Introduction to HDFS:
Hadoop Distributed File System

What’s Hadoop

- **Hadoop** is an open-source framework that is used to
  - efficiently **store large datasets** ranging in size from gigabytes to petabytes of data.
  - efficiently **process large datasets**
- **HDFS** - Hadoop DFS is **a distributed file system that handles large data sets running on commodity hardware**.
- Hadoop implements MapReduce paradigm

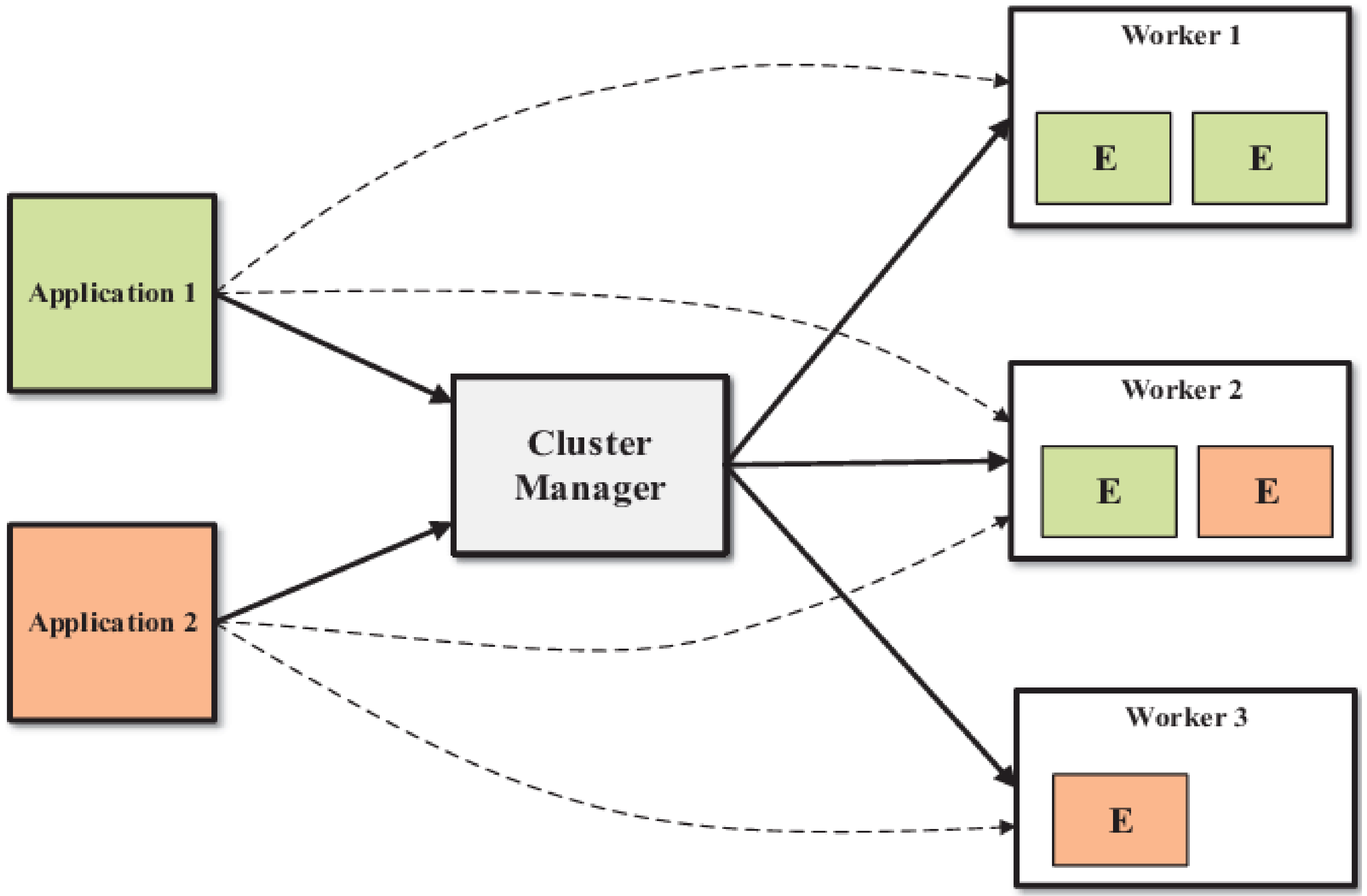
What’s Hadoop Functionality

- Hadoop implements MapReduce paradigm:
  - map(): extract some useful information from each record
  - reduce(): aggregates values for a given key
  - combine():

MapReduce Cluster

- One (or more) master node
  - can have more than one master
- Set of worker nodes (10’s, 100’s, 1000’s)
- Example: cluster of 102 nodes
  - One master node (another one: standby)
  - 100 worker nodes
- Question: What if master node crashes
- Question: What if some worker node crashes

MapReduce Cluster
E = Executor service



MapReduce Motivation

- Process lots of data
  - Google processed about 24 petabytes of data per day in 2009.
  - Facebook processed 60 petabytes of data per day in 2020
- **A single machine** cannot serve all the data
  - You need a distributed system to store and process **in parallel**
- Parallel programming?
  - **Threading** is hard!
  - How do you facilitate communication between nodes?
  - How do you **scale to more machines**?
  - How do you handle machine failures

What’s HDFS

- HDFS is a distributed file system that is
  - Fault tolerant
  - Scalable
  - Easy to expand
  - Used in MapReduce
- HDFS is the primary distributed storage for Hadoop applications.
- HDFS provides interfaces for applications to move themselves closer to data

HDFS Overview -1

- Purpose of HDFS:
- Designed for Storing Very Large Files:
    - Files in the range of hundreds of megabytes, gigabytes, or terabytes.
    - Hadoop clusters can store petabytes of data.
- Key Characteristics:
- Streaming Data Access:
    - Optimized for write-once, read-many-times patterns.
    - Suitable for processing large datasets where reading the entire dataset is crucial.
    - Prioritizes throughput over the latency of accessing the first record.
- Commodity Hardware:
    - Operates on clusters of commonly available hardware from multiple vendors.
    - Designed to handle node failures without interrupting user operations.

HDFS Overview -2

- Applications Not Suited for HDFS:
- Low-Latency Data Access:
  - HDFS is not optimized for applications requiring latency in the tens of milliseconds.
  - HBase is a better option for low-latency access.
- Handling Lots of Small Files:
  - Namenode holds filesystem metadata in memory.
  - Limit to the number of files is determined by Namenode memory.
  - Memory Usage Example:
    - Each file, directory, and block uses about 150 bytes of memory.
    - One million files (each with one block) would require at least 300 MB of memory.
    - Storing billions of files is beyond current hardware capabilities.

HDFS Overview -3

- File Writing in HDFS:
- Single Writer, Append-Only:
  - Files can only be written by a single writer.
  - Writes are made at the end of the file in an append-only fashion.
  - No Support for:
    - Multiple writers.
    - Modifications at arbitrary offsets within the file.
  - Future support for these features may be inefficient.

HDFS Concepts

Blocks:

A disk block is the minimum amount of data a disk can read or write.

Filesystems for a single disk handle data in blocks, which are multiples of the disk block size.

Filesystem blocks are typically a few kilobytes in size, whereas disk blocks are normally 512 bytes.

Filesystem block size is usually transparent to users reading or writing files.

Maintenance tools like “df” and “fsck” operate at the filesystem block level.

HDFS Blocks:

HDFS also uses the concept of blocks, but they are much larger (128 MB by default).

Files in HDFS are broken into block-sized chunks, stored as independent units.

Unlike single-disk filesystems, a file in HDFS smaller than a block does not occupy a full block's worth of storage.

