

Hadoop Distributed File System

Requirements/Features

- Highly **fault-tolerant** VOLUME CONTAINS A FUNCTIONAL REDUNDANT SET
 $MTBF = \text{MEAN TIME BETWEEN FAILURES} \sim 3 \text{ ANNI (W MEDIA)}$
- Failure is the norm rather than exception LOOK SLOWLY
- High **throughput** $\rightarrow \frac{1}{365} = \text{PROBABILITY OF HARD DISK FAILURE} \sim 3 \text{ HARD DISK SLIPPERY AT CAPACITY}$
 $MTBF = 1 \text{ ANNO}$
- May consist of thousands of server machines, each storing part of the file system's data NOT POSSIBLE TO GET ONE A CAPACITY ON DATABASE
- Suitable for applications with **large data sets** IMPORTANT TO STORE THE WHOLE DATASET
- Time to read the whole file is more important than the reading the first record
- Not fit for
 - Low latency data access NOT NECESSARY
 - Lots of small files
 - Multiple writers, arbitrary file modifications NOT FREQUENT
- **Streaming access** to file system data
- Can be built out of **commodity hardware**

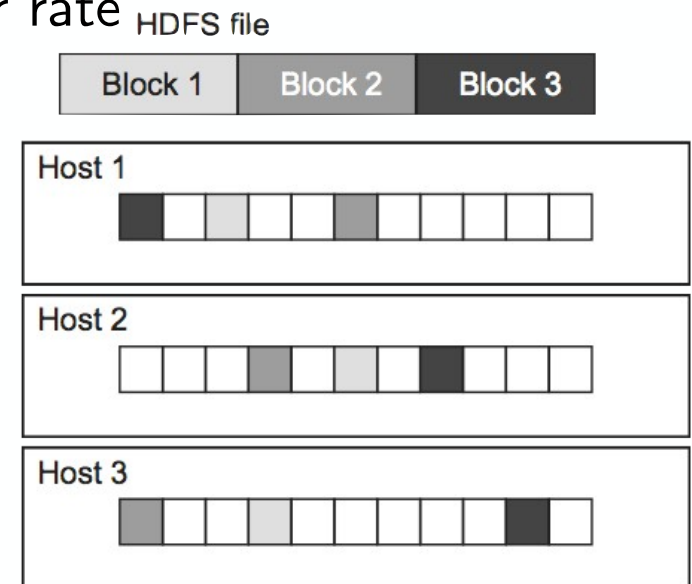
Organization

- Files are divided into **chunks** (or **blocks**)
 - UTILIZATION PER CLIENT'S DATASET
 - ALL VECTORS' PARALLEL CAN USE CONNECTIONS AT 1GBPS
- typically 64/128 megabytes in size ATOMIC UNIT STORAGE
- Blocks are **replicated** at **different** compute nodes (usually 3+)
- Nodes holding copies of one block are located on **different racks**
- **Block size** and the **degree of replication** can be decided by the **user**
- A special ^{node} ~~file~~ (the **master node**) stores, for each file, the positions of its blocks
- The master node is itself **replicated**
- A **directory** (or **tree**) for the file system knows where to find the master node
- The directory itself can be **replicated**
- All participants using the DFS know where the directory copies are

