameNodes is in an Active state, and the other is in a Standby state.
- The Active NameNode is responsible for all client operations in the cluster, while the Standby is simply acting as a slave, maintaining enough state to provide a fast failover if necessary.
- In order for the Standby node to keep its state synchronized with the Active node, the current implementation requires that the two nodes both have access to a shared directory on a shared storage device.
- When any namespace modification is performed by the Active node, it durably logs a record of the modification to an edit log file stored in the shared directory.

NameNode High Availability

- The Standby node is constantly watching this directory for edits, and as it sees the edits, it applies them to its own namespace.
- In the event of a failover, the Standby will ensure that it has read all of the edits from the shared storage before promoting itself to the Active state.
- This ensures that the namespace state is fully synchronized before a failover occurs.
- In order to provide a fast failover, it is also necessary that the Standby node has up-to-date information regarding the location of blocks in the cluster.
- In order to achieve this, the DataNodes are configured with the location of both NameNodes, and send block location information and heartbeats to both.

NameNode HA Architecture



DataNodes send block location information & heartbeats to both NNs but listens to the orders only from the active one!

HDFS Highly-Available Federated Cluster

Any combination is possible:

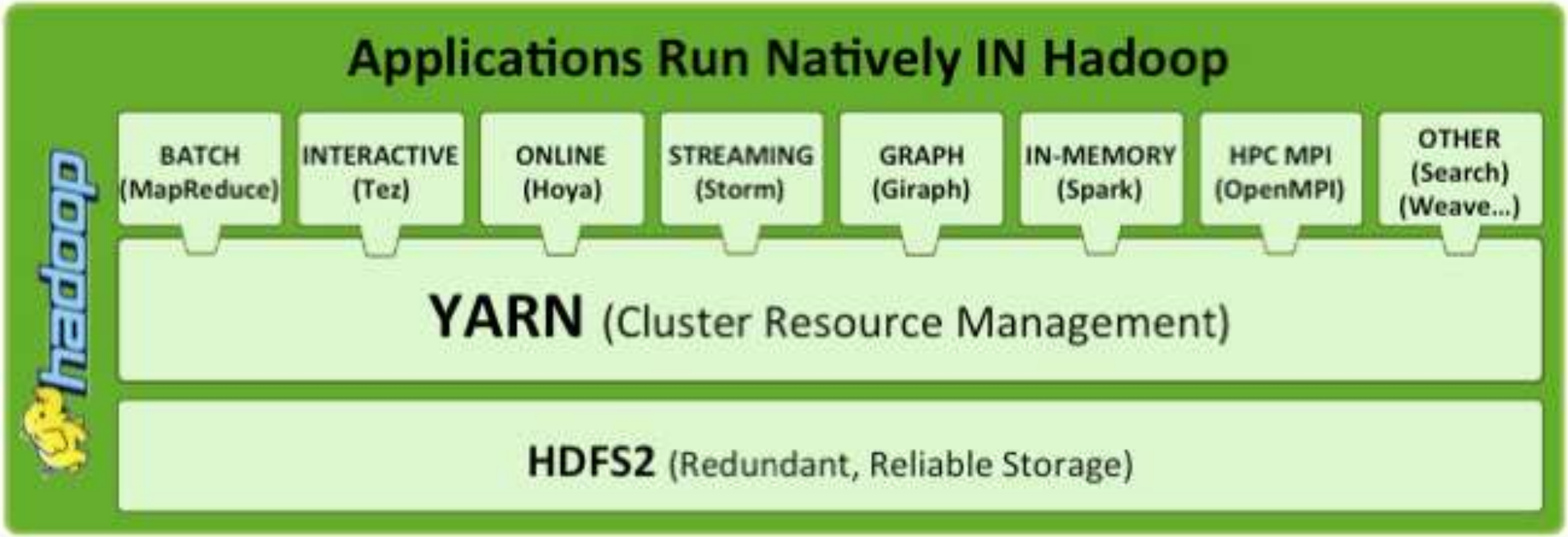
- Federation without HA
- HA without Federation
- HA with Federation
  - e.g. NN HA for HBase, but not for others.