Example: A 1 MB file with a 128 MB block size only uses 1 MB of disk space.

Benefits of Block Abstraction in Distributed Filesystems

- File Size Flexibility:
  - A file can be larger than any single disk in the network.
  - Blocks from a file can be stored on different disks in the cluster.
  - Possible to store a single file across all disks in an HDFS cluster.
- Simplified Storage Subsystem:
  - Blocks, not files, are the unit of abstraction, simplifying storage management.
  - Fixed block size makes it easier to calculate disk storage capacity.
  - Blocks eliminate metadata concerns; file metadata is managed separately.
- Fault Tolerance and Availability:
  - Blocks support replication to ensure fault tolerance.
  - Each block is typically replicated to three separate machines.
  - If a block becomes unavailable, a copy is read from another location, transparent to the client.
  - Corrupted or unavailable blocks can be replicated from other locations to maintain the replication factor.
  - High replication factors can be set for popular files to distribute the read load across the cluster.

Large Block Size

Large HDFS Blocks vs. Disk Blocks:

HDFS blocks are larger than disk blocks to minimize the cost of seeks.

Purpose of Large Blocks:

If blocks are large enough, data transfer time from the disk can be longer than the seek time.

Transferring large files made of multiple blocks operates at the disk transfer rate.

Calculation Example:

Seek time: ~10 ms.

Transfer rate: 100 MB/s.

To keep seek time at 1% of transfer time, block size should be around 100 MB.

Default HDFS block size is 128 MB, but some installations use larger sizes.

Block size continues to increase as disk transfer speeds improve.

Caution:

Map tasks in MapReduce usually operate on one block at a time.

Having too few tasks (fewer than nodes in the cluster) can slow down job execution.

Data Size HDFS: Example

- Cluster: 15 Nodes
  - One Master (does not store actual data, stores metadata)
  - 14 worker nodes
  - Each worker node can store: 100TB
- Total size = 14 x 100TB = 1400TB
- HDFS can utilize 1400 TB of disk

Components of HDFS

Namenode (Master):

Manages the filesystem namespace.

Maintains the filesystem tree and metadata for all files and directories.

Stores metadata persistently on local disk as two files: namespace image and edit log.

Knows the Datanodes where file blocks are located but does not store this persistently.

DataNodes (Workers):

Store and retrieve blocks as instructed by the Namenode or clients.

Periodically report back to the Namenode with the list of stored blocks.

Client Interaction:

Accesses the filesystem by communicating with Namenode and Datanodes.

Presents a POSIX-like filesystem interface to users.

Namenode Criticality:

Filesystem is unusable without the Namenode.

Loss of the Namenode means loss of all files, as block reconstruction would be impossible.

Namenode Resilience Mechanisms

Persistent State Backup:

Namenode writes its persistent state to multiple filesystems synchronously and atomically.

Typically writes to local disk and remote NFS mount.

Secondary Namenode:

Periodically merges namespace image with the edit log to prevent the edit log from growing too large.

Runs on a separate machine with similar CPU and memory resources as the Namenode.

Keeps a copy of the merged namespace image for use in case of Namenode failure.

In case of Namenode failure, the secondary Namenode can take over, but data loss is likely due to lag.

The usual recovery involves copying the Namenode's metadata files from NFS to the secondary Namenode and promoting it to the primary.

HDFS High Availability -1

Limitations of Traditional Namenode Setup:

Single Point of Failure (SPOF):

- Namenode is critical as it stores all metadata and file-to-block mappings.
- Failure leads to inability for clients and MapReduce jobs to access files.
- Entire Hadoop system is down until a new Namenode is brought online.

Recovery Process from Namenode Failure:

- Administrator starts a new primary Namenode using a metadata replica.
- New Namenode process:
  - Load namespace image into memory.
  - Replay edit log.
  - Receive block reports from Datanodes to exit safe mode.

Recovery can take 30 minutes or more in large clusters.

HDFS High Availability -2

Challenges with Namenode Downtime:

- Long Recovery Times:
  - Delays affect both unexpected failures and routine maintenance.
  - Planned downtime is more common but still disruptive.

Hadoop 2 and High Availability (HA):

- Active-Standby Namenode Configuration:
  - Active Namenode handles client requests.
  - Standby Namenode takes over upon failure with minimal interruption.

Architectural Changes for HA:

- Use of highly available shared storage for the edit log.
- Datanodes report block information to both Namenodes.
- Clients handle Namenode failover transparently.
- Standby Namenode replaces the secondary Namenode’s role for checkpoints.

HDFS High Availability -3

- Shared Storage Options for HA:
- NFS Filer
  - Standard option but less preferred.
- Quorum Journal Manager (QJM):
  - Dedicated HDFS implementation for high availability of the edit log.
  - Operates with a group of journal nodes (typically three).
  - Writes require a majority of nodes, allowing the system to tolerate one failure.
  - QJM vs. ZooKeeper:
    - Similar operation, but QJM does not use ZooKeeper.
    - ZooKeeper is used for electing the active namenode.
- Failover and Recovery:
- Fast Failover:
  - Standby Namenode takes over quickly (within tens of seconds).
  - Has the latest state in memory including edit log entries and block mapping.
- Cold Start in Exceptional Cases:
  - If standby is down during active Namenode failure, it can still be started from cold.
  - This process is no worse than the non-HA scenario and is built into Hadoop.

Failover and Fencing in HDFS HA -1

- Failover Controller:
- Role:
  - Manages the transition between active and standby Namenodes.
  - Ensures only one Namenode is active at any given time.
- Implementation:
  - Default uses ZooKeeper to coordinate failover.
  - Each Namenode runs a lightweight failover controller process.
  - Monitors Namenode health via heartbeating and triggers failover on failure.
- Types of Failover:
- Graceful Failover:
  - Initiated manually by an administrator (routine maintenance).
  - Allows orderly transition of roles between Namenodes.
- Ungraceful Failover:
  - Occurs when the previously active Namenode might still be running.
  - Risk of stale read requests and potential corruption.

Failover and Fencing in HDFS HA -2

- Fencing Mechanisms:
- Purpose:
  - Prevent the previously active Namenode from causing corruption during ungraceful failover.
- QJM Fencing:
  - Only allows one Namenode to write to the edit log at a time.
  - Recommended to use an **SSH fencing command** to kill the old Namenode's process.
- NFS Fencing:
  - More complex due to the lack of single-writer enforcement.
  - **Fencing Options:**
    - Revoke Namenode's access to the shared storage directory (vendor-specific NFS command).
    - Disable network port via remote management command.
    - **STONITH ("Shoot the Other Node in the Head"):**
      - Uses a specialized power distribution unit to forcibly power down the machine.

Failure in MapReduce

- Failures are norm in commodity hardware
- Worker failure
  - Detect failure via periodic heartbeats
  - Re-execute in-progress map/reduce tasks
- Master failure
  - Single point of failure; Resume from Execution Log
- Robust
  - Google's experience: lost 1600 of 1800 machines once!, but finished fine.