Don't confuse a master node

Blocks

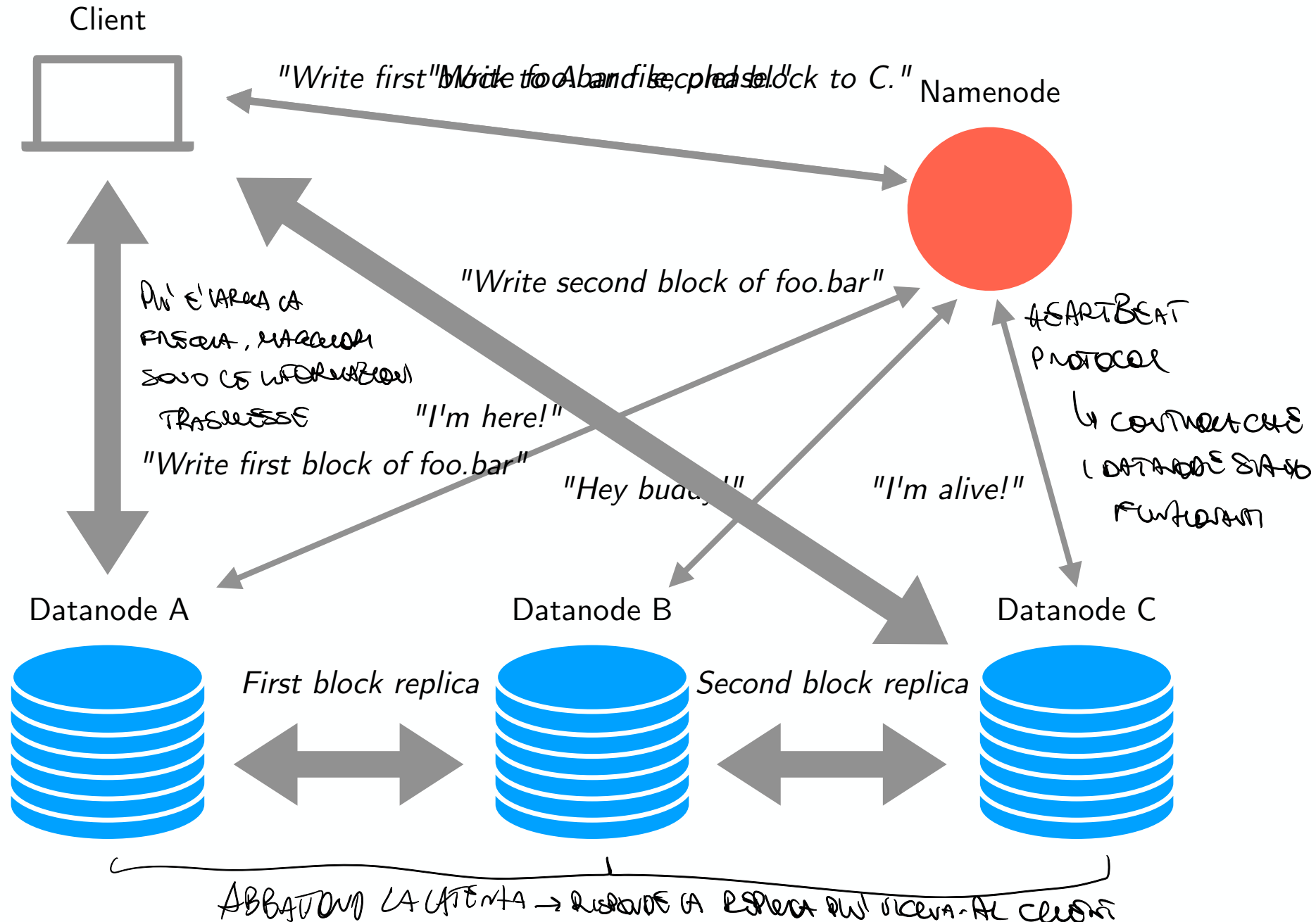
- Minimum amount of data that it can read or write
- File System Blocks are typically few KB
- Disk blocks are normally 512 bytes
- HDFS Block is much larger – 64 MB by default
 - Unlike file system the smaller file does not occupy the full 64MB block size
 - Large to minimize the cost of seeks
 - Time to transfer blocks happens at disk transfer rate
- Block abstractions allows
 - Files can be larger than block
 - Need not be stored on the same disk
 - Simplifies the storage subsystem
 - Fit well for replications
 - Copies can be read transparent to the client