YARN

- YARN (Yet Another Resource Negotiator) is Hadoop's cluster resource management system which became a sub-project of the larger Apache Hadoop project in 2012 and quickly became a key feature of Hadoop 2.
- Originally described by Apache as a redesigned resource manager, YARN is now characterized as a large-scale, distributed operating system for big data applications.
- Hadoop 1.x closely paired HDFS with the batch-oriented MapReduce programming framework, which handles resource management and job scheduling on Hadoop systems.
- But YARN, sometimes called as MapReduce 2.x, is a software rewrite that decouples MapReduce's resource management and scheduling capabilities from the data processing component, enabling Hadoop to support more varied processing approaches and a broader array of applications.

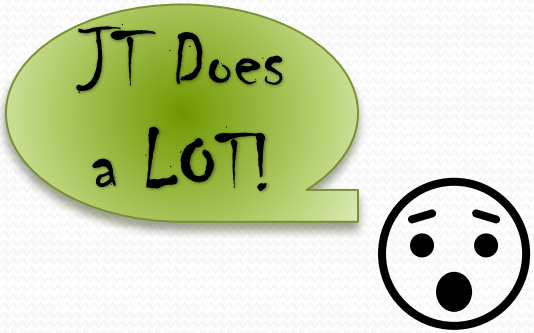
YARN Taking Hadoop Beyond Batch

- Hadoop cluster can now run interactive querying and streaming data applications simultaneously with MapReduce batch jobs.
- Store ALL DATA in one place – Interact with that data in multiple ways



Motivation for YARN

- In Hadoop 1 which is suitable only for MapReduce jobs, it is the responsibility of the MR framework to manage resources, schedule tasks and also perform the data computation job.
- In MRv1, the JobTracker was overloaded.
  - Manages the computational resources (map and reduce slots)
  - Keeps track of thousands of TTs
  - Schedules all user jobs
    - Schedules all the tasks that belong to a job
    - Monitors task execution
    - Restarts failed and speculatively runs slow tasks
    - Calculates job counters totals
    - Stores job history for completed jobs



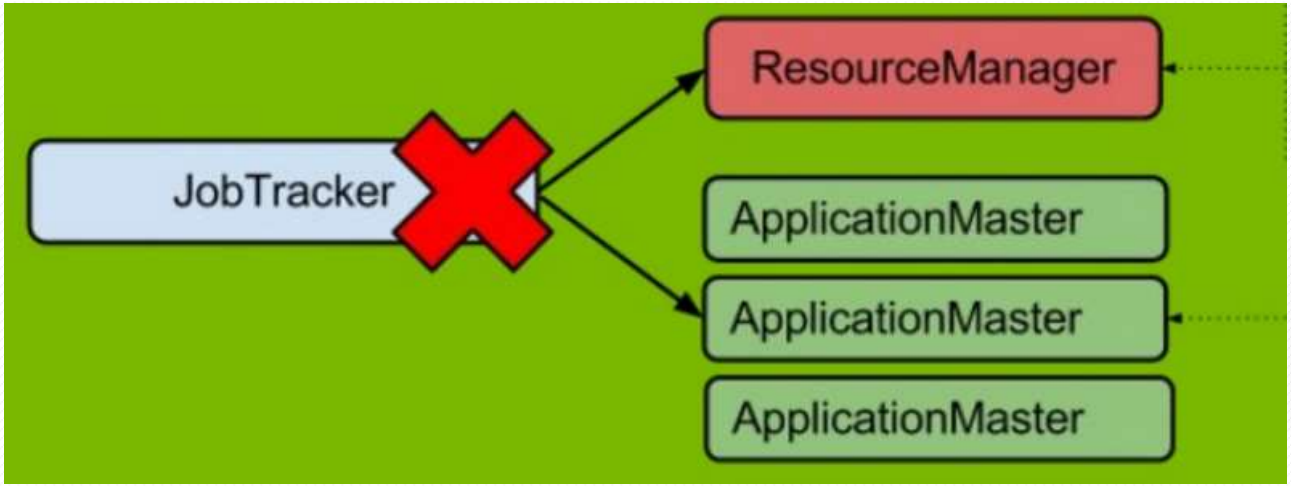
Need to redesign JobTracker

- Separate cluster resource management from job coordination
- Use slaves (many of them!) to manage jobs' life cycle
- Scale to at least 10K nodes, 10K jobs and 100K tasks

Hadoop 2 - YARN:
Resource Manager & Application Master

Hadoop 1.x JobTracker is now divided into two components:

- **Resource Manager** :- To manage resources in the cluster
- **Application Master** :- To manage life cycle of applications like MapReduce, Spark etc.