Fault tolerance: Handled via re-execution

- On worker failure:
  - Detect failure via periodic heartbeats
  - Re-execute completed and in-progress *map* tasks
  - Task completion committed through master

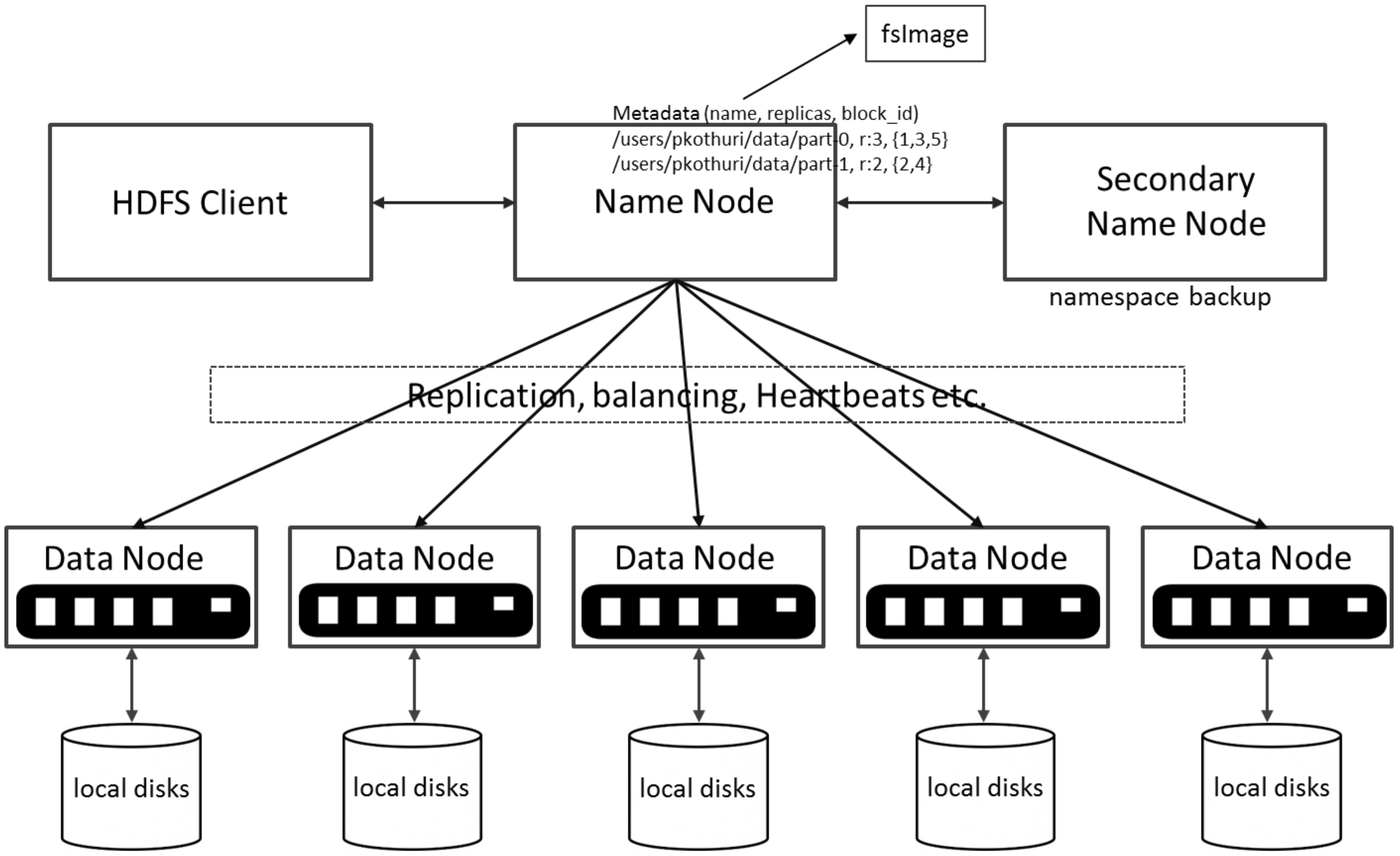
Fault tolerance

- Replication
  - High availability for reads
  - User controllable, default 3 (non-RAID)
  - Provides read/seek bandwidth
  - Master is responsible for directing re-replication if a data node dies
- Online check-summing in data nodes
  - Verified on reads

File Permissions in HDFS

- HDFS Permissions Model:
- POSIX-like Model:
  - Three types of permissions:
    - **Read (r):** Required to read files or list directory contents.
    - **Write (w):** Required to write files, create/delete files or directories.
    - **Execute (x):** Ignored for files, required to access directory’s children.
- File and Directory Attributes:
  - **Owner:** User who owns the file/directory.
  - **Group:** Group associated with the file/directory.
  - **Mode:**
    - Permissions for the owner.
    - Permissions for group members.
    - Permissions for others (neither owner nor group members).

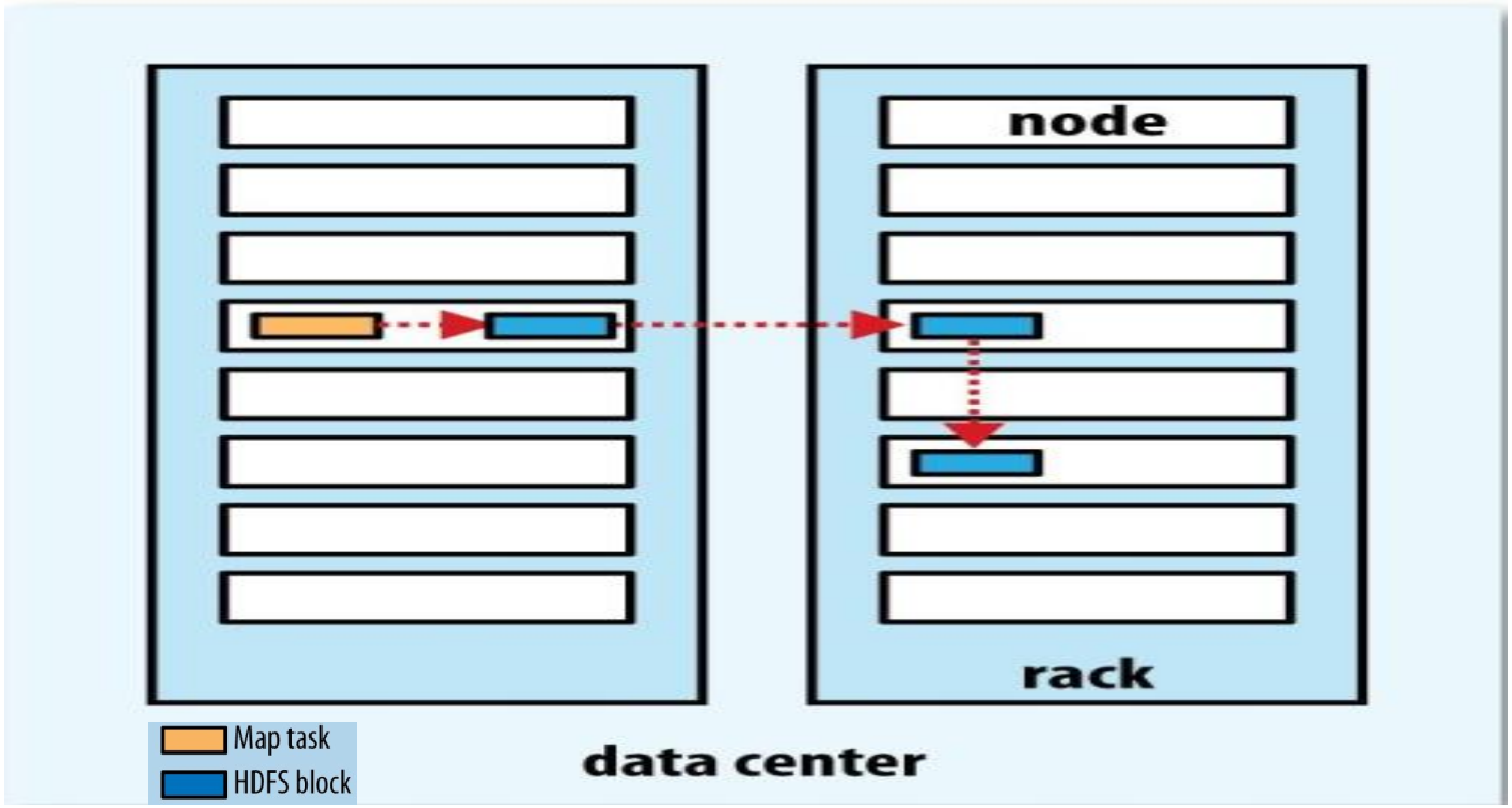
HDFS Architecture



HDFS – Data Organization

- Each file written into HDFS is split into **data blocks**
- Each block is stored on one or more data nodes
- Each copy of the block is called replica
- Block placement policy
  - First replica is placed on the local node
  - Second replica is placed in a different rack
  - Third replica is placed in the same rack as the second replica