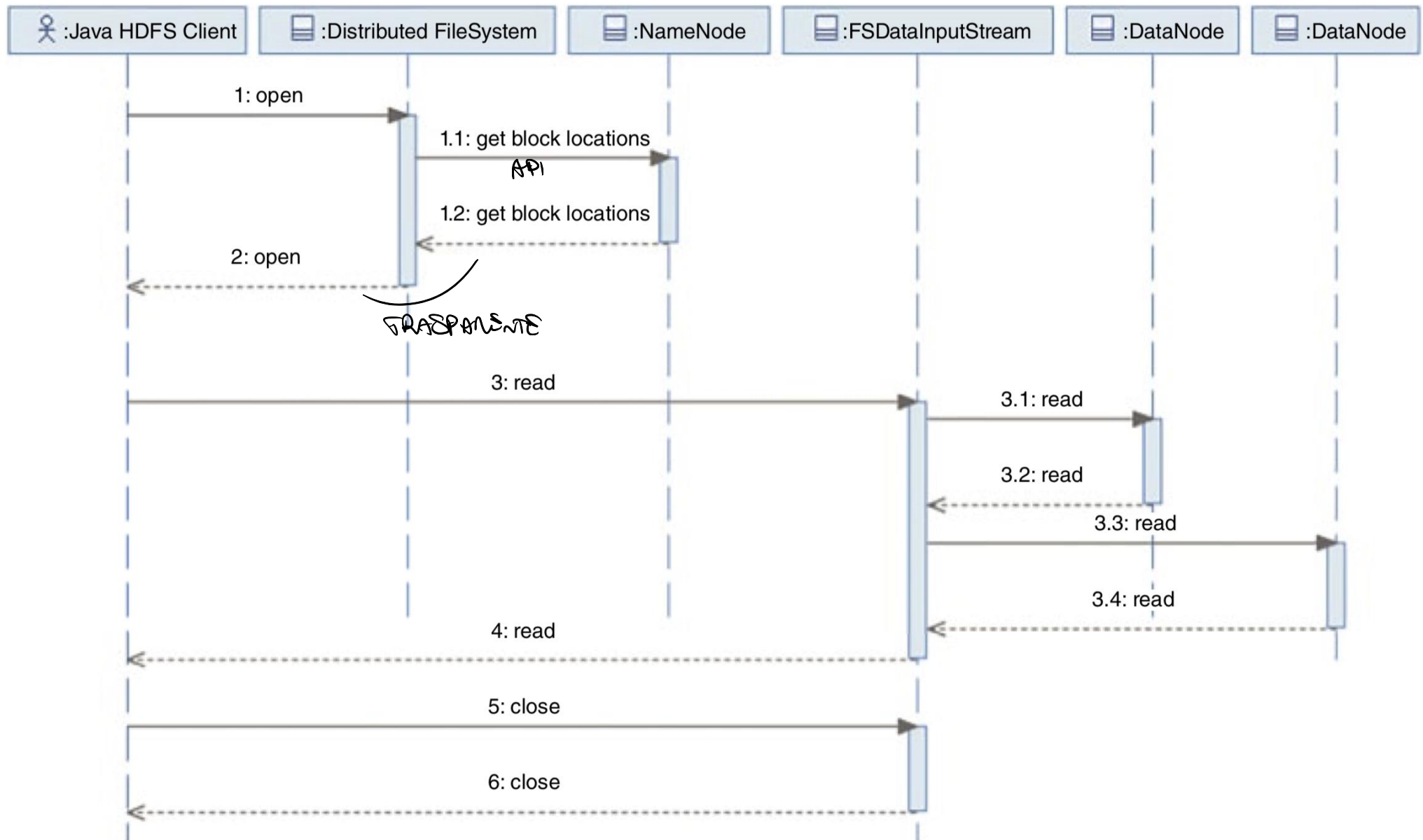
Namenodes & Datanodes

- Master/slave architecture
- DFS cluster consists of a single name node, a master server that manages the file system namespace and regulates access to files by clients.
 - Metadata
 - Directory structure → *ARTICLE / SUB-ARTICLE*
 - File-to-block mapping → *ARTICLE / BLOCKS*
 - Location of blocks → *PHYSICAL LOCATION OF*
 - Access permissions → *PERMISSION*
- There are a number of data nodes, usually one per node in a cluster.
 - A file is split into one or more blocks and set of blocks are stored in data nodes.
 - The data nodes manage storage attached to the nodes that they run on.
 - Data nodes serve read, write requests, perform block creation, deletion, and replication upon instruction from name node.
→ *SOO QWAKO A NAME NODE ORDINA DIFANG*