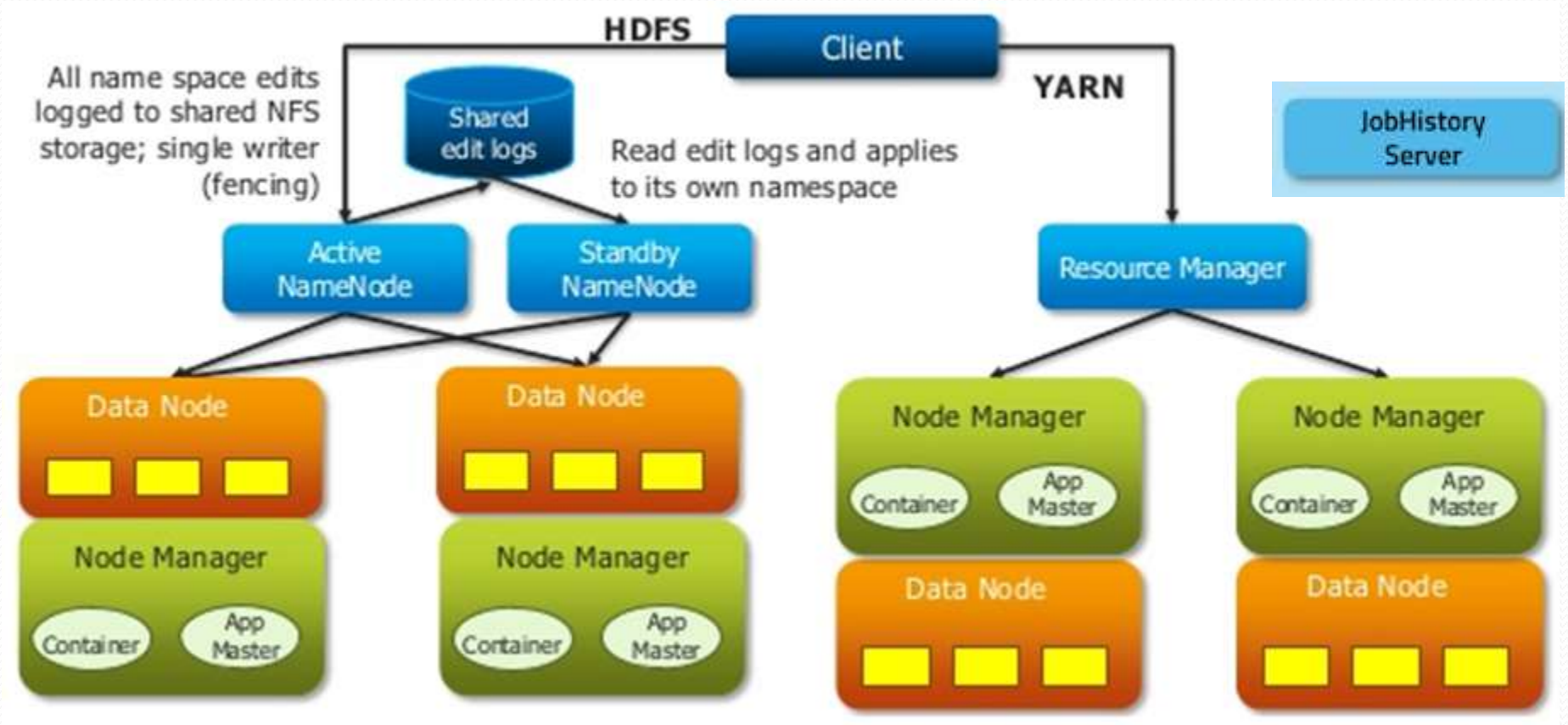


One per cluster
Application level scheduling