HDFS – Data Organization



HDFS – Example: Single File

- Let data block size = 128 MB
- **Let a file (sample.txt) = 400 MB**
- Therefore, we need 4 data blocks to store file in HDFS:
- 400 = 128 + 128 + 128 + 16
- **B1** = Block1 = 128 MB
- **B2** = Block2 = 128 MB
- **B3** = Block3 = 128 MB
- **B4** = Block3 = 16 MB

HDFS – Example

- Consider a cluster of 6 nodes: {M, N1, N2, N3, N4, N5}
- 1 master + 5 data nodes – and master does not store any data at all (maste*r node just stores metadata)*
- M: Master = stores metadata only
- N1 = data node 1
- N2 = data node 2
- N3 = data node 3
- N4 = data node 4
- N5 = data node 5

HDFS – Replication of 1

File sample.txt has 4 blocks: B1, B2, B3, B4

Each block is replicated ONCE

- N1 = {B2}
- N2 = {B3}
- N3 = {B1}
- N4 = {B4}
- N5 = {}

- QUESTION: what happens if any of data nodes fail?

HDFS – Replication of 2

File sample.txt has 4 blocks: { B1, B2, B3, B4 }

Each block is replicated 2 times

- N1 = {B1}
- N2 = {B3, B4}
- N3 = {B1, B2}
- N4 = {B4, B3}
- N5 = {B2}
- QUESTION: what happens if any of data nodes fail?

HDFS – Replication of 3

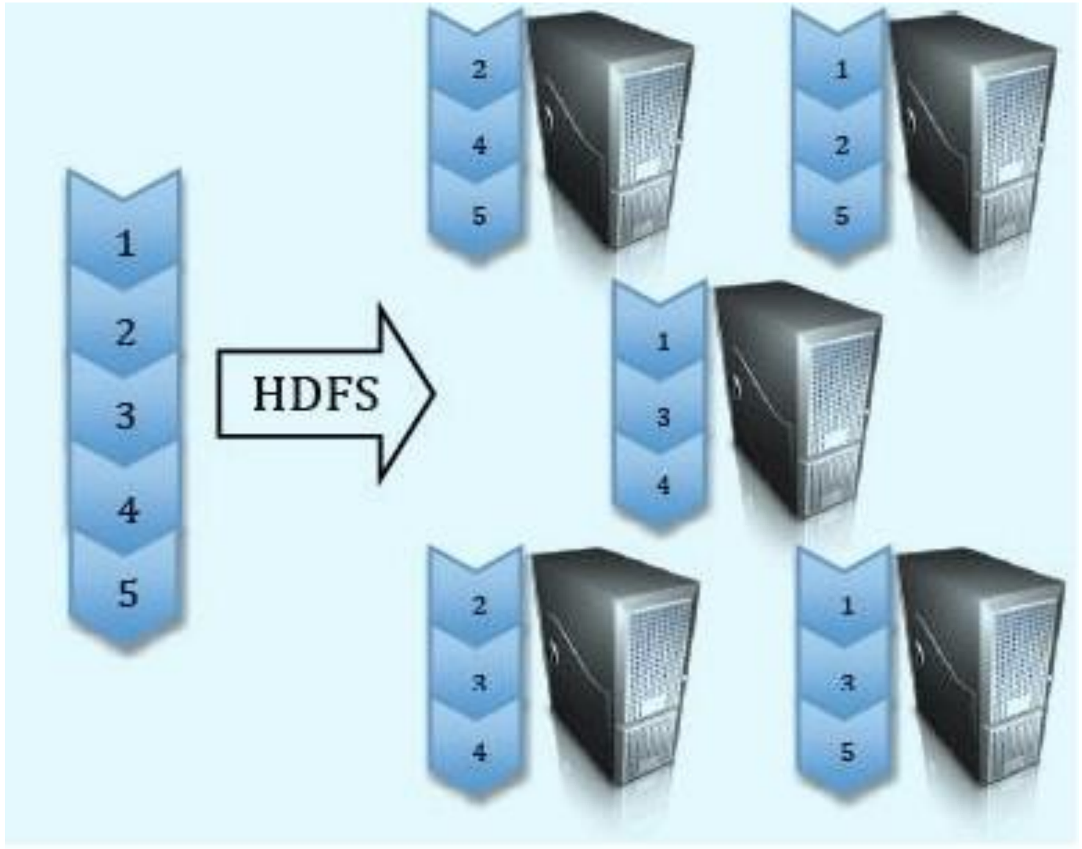
File sample.txt has 4 blocks: B1, B2, B3, B4

Each block is replicated 3 times

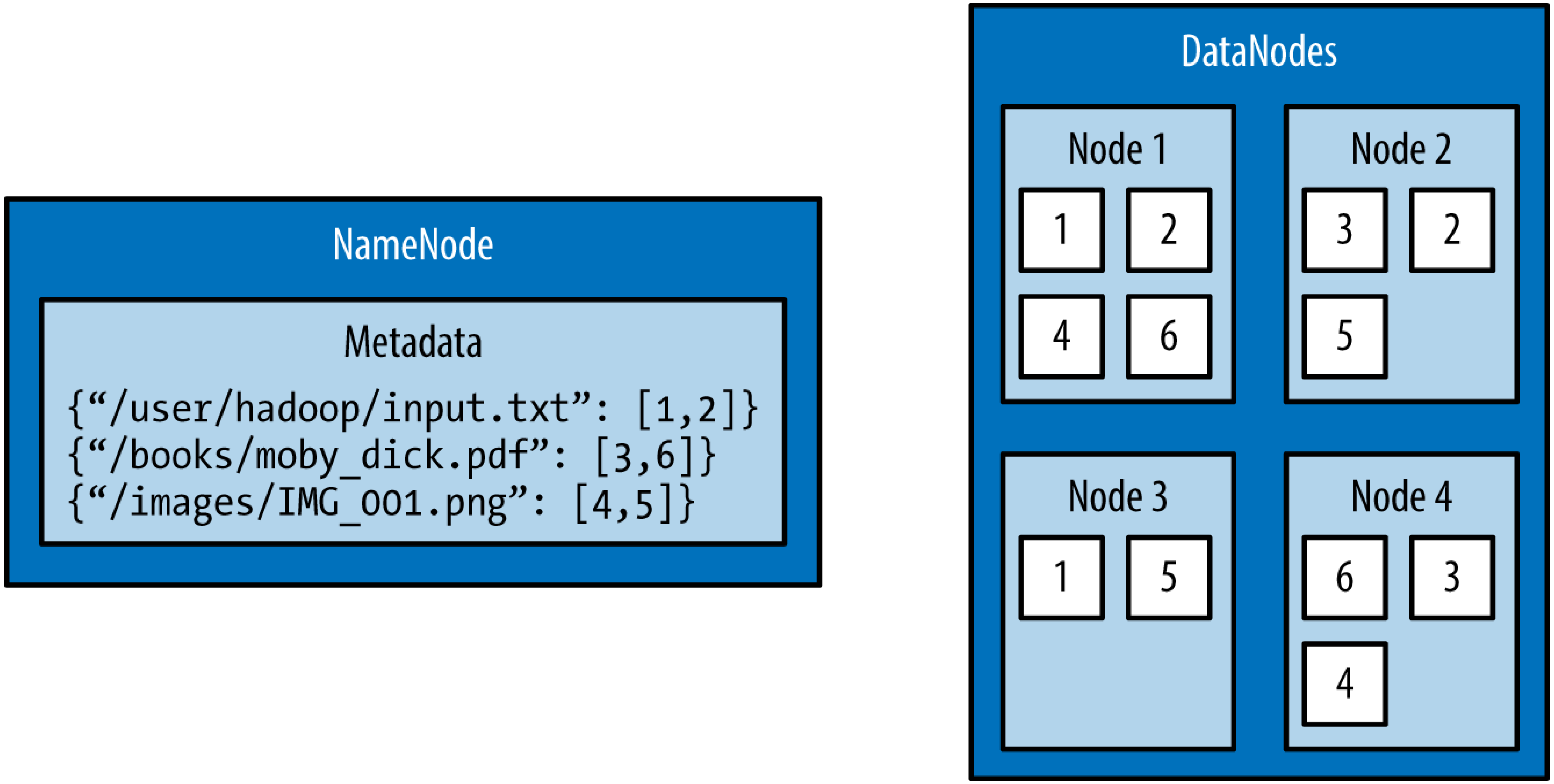
- N1 = {B1, B2}
- N2 = {B1, B3, B4}
- N3 = {B1, B2, B3}
- N4 = {B4, B3}
- N5 = {B2, B4}