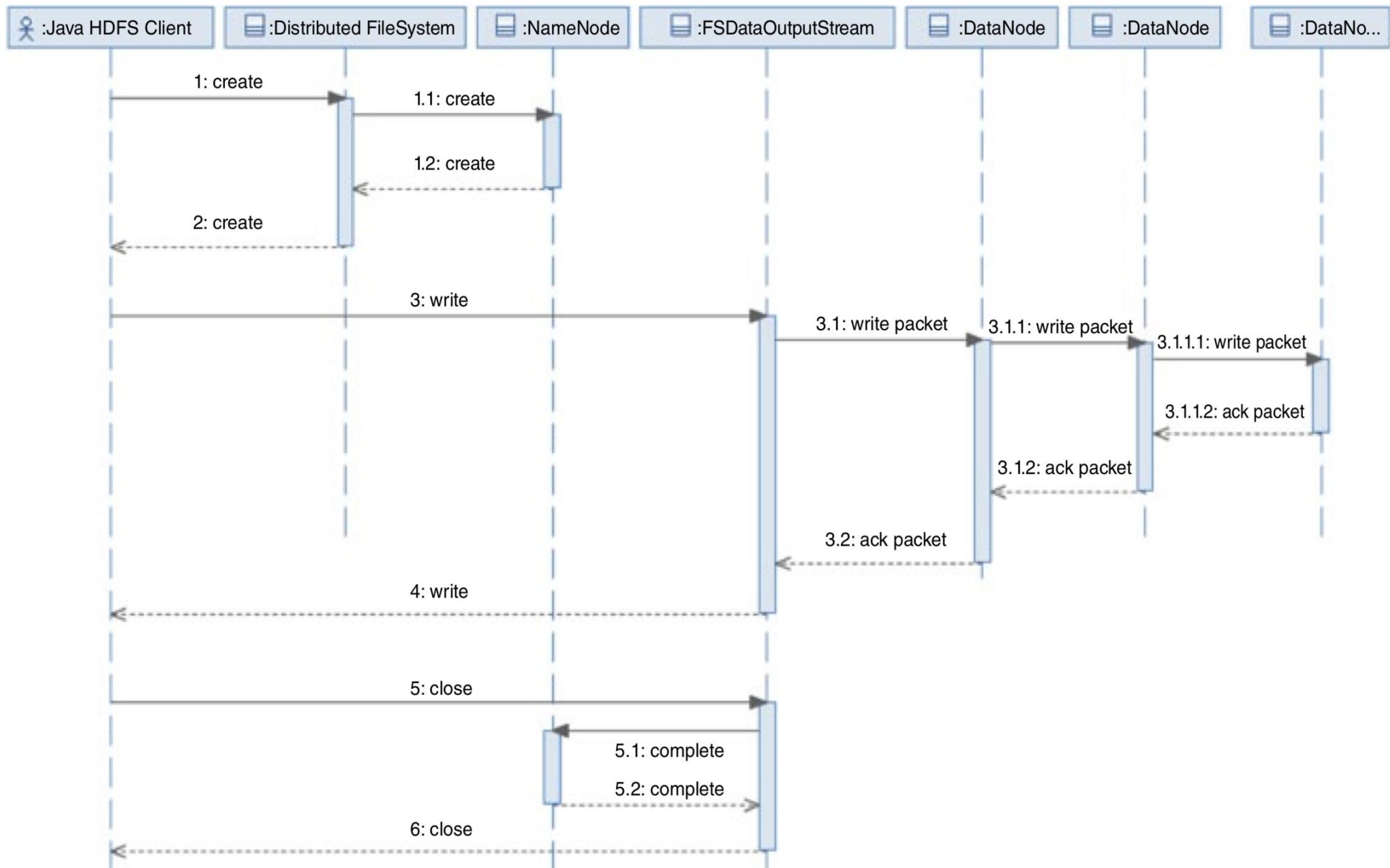
HDFS Architecture



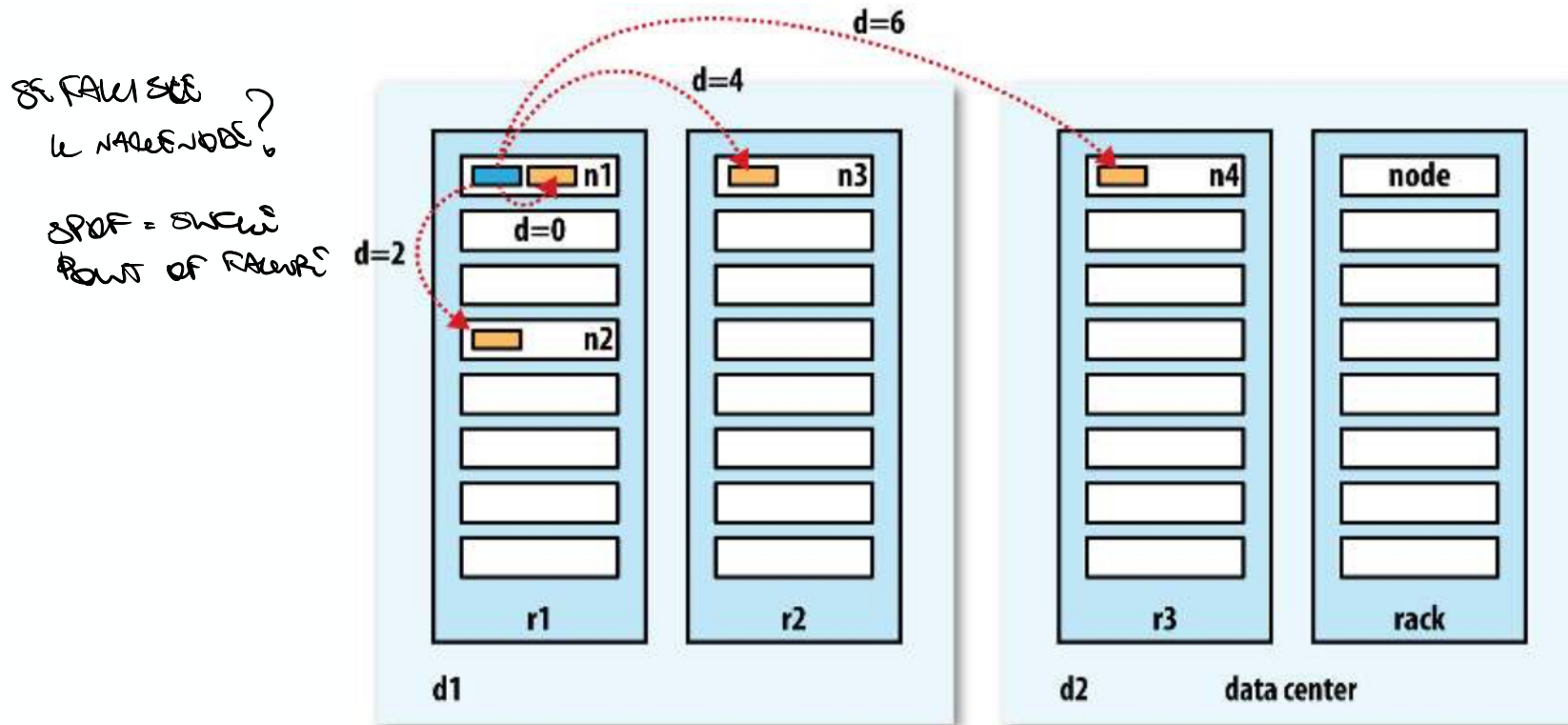
Anatomy of a read



Anatomy of a write



Network Topology and Hadoop



- **First replica** on the same node as the client
 - For clients running outside the cluster, a node is chosen at random
- **Second replica** on a different rack from the first, chosen at random
- **Third replica** on the same rack as the second, but on a different node chosen at random
- **Further replicas** are placed on random nodes in the cluster
- The system always tries to avoid placing too many replicas on the same rack/node

Hadoop Distributed Resource Management

Hadoop 1.0

Ad-hoc DISTRIBUTION SCHEDULER
FOR HADOOP

- **Job**

- Unit of work that the client wants to be performed → UNA APPROPRIATE MAP-REDUCE

- **Task**

- Unit of work that Hadoop schedules and runs on nodes in the cluster (map & reduce)

- **Slot**

- Processing element for tasks (map & reduce)