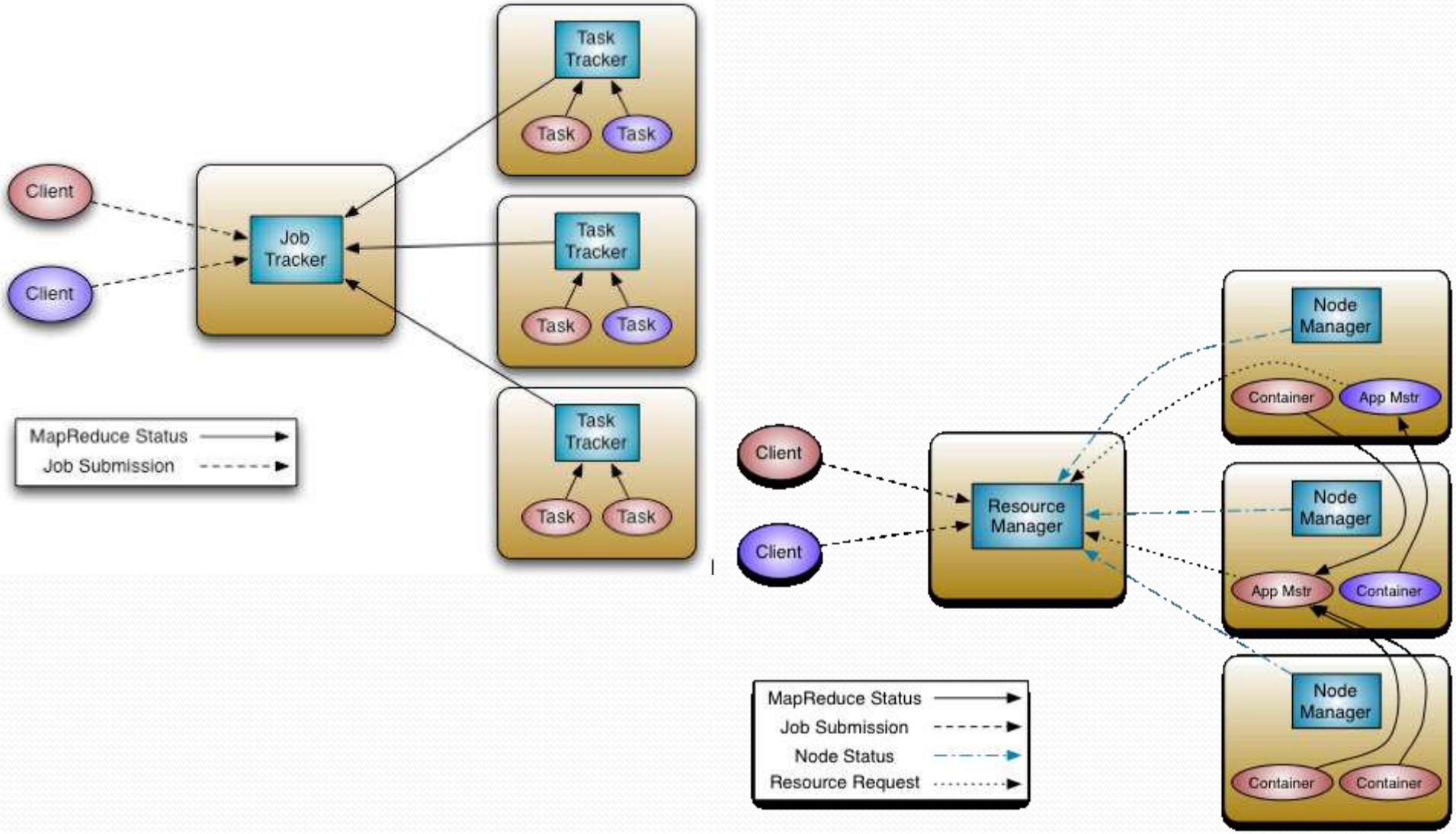
One per application
Task level scheduling and monitoring