- QUESTION: what happens if any of data nodes fail?

Data Replication Factor = 3

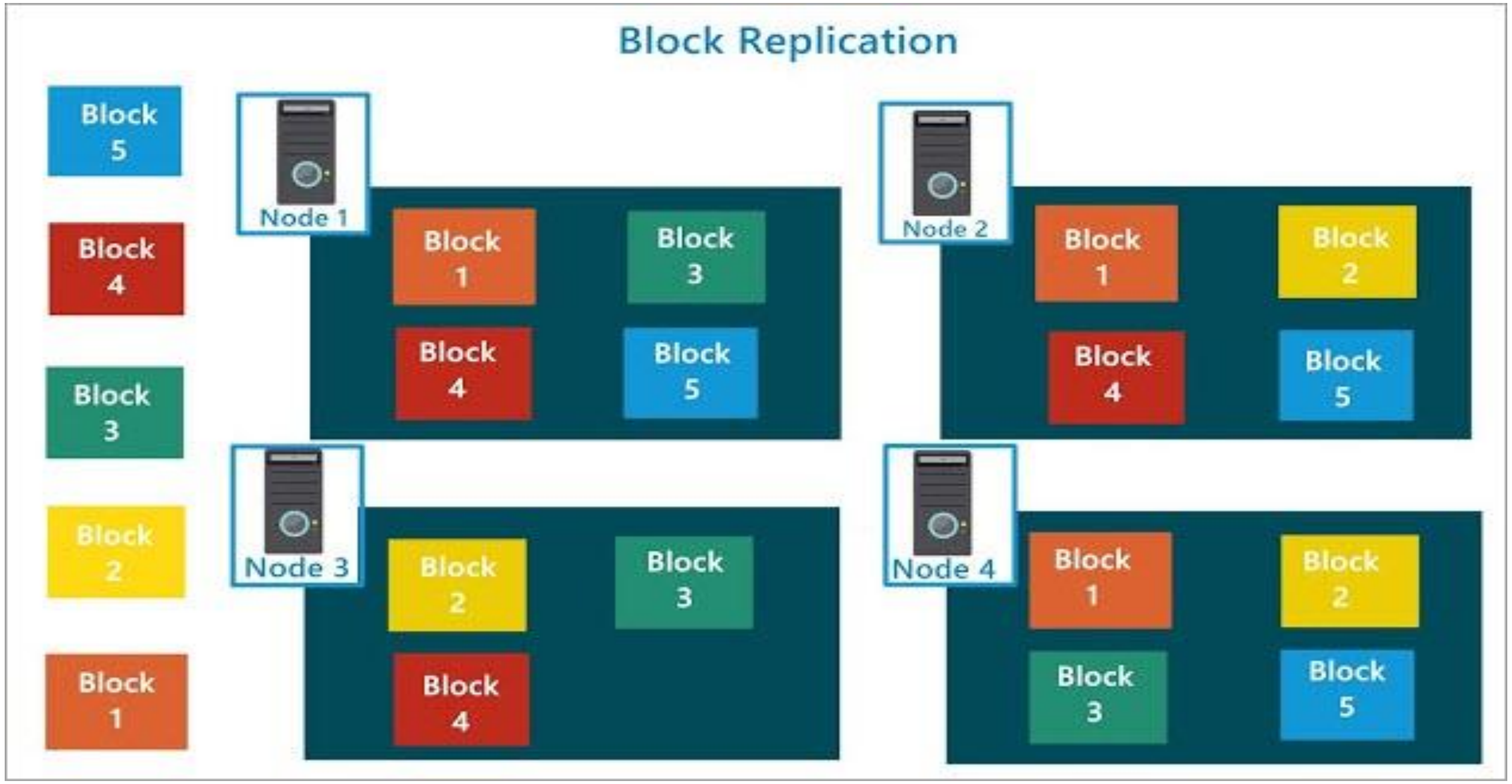


HDFS – Example: Multiple Files : 3 files Replication Factor = 2



```
{"/user/hadoop/input.txt": [1,2]}
{"/books/moby_dick.pdf": [3,6]}
{"/images/IMG_001.png": [4,5]}
```

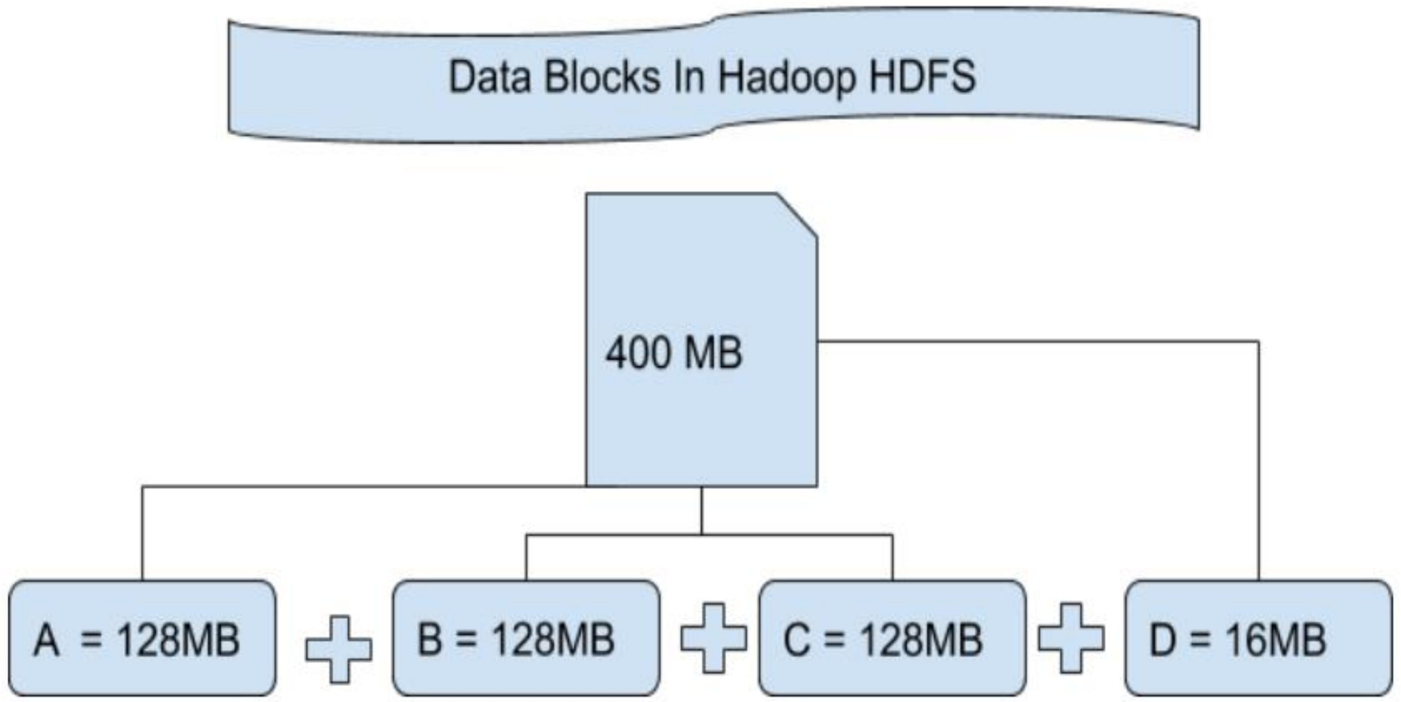
HDFS – Example: 5 blocks, 4 DataNodes Replication Factor = 3