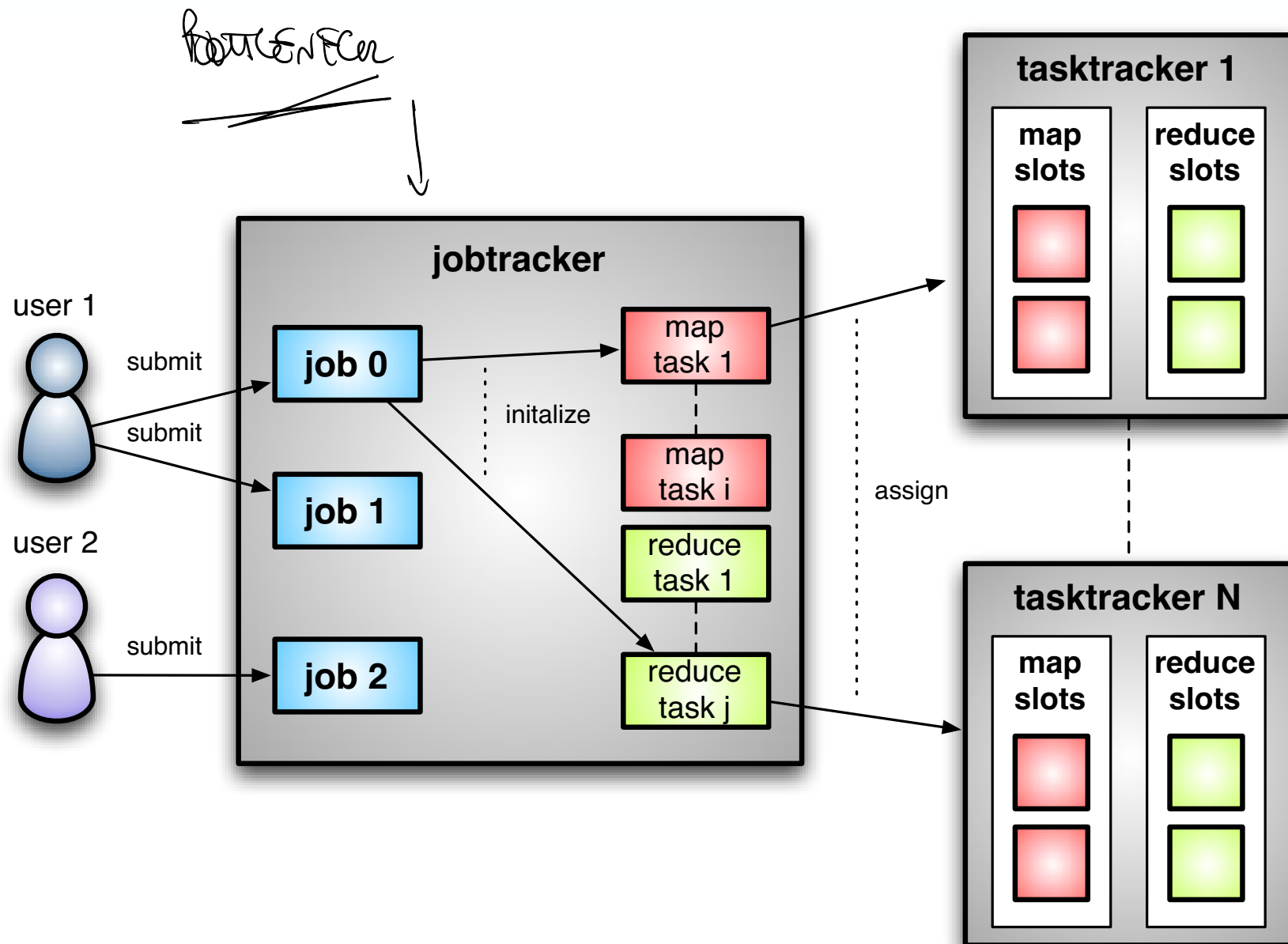
- **Job Tracker** ~ NAME-NODE

- Accepts jobs submitted by users
- Creates tasks
- Assigns map and reduce tasks to Task Trackers
- Monitors tasks and Task Trackers status, re-executes tasks upon failure

- **Task Tracker**

- Runs map and reduce tasks upon instruction from the Job Tracker
- Manages storage and transmission of intermediate output