Hadoop 2.x Architecture

NN high availability



MR 1 Vs. MR 2

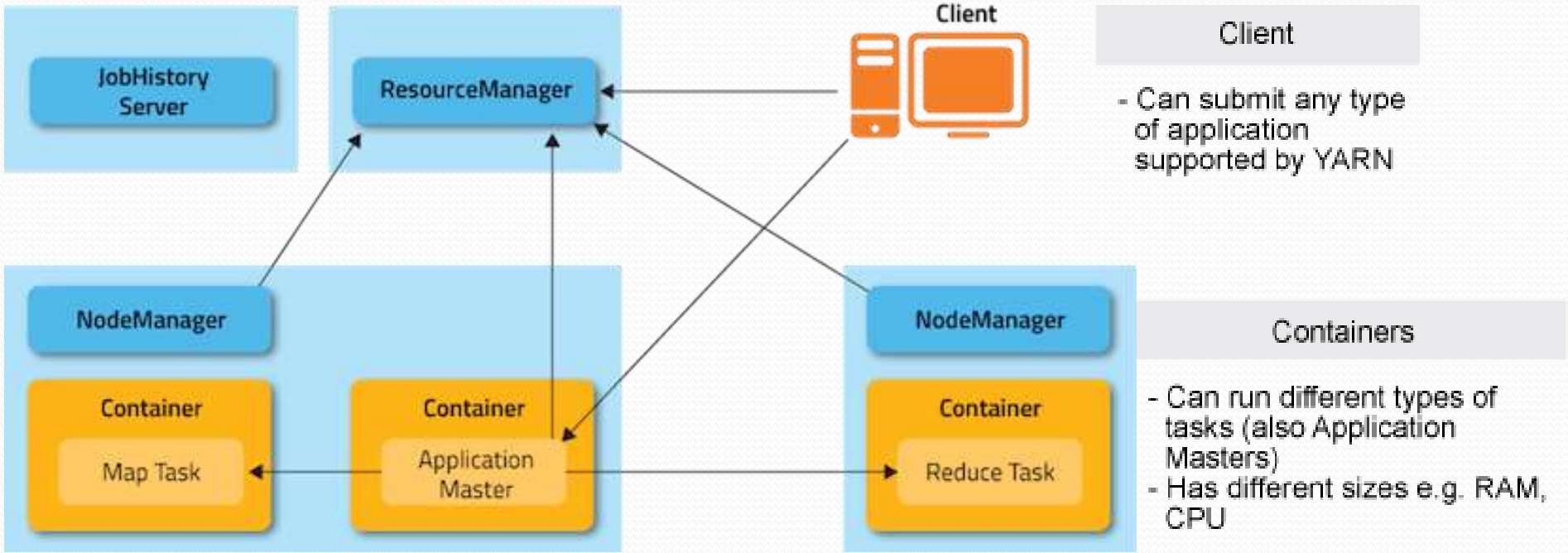


YARN Specific Daemons

(1) Resource Manager (2) Node Manager
(3) Application Master (4) Containers

ResourceManager (RM)

- Keeps track of live NodeManagers and available resources
- Allocates available resources to appropriate applications and tasks
- Monitors application masters



Client

- Can submit any type of application supported by YARN

Containers

- Can run different types of tasks (also Application Masters)
- Has different sizes e.g. RAM, CPU

ApplicationMaster (AM)

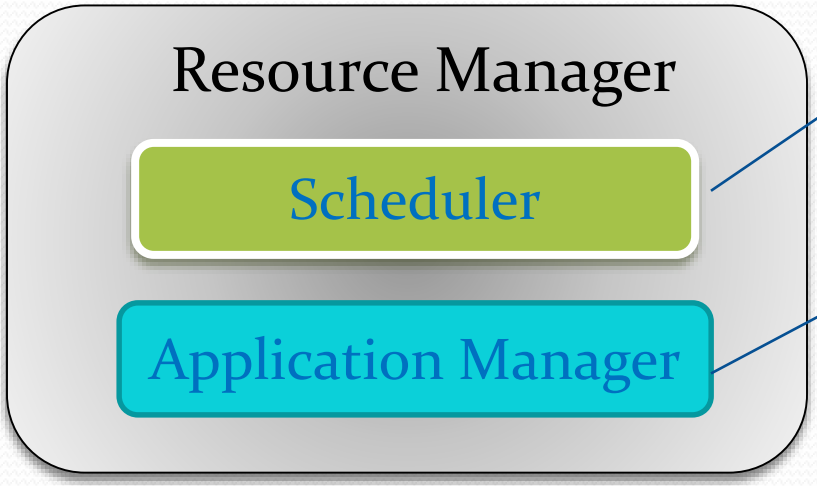
- Coordinates the execution of all tasks within its application
- Asks for appropriate resource containers to run tasks

NodeManager (NM)

- Provides computational resources in form of containers
- Managers processes running in containers

YARN Specific Daemons Contd.