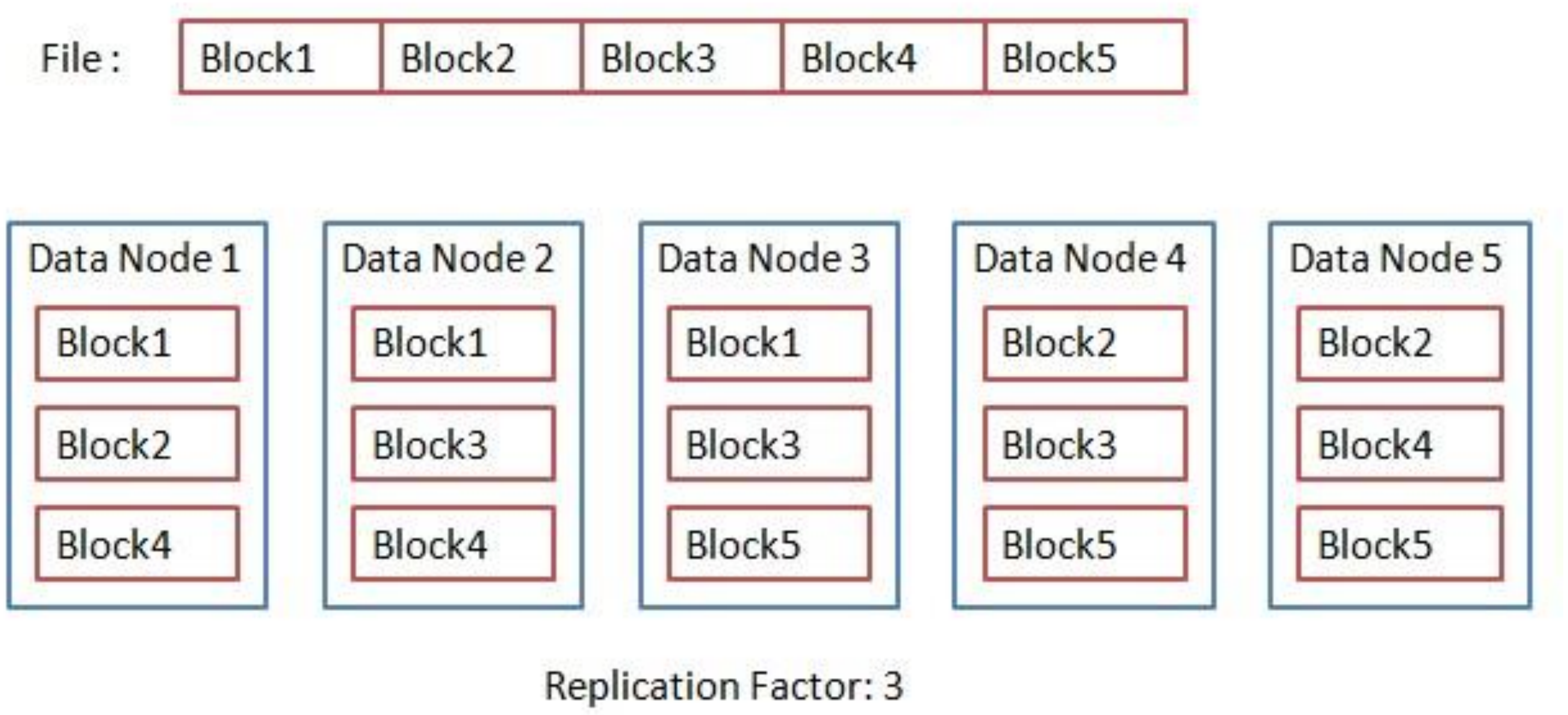
Data and Block Size

- Data Block Size = 128 MB
- File SAMPLE.TXT = 400 MB
- We need 5 blocks to store file SAMPLE.TXT:
  - A: Block1 = 128 MB
  - B: Block2 = 128 MB
  - C: Block3 = 128 MB
  - D: Block4 = 16 MB

Data and Block Size



Data Replication Factor = 3



Higher Replication means Less Usable Storage
Higher Replication means More Cost

- Consider a cluster of 5 nodes: Master + 4 DataNodes
- Assume master does not store any data other than metadata
- Assume each DataNode can store 50 TB
- With replication of 1: usable storage = 4 * 50 = 200 TB
- With replication of 2: usable storage = 200 / 2 = 100 TB
- With replication of 4: usable storage = 200 / 4 = 50 TB
- With replication of 5: usable storage = 200 / 5 = 40 TB

Cost of Replication?

- Consider a cluster of 5 nodes: Master + 4 DataNodes
- Assume each DataNode can store 50 TB
- With replication of 1: usable storage = 4 * 50 = 200 TB
- With replication of 5: usable storage = 200 / 5 = 40 TB

- With replication of 5, only 40 TB is useable → More Cost
- So the cost is 160 TB of disk space (which can not be used)

HDFS – Fault Tolerance

- How many nodes can safely fail in a cluster?
- With replication Factor of **N**,
- **N-1** nodes can safely fail