Hadoop 1.0

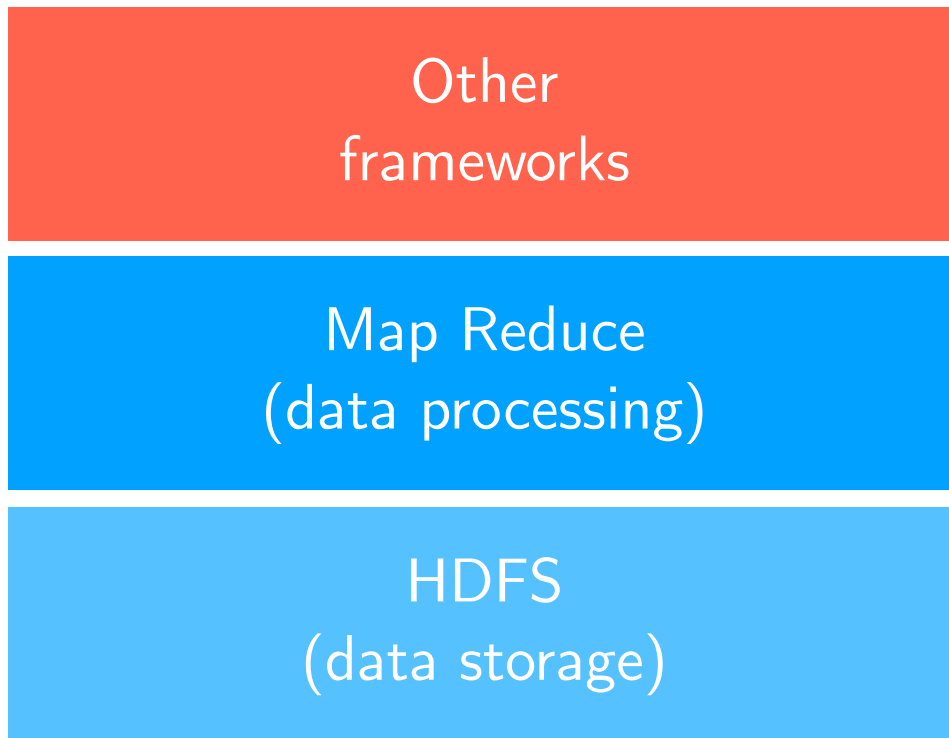


Hadoop 1.0 Limitations

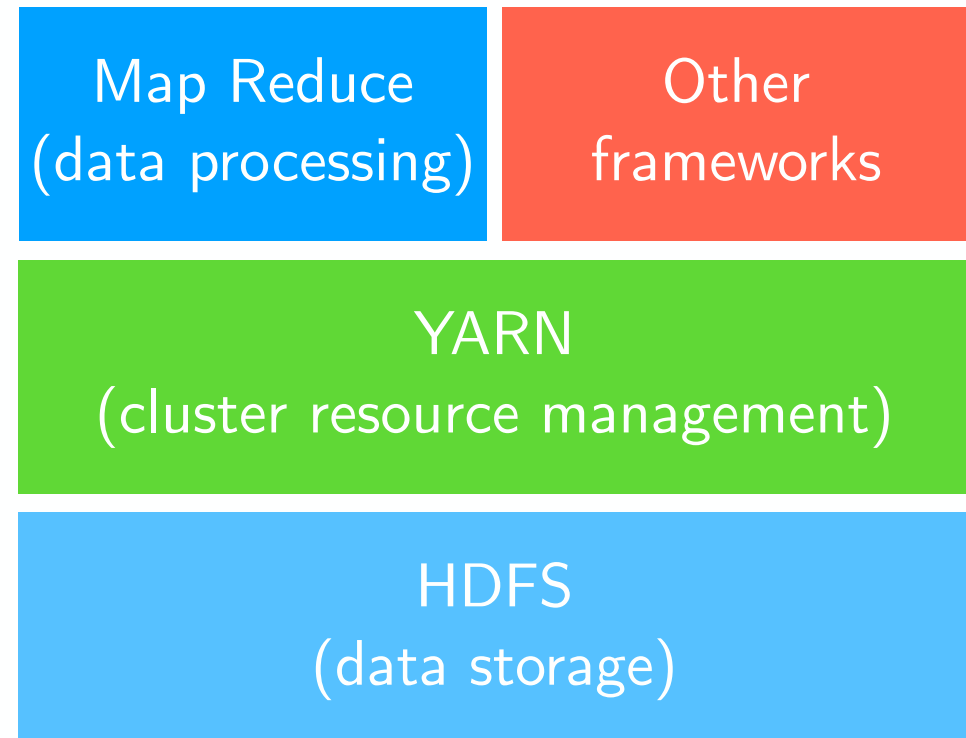
- Scalability
 - Due to the single job tracker, it becomes a **bottleneck**
 - No more than 4'000 nodes and 40'000 concurrent tasks → *COMMON HARDWARE*
TASK TRACKER
- Availability
 - Job tracker is a **single point of failure**
 - Any failure kills all queued and running jobs
 - Jobs need to be resubmitted by the users
- Resource Utilization
 - Due to the **predefined number** of map and reduce slots for each task tracker, utilization issues occur
- Limitations in running non-Hadoop applications
 - Problems in performing real-time analysis and running ad-hoc queries as Hadoop is **batch-driven**