- **Resource Manager**
  - The ultimate authority that arbitrates resources (CPU, memory, disk, network) among all the applications in the system.
  - It is the YARN's master process and is responsible for scheduling and managing resources, called "containers".
  - Single Resource Manager for the entire cluster
  - Long life, high quality hardware



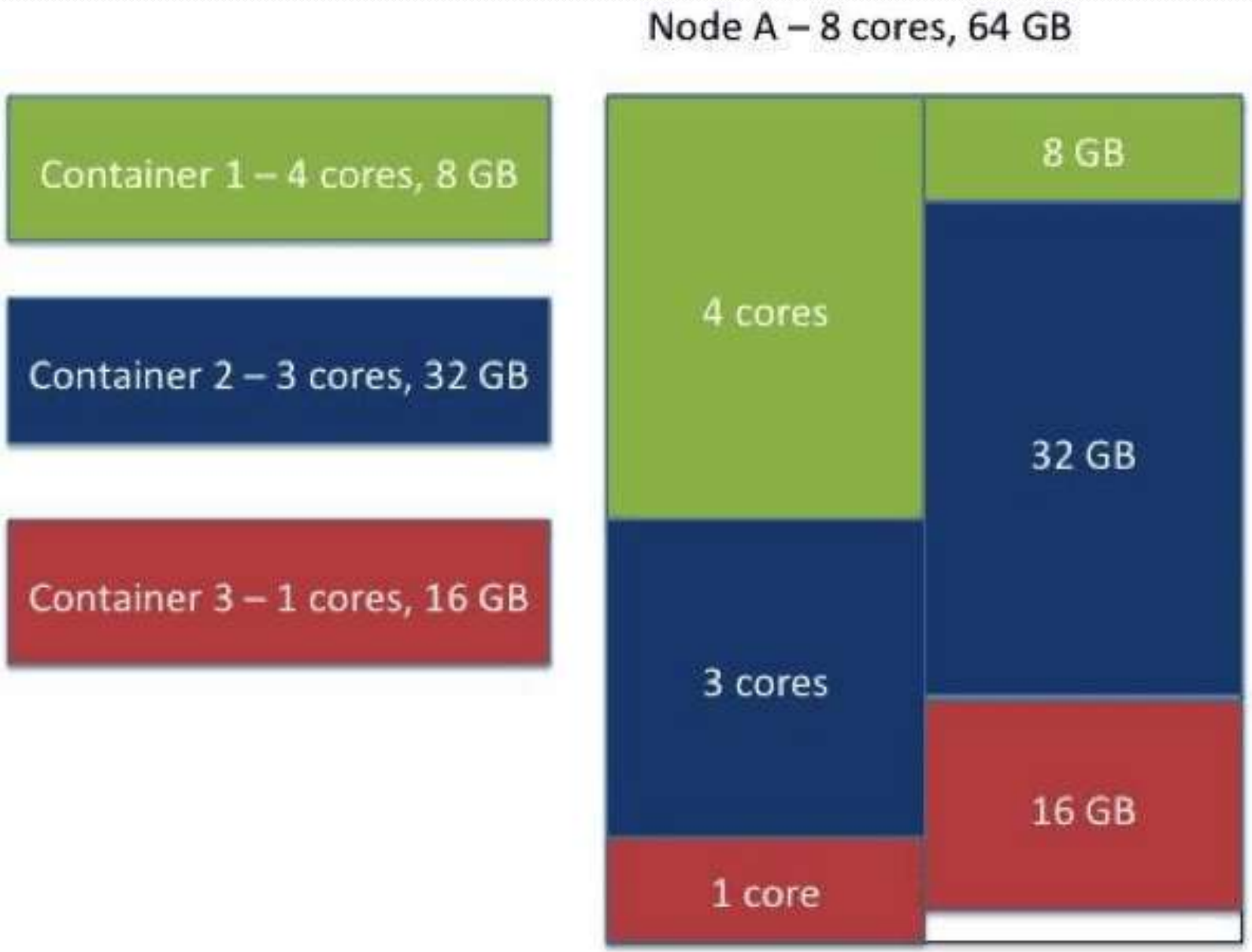
Pure scheduling. No monitoring, no restarting of jobs or tasks.