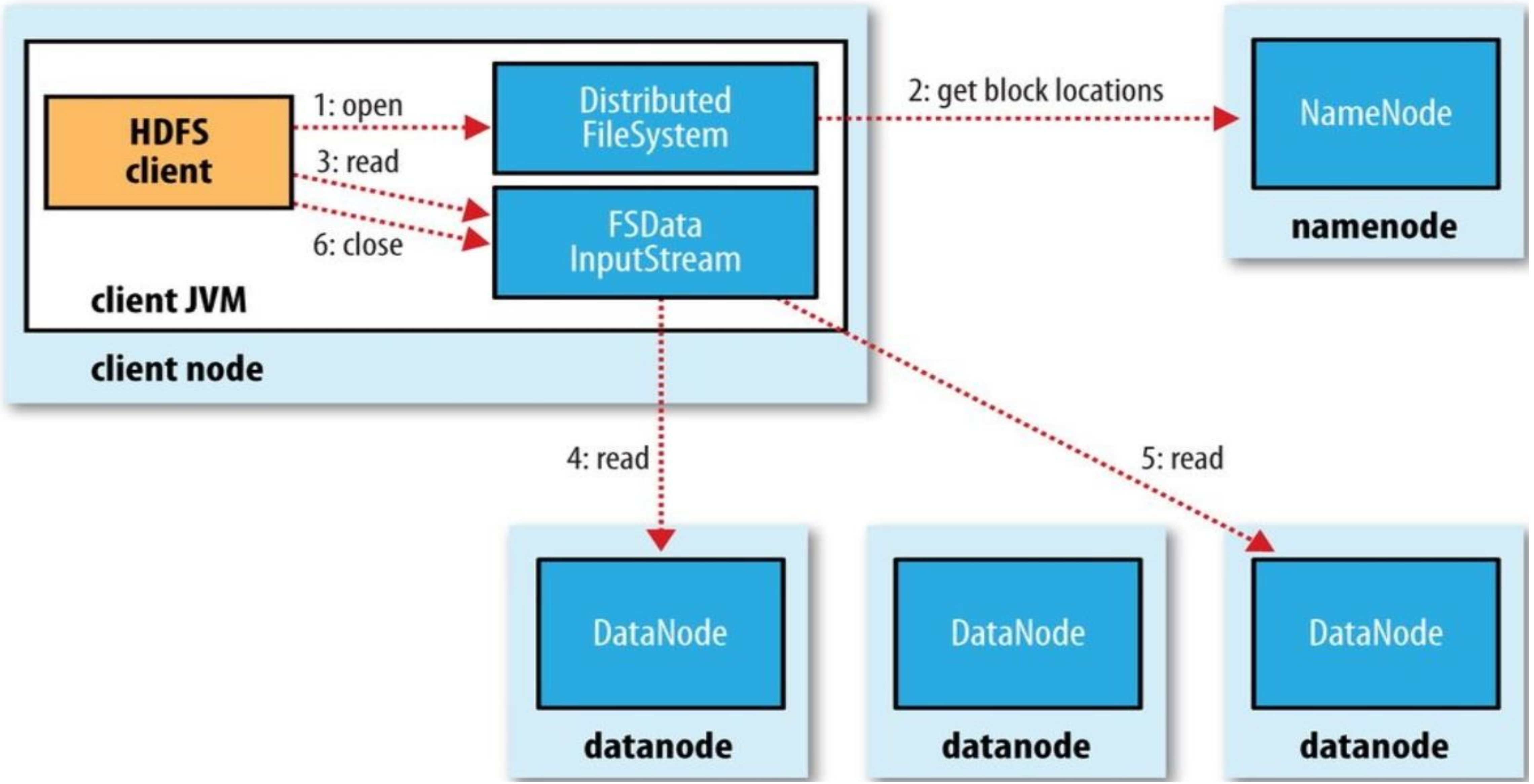
- Example:
    - So if replication factor is 3,
    - then only 2 nodes can safely fail

Fault Tolerance: N data nodes, Replication of N

- Let’s have a cluster of N+1 nodes
  - Master with no data
  - N data nodes
- With replication Factor of **N**,
- **How many** nodes can safely fail

- Answer: (N-1) nodes can safely fail: BUT VERY EXPENSIVE

Read Operation in HDFS



Anatomy of File Read in Hadoop

File Opening:

- Client opens the file using open() on the FileSystem object.
- For HDFS, this is an instance of DistributedFileSystem.
- DistributedFileSystem contacts the Namenode via RPCs to determine locations of the first few blocks (Step 1 & 2).

Namenode Response:

- Namenode returns addresses of Datanodes.
- Datanodes are sorted by proximity to the client based on network topology.
- If the client is a Datanode (e.g., in MapReduce), it reads from the local Datanode.

Stream Creation:

- DistributedFileSystem returns FSDataInputStream to the client.
- FSDataInputStream wraps DFSInputStream, which manages I/O with Datanodes and Namenode.

Anatomy of File Read in Hadoop

Data Reading Process:

- Client calls read() on the stream (Step 3).
- DFSInputStream connects to the closest Datanode for the first block.
- Data is streamed from the Datanode to the client, with repeated read() calls (Step 4).

Handling Block Transitions:

- On reaching the end of a block, DFSInputStream closes the connection.
- It then finds the best Datanode for the next block (Step 5).
- This process is transparent to the client, which perceives a continuous stream.

Block Management:

- Blocks are read sequentially with new connections opened to Datanodes as needed.
- Namenode is contacted for the next batch of block locations when necessary.

Anatomy of File Read in Hadoop

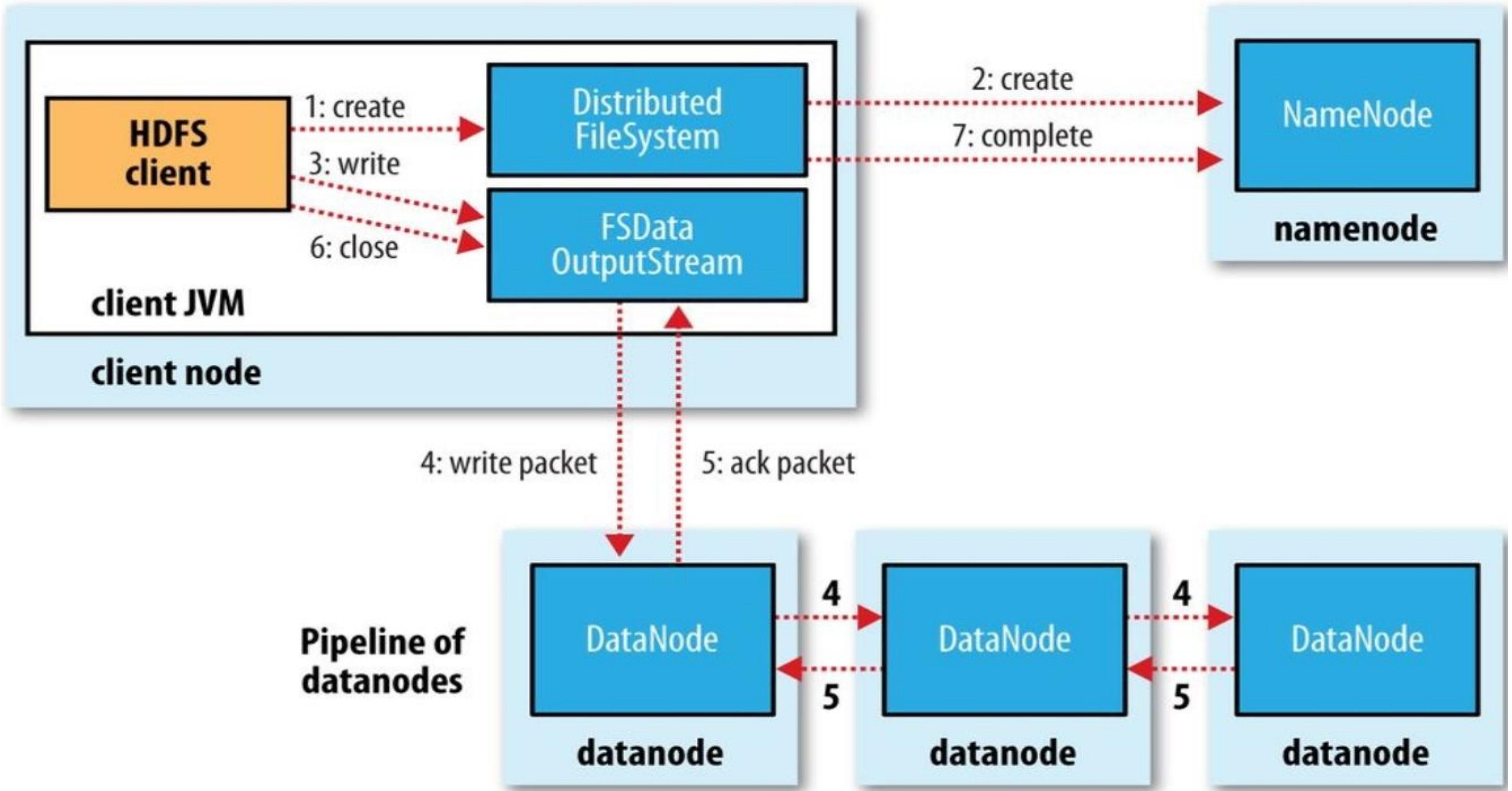
Error Handling:

- If DFSInputStream encounters an error with a Datanode, it tries the next closest one.
- Failed Datanodes are remembered to avoid retries for later blocks.
- Checksums are verified for data integrity, and corrupted blocks are reported to the Namenode.