Need of YARN

Hadoop 1.0



Hadoop 2.0, 3.0



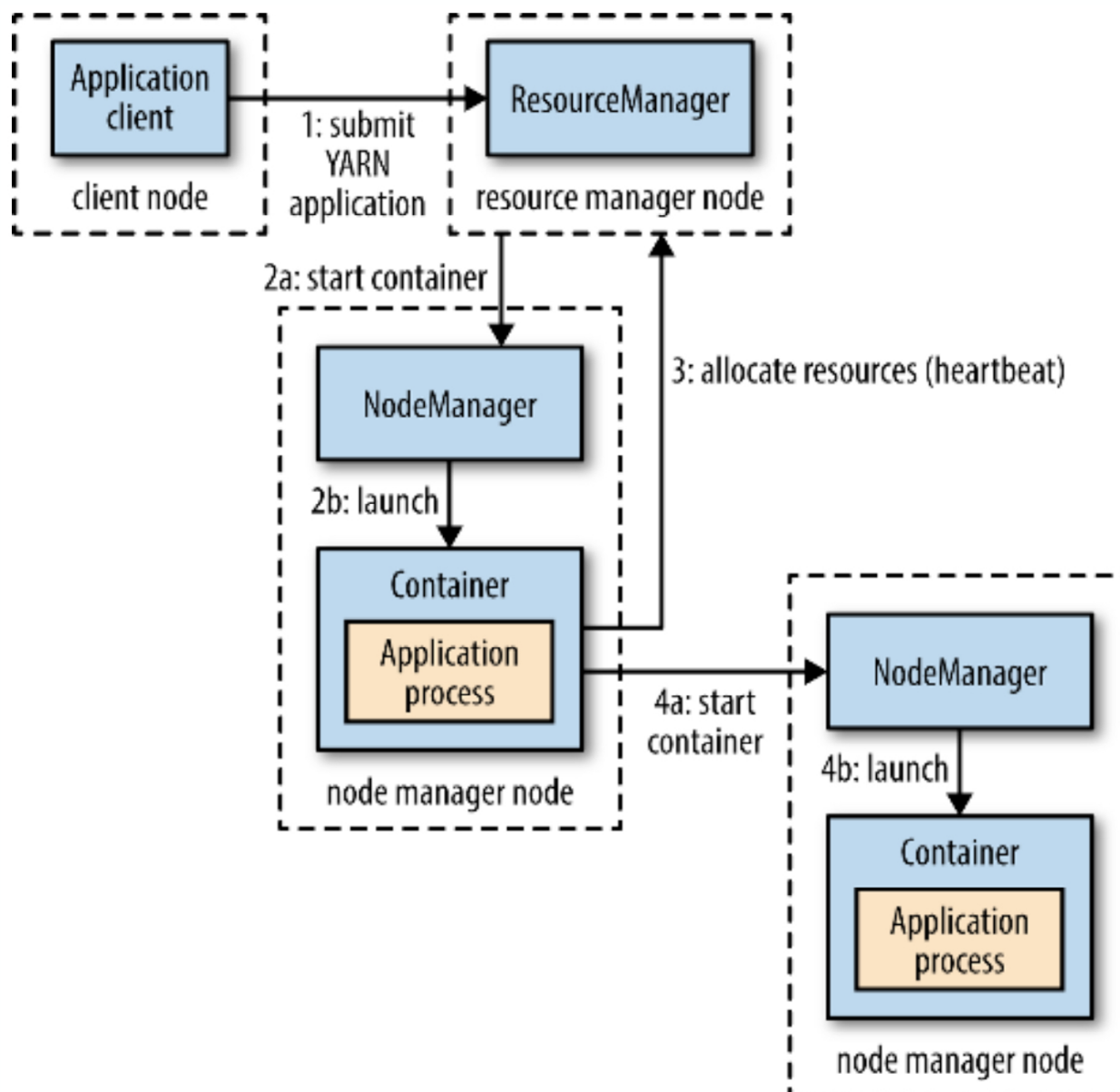
YARN Benefits

- **Scalability**
 - Cluster size of more than 10'000 nodes
 - More than 100'000 concurrent jobs
- **Compatibility**
 - Applications developed for Hadoop 1.0 runs on YARN *Back - compatibility*
 - No disruption/availability issues
- **Resource Utilization**
 - Dynamic allocation of cluster resources
 - Improved use of resources
- **Multi-tenancy**
 - Can use open-source and proprietary data access engines
 - Can perform real-time analysis and execute ad-hoc queries
 - Is used in many more distributed frameworks (e.g., Spark)

YARN Components

- YARN provides its core services via two types of long-running daemon
 - a **resource manager** (one per cluster) to manage the use of resources across the cluster
 - **node managers** running (one per node in the cluster) to launch and monitor containers
- A **container** executes an application-specific process (a.k.a. **application master**) with a constrained set of resources (memory, CPU, and so on)
- A **resource request** for a set of containers can express
 - the amount of computer resources required for each container (memory and CPU)
 - locality constraints for the containers in that request.
- If the **locality constraint** cannot be met
 - no allocation is made or
 - the constraint can be loosened
- YARN itself does not provide any **communication mechanism** for the parts of the application

Anatomy of a YARN application run



YARN Applications

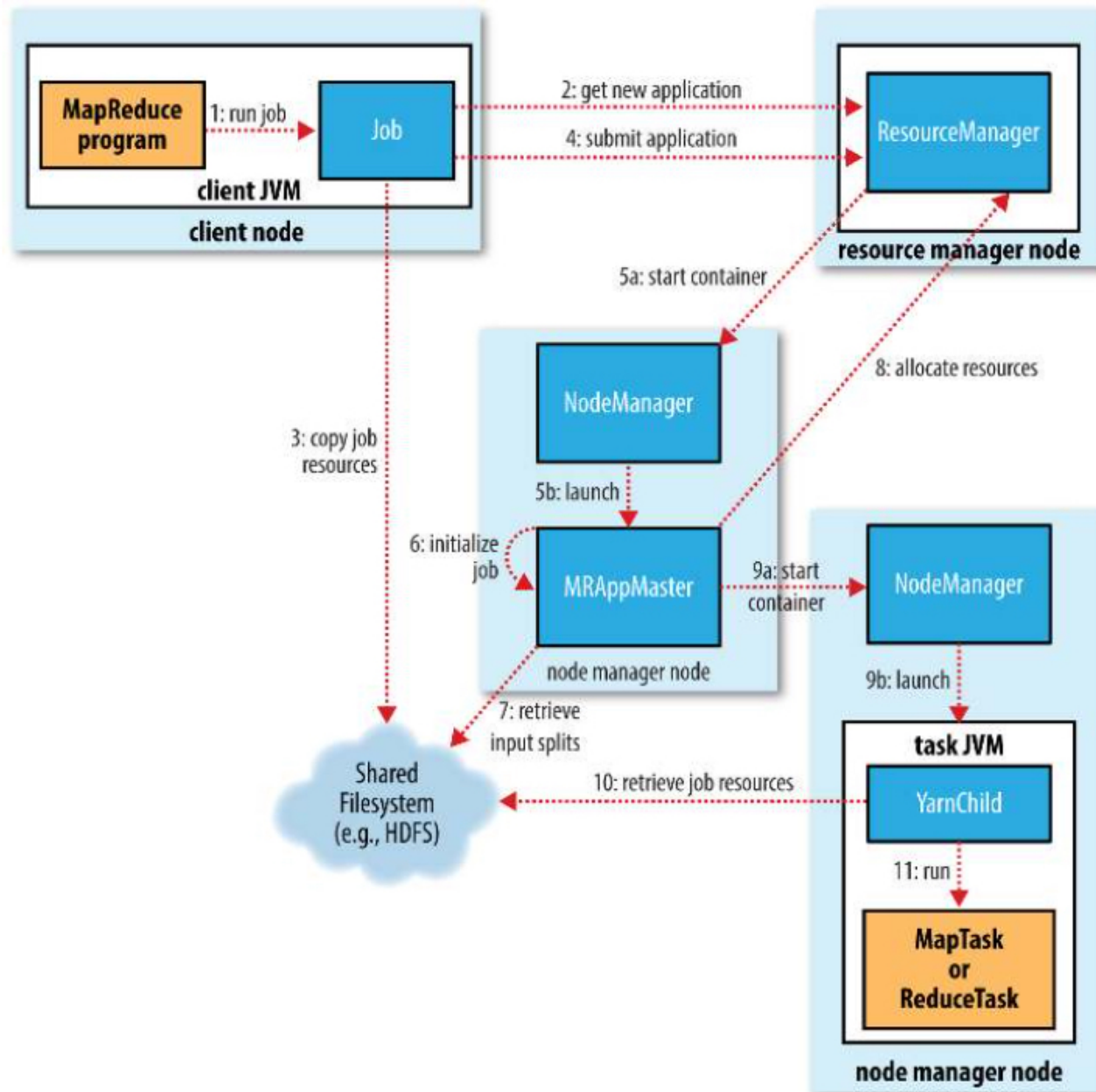
The lifespan of a YARN applications can be categorized in terms of how they map to the jobs that users run:

- First: one application per user job, e.g., MapReduce jobs
- Second: one application per workflow or user session of (possibly unrelated) jobs., e.g., Spark jobs
 - This approach can be more efficient than the first, since containers can be reused between jobs
 - There is also the potential to cache intermediate data between jobs.
- Third: long-running application that is shared by different users.
 - Such an application often acts in some kind of coordination role
 - For example, a long-running application master for launching other applications on the cluster
 - The “always on” application master means that users have very low latency

YARN Scheduling

- YARN provides a **choice of configurable schedulers**
 - Scheduling in general is a **NP-Hard** problem and there is no one “best” policy
- The **FIFO Scheduler** places applications in a **single queue** and runs them in the order of submission (first in, first out)
 - Simple to understand and no configuration needed
 - Not suitable for shared clusters. Large applications will use all the resources in a cluster, so each application has to wait its turn
- The **Capacity Scheduler** has separate **dedicated queues**, each configured to use a given fraction of the cluster capacity
 - Within a queue, applications are scheduled using FIFO scheduling
- The **Fair Scheduler** will dynamically balance resources between all running jobs
 - There is no need to reserve a set amount of capacity
- **Delay scheduling** is supported by both the Capacity Scheduler and the Fair Scheduler
 - In practice, waiting a short time (no more than a few seconds) can dramatically increase the chances of being allocated a container on the requested node, and therefore increase the efficiency of the cluster
- Every node manager in a YARN cluster periodically sends a **heartbeat request** to the resource manager
 - Heartbeats carry information about the node manager’s running containers and the resources available for new containers
 - Each heartbeat is a potential **scheduling opportunity** for an application to run a container

Anatomy of a MapReduce application run



Fault Tolerance (I)

- **Task Failure**

- The user code in the map or reduce task throws a **runtime exception**
 - the task JVM reports the error back to its parent application master before it exits.
 - The application master marks the task attempt as failed
 - The application master frees up the container
- **Sudden exit** of the task JVM
 - the node manager informs the application master
 - The application master marks the task attempt as failed
 - The application master frees up the container
- **Hanging tasks**
 - The application master notices that it hasn't received a progress update for a while
 - The task JVM process will be killed automatically after this period
 - The application master marks the task as failed
- When the application master is notified of a task attempt that has failed, it will reschedule execution of the task
- The application master will try to avoid rescheduling the task on a node manager where it has previously failed

Fault Tolerance (II)

- **Application Master Failure**

- The resource manager will detect the failure and start a new instance of the master running in a new container (managed by a node manager)
- The client needs to go back to the resource manager to ask for the new application master's address (this process is transparent to the user)
- If a MapReduce application master fails **twice** it will not be tried again and the job will fail

- **Node Manager Failure**

- The resource manager will notice a node manager that has stopped sending heartbeats if it hasn't received one for 10 minutes
- The resource manager will remove it from its pool of nodes to schedule containers on
- Any task or application master running on the failed node manager will be considered failed
- Node managers may be blacklisted if the number of failures for the application is high
- **Blacklisting** is done by the application master

Fault Tolerance (III)

- **Resource Manager Failure**

- In the default configuration, the resource manager is a **single point of failure**
- To achieve **high availability**, it is necessary to run a pair of resource managers in an **active-standby** configuration
- Information about all the running applications is stored in a highly available state store
- The standby resource manager can recover the core state of the failed active resource manager
- The transition of a resource manager from standby to active is handled by a **failover controller**
- The default failover controller uses **leader election** to ensure that there is only a single active resource manager at one time

Speculative Execution

- Problem: **Stragglers** (i.e., slow workers) significantly lengthen the completion time
 - Other jobs may be consuming resources on machine
 - Bad disks with soft (i.e., correctable) errors transfer data very slowly
 - Other weird things: processor caches disabled at machine init
- Solution: Close to completion, **spawn backup copies** of the remaining in-progress tasks.
 - Whichever one finishes first, “wins”
- Additional cost: a few percent more resource usage
- Example: A sort program without backup = 44% longer

Shuffle and Sort