Accepting application submissions, negotiating the first container for the ApplicationMaster, restarting it on failure.

YARN Specific Daemons Contd.

- **Node Manager**
  - The slave YARN process that runs on each DataNode. It is more flexible and efficient than TT and it is responsible for launching and managing containers and reporting the same to the RM.
  - Executes any computation that makes sense to ApplicationMaster (Not only map or reduce tasks)
  - Containers with variable resource sizes (e.g. RAM, CPU, n/w, disk)
    - No hard-coded split into map and reduce slots
- **Node Manager Containers**
  - Containers are YARN application-specific processes that perform some function pertinent to the application.
  - NodeManager creates a container for each task.
  - Container contains variable resource sizes.
    - E.g. 2GB RAM, 1 CPU, 1 disk

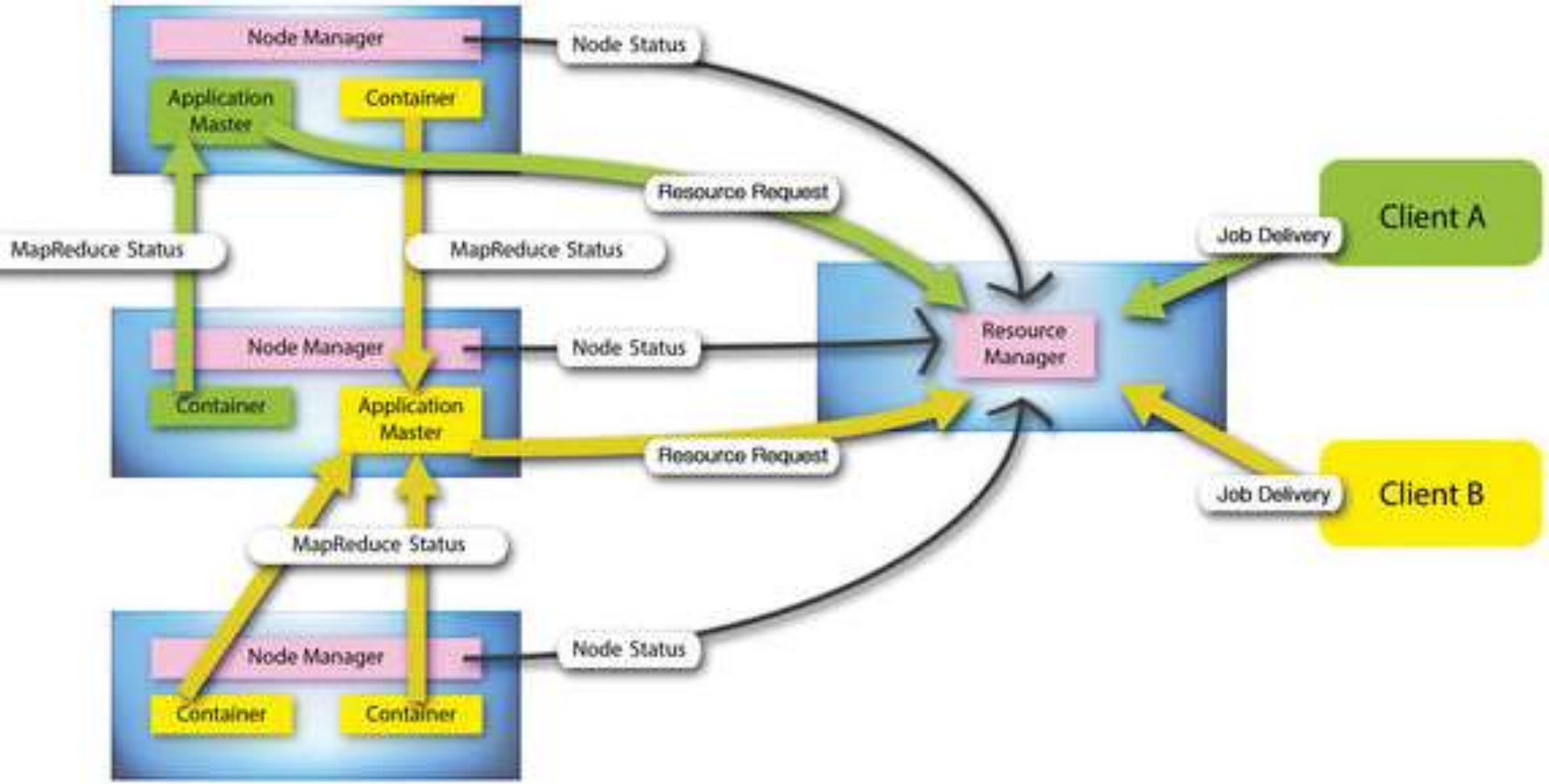
Container Allocation on a Node



YARN Specific Daemons Contd.

- **Application Master**
  - One per application and it has short life.
  - It's a framework specific library and is tasked with negotiating resources from the ResourceManager and working with the NodeManager(s) to execute and monitor the tasks.
  - It is created by the ResourceManager and is responsible for requesting containers to perform application-specific work.

YARN MR Application Execution Flow



Anatomy of a YARN Application Run

1. To run an application, a client contacts the resource manager and asks it to run an ApplicationMaster process.
2. The ResourceManager then finds a NodeManager that can launch an ApplicationMaster in a Container.
3. The ApplicationMaster requests subsequent containers from the ResourceManager that are allocated to run tasks for the application. Those tasks do most of the status communication with the ApplicationMaster allocated in Step 2.
4. Once all tasks are finished, the ApplicationMaster exits. The last container is de-allocated from the cluster.