Client-Namenode-Datanode Interaction:

- Client directly contacts Datanodes to retrieve data, guided by the Namenode.
- This design allows HDFS to scale efficiently with many concurrent clients.
- Namenode handles only block location requests, avoiding data bottlenecks.

Write Operation in HDFS



Anatomy of File Write in Hadoop

File Creation:

- Client initiates file creation using create() on DistributedFileSystem (Step 1).
- DistributedFileSystem makes an RPC call to the Namenode to create a new file in the namespace (Step 2).
- Namenode Checks:
  - Verifies the file doesn’t already exist.
  - Ensures the client has the correct permissions.
  - If checks pass, the file is recorded; otherwise, an IOException is thrown.
- An FSDataOutputStream is returned for the client to start writing data.
- FSDataOutputStream wraps DFSOutputStream, handling communication with Datanodes and the Namenode.

Anatomy of File Write in Hadoop

Data Writing Process:

- As the client writes data (Step 3), DFSOutputStream splits it into packets.
- Packets are written to an internal queue called the data queue.
- DataStreamer:
  - Consumes the data queue.
  - Requests the namenode to allocate new blocks.
  - Selects suitable datanodes to store replicas (forming a pipeline).
  - Streams packets to the first datanode, which forwards to the second, and so on (Step 4).

Acknowledgment Process:

- DFSOutputStream maintains an ack queue for packets awaiting acknowledgment from datanodes.
- A packet is removed from the ack queue only after acknowledgment by all Datanodes in the pipeline (Step 5).

Anatomy of File Write in Hadoop

Failure Handling:

- If a Datanode fails during writing, the pipeline is closed.
- Packet Management:
    - Packets in the ack queue are added to the front of the data queue.
    - Current block on good Datanodes gets a new identity, communicated to the Namenode.
- Failed datanode is removed, and a new pipeline is constructed from the remaining good Datanodes.
- The remainder of the block’s data is written to the good Datanodes.
- Namenode notices under-replicated blocks and arranges for a new replica.

Handling Multiple Failures:

- If multiple Datanodes fail, as long as ‘dfs.namenode.replication.min’ replicas are written, the write succeeds.
- The block is asynchronously replicated until it reaches the target replication factor (dfs.replication).

Anatomy of File Write in Hadoop

Completion of Writing:

- Client calls close() on the stream after finishing writing (Step 6).
- Final Steps:
  - Flushes remaining packets to the Datanode pipeline.
  - Waits for acknowledgments.
  - Contacts the Namenode to signal file completion (Step 7).
- Namenode waits for blocks to be minimally replicated before confirming completion.

HDFS Security

- Authentication to Hadoop
  - Simple – insecure way of using OS username to determine hadoop identity
  - Kerberos – authentication using kerberos ticket
  - Set by `hadoop.security.authentication=simple|kerberos`
- File and Directory permissions are same like in POSIX
  - read (r), write (w), and execute (x) permissions
  - also has an owner, group and mode
  - enabled by default `(dfs.permissions.enabled=true)`
- ACLs are used for implemention permissions that differ from natural hierarchy of users and groups
  - enabled by `dfs.namenode.acls.enabled=true`

HDFS Configuration

HDFS Defaults

- Block Size – 256 MB = `268435456 bytes`
- Replication Factor : 3
- Web UI Port : 50070

HDFS conf file - `/etc/hadoop/conf/hdfs-site.xml`

```
<property>
   <name>dfs.namenode.name.dir</name>
   <value>file:///data1/cloudera/dfs/nn,file:///data2/cloudera/dfs/nn</value>
</property>

<property>
   <name>dfs.blocksize</name><value>268435456</value>
</property>

<property>
   <name>dfs.replication</name><value>3</value>
</property>

<property>
   <name>dfs.namenode.http-address</name><value>itracXXX.cern.ch:50070</value>
</property>
```

Interfaces to HDFS

- Java API (`DistributedFileSystem`)
- C wrapper (`libhdfs`)
- HTTP protocol
- WebDAV protocol
- Shell Commands

However, the command line is one of the simplest and most familiar

HDFS – Shell Commands

There are two types of shell commands

User Commands

- `hdfs dfs` – runs filesystem commands on the HDFS
- `hdfs fsck` – runs a HDFS filesystem checking command

Administration Commands

- `hdfs dfsadmin` – runs HDFS administration commands

HDFS – User Commands (dfs)

List directory contents

```
hdfs dfs -ls
hdfs dfs -ls /
hdfs dfs -ls -R /var
```

Display the disk space used by files

```
hdfs dfs -du -h /
hdfs dfs -du /hbase/data/hbase/namespace/
hdfs dfs -du -h /hbase/data/hbase/namespace/
hdfs dfs -du -s /hbase/data/hbase/namespace/
```

HDFS – User Commands (dfs)

Copy data to HDFS

```
hdfs dfs -mkdir tdata
hdfs dfs -ls
hdfs dfs -copyFromLocal /tmp/test.csv /data/
hdfs dfs -ls -R
```

```
cd tutorials/data/
hdfs dfs -copyToLocal /data/test.csv test.csv.hdfs
```

Copy the file back to local filesystem

HDFS – User Commands (acls)

List acl for a file

```
hdfs dfs -getfacl /data/test.csv
```

List the file statistics – (%r – replication factor)

```
hdfs dfs -stat "%r" /data/test.csv
```

Write to hdfs reading from stdin

```
echo "blah blah" | hdfs dfs -put - /data/myfile.txt
hdfs dfs -ls -R
hdfs dfs -cat /data/myfile.txt
```

HDFS – User Commands (fsck)

Removing a file

```
hdfs dfs -rm /dir3/tfile.txt
hdfs dfs -ls -R
```

List the blocks of a file and their locations

```
hdfs fsck /dir5/geneva.csv -files -blocks –
locations
```

Print missing blocks and the files they belong to

```
hdfs fsck / -list-corruptfileblocks
```

HDFS – Adminstration Commands

Comprehensive status report of HDFS cluster

```
hdfs dfsadmin -report
```

Prints a tree of racks and their nodes

```
hdfs dfsadmin -printTopology
```

```
hdfs dfsadmin -getDatanodeInfo
localhost:50020
```

HDFS – Advanced Commands

Get a list of namenodes in the Hadoop cluster

```
hdfs getconf -namenodes
```

```
The general command line syntax is
```

`hdfs command [genericOptions] [commandOptions]`

Summary: HDFS

- HDFS **provides a fault-tolerant storage layer for Hadoop and its other components**
- **Spark, Tez, … can use HDFS**
- HDFS Replication of data helps us to attain **fault-tolerant** feature.
- HDFS stores data reliably, even in the case of hardware failure.
- HDFS provides high throughput access to application data by providing the data access in parallel.
- `HDFS is write-once-read-many-times`