5. Precisely what the AM does once it is running depends on the application. It could simply run a computation in the container it is running in and return the result to the client. Or, it could request more containers from the resource manager and use them to run a distributed computation (MapReduce does this).

RM High Availability (HA)

- Prior to Hadoop 2.4, ResourceManager was the SPOF in a YARN cluster.
- The ResourceManager High Availability feature adds redundancy in the form of an Active/Standby RM pair to remove this SPOF.
- Upon failover from the active ResourceManager to the standby, the applications can resume from the last state saved; e.g., map tasks in a MR job are not executed again if a failover to a new active RM occurs after the completion of the map phase.
- When there are multiple RMs, the configuration (yarn-site.xml) used by clients and nodes is expected to list all the RMs. Clients, AMs and NMs try connecting to the RMs in a round-robin fashion until they hit the Active RM. If the Active goes down, they resume the round-robin polling until they hit the "new" Active.

Basic Hadoop Admin Commands

The **~/hadoop/bin** directory contains some scripts used to launch Hadoop DFS and Hadoop Map/Reduce daemons. These are:

- **start-all.sh** - Starts all Hadoop daemons, the namenode, datanodes, the jobtracker and tasktrackers.
- **stop-all.sh** - Stops all Hadoop daemons.
- **start-mapred.sh** - Starts the Hadoop Map/Reduce daemons, the jobtracker and tasktrackers.
- **stop-mapred.sh** - Stops the Hadoop Map/Reduce daemons.
- **start-dfs.sh** - Starts the Hadoop DFS daemons, the namenode and datanodes.
- **stop-dfs.sh** - Stops the Hadoop DFS daemons.

Basic HDFS Commands

Hadoop supports shell-like commands to interact with HDFS directly.

- **Creating and listing directories**
- hadoop fs –mkdir programs
- hadoop fs –ls

- **Copy local file into HDFS file system**
- hadoop fs –copyFromLocal input/data.txt /user/cloudera/data.txt
- hadoop fs -put input/data.txt /user/cloudera/data.txt

- **Copy HDFS file into local file system**
- hadoop fs –copyToLocal data.txt /tmp/data.txt
- hadoop fs –get data.txt /tmp/data.txt

- **Remove a file from HDFS**
- hadoop fs –rm /user/cloudera/data.txt