# STORAGE: HDFS, GFS, LUSTRE


1. Lam Truong An
2. Pham Hoang Duy Duc
3. Nguyen Hai Vinh Cuong
4. Phan Thanh Nhuan
5. Le An Pha

# OUTLINE

- Hadoop File System (HDFS)
- Google File System (GFS)
- Lustre

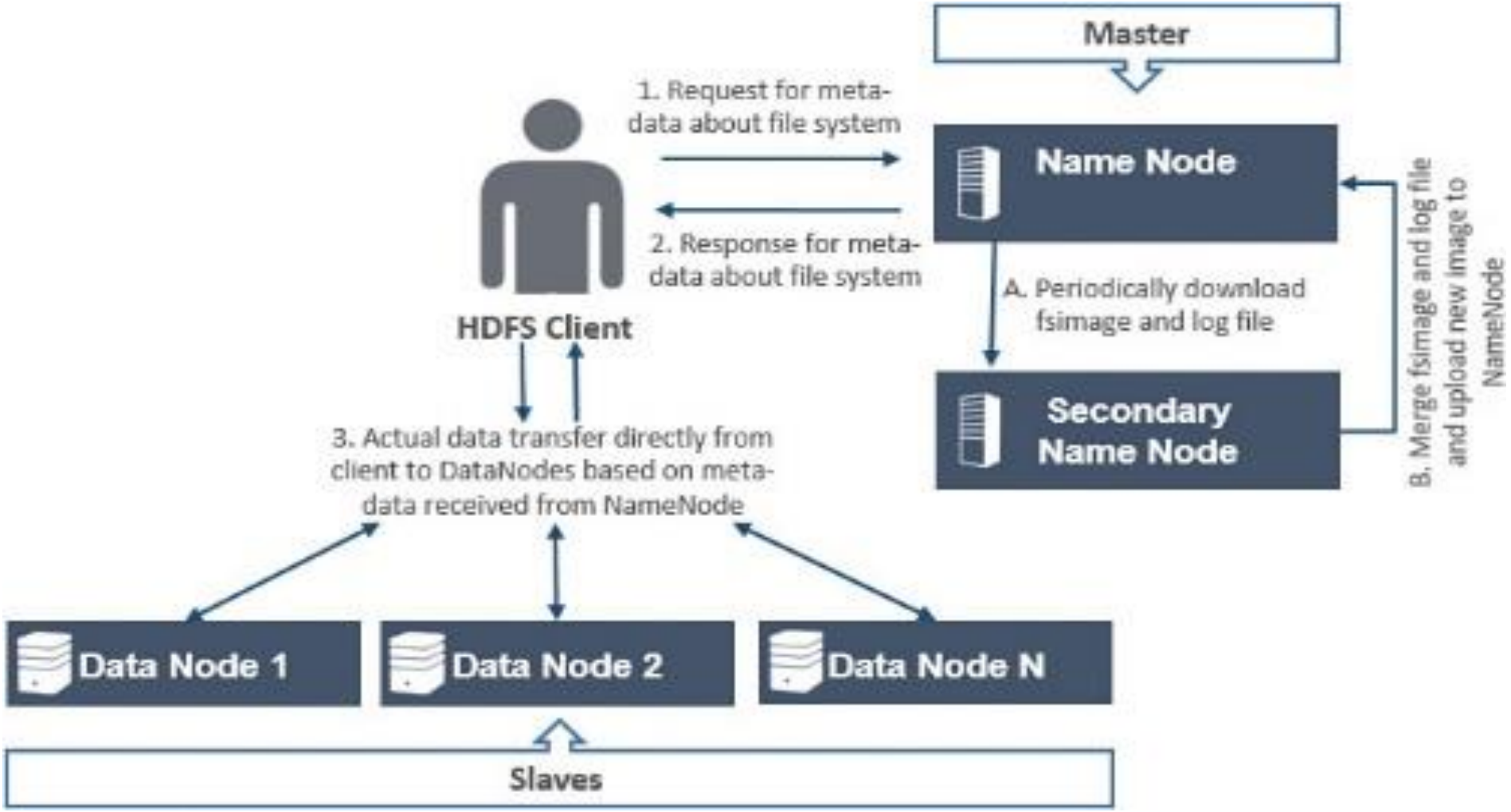
# HDFS

- INTRODUCTION
- HDFS ARCHITECTURE
- FILE I/O OPERATIONS AND REPLICA MANAGEMENT
- HANDLING FAILURES

## Introduction

- Hadoop Distributed File System.
- Scalable, fault-tolerant, distributed storage system.
- A distributed Java-based file system

# Architecture




How a client reads and writes to and from HDFS.

## Name node

- Managing file system namespace operations.
- Records any changes to the file system namespace or its properties.
- Contains information related to the replication, along with the map of the blocks.

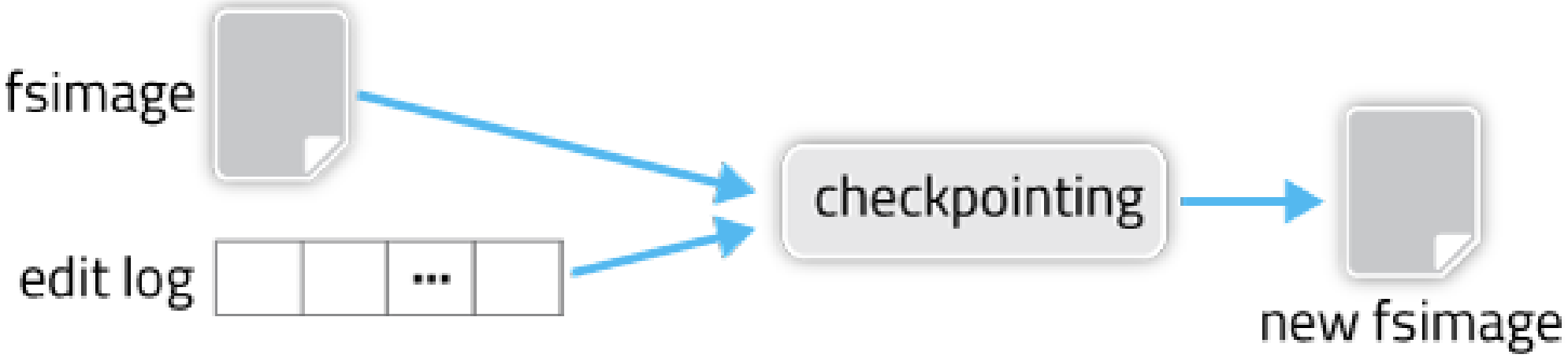
# Data nodes

- Serving read and write requests from the HDFS clients
- Perform operations such as block creation, deletion, and replication .
- Report back to the name node periodically with lists of blocks that they are storing.

# Secondary name node

- Periodically download the name node fsimage and edit the log file from the name node.
- Create a new fsimage by merging the older fsimage and edit the log file, and upload the new fsimage back to the name node

# Secondary name node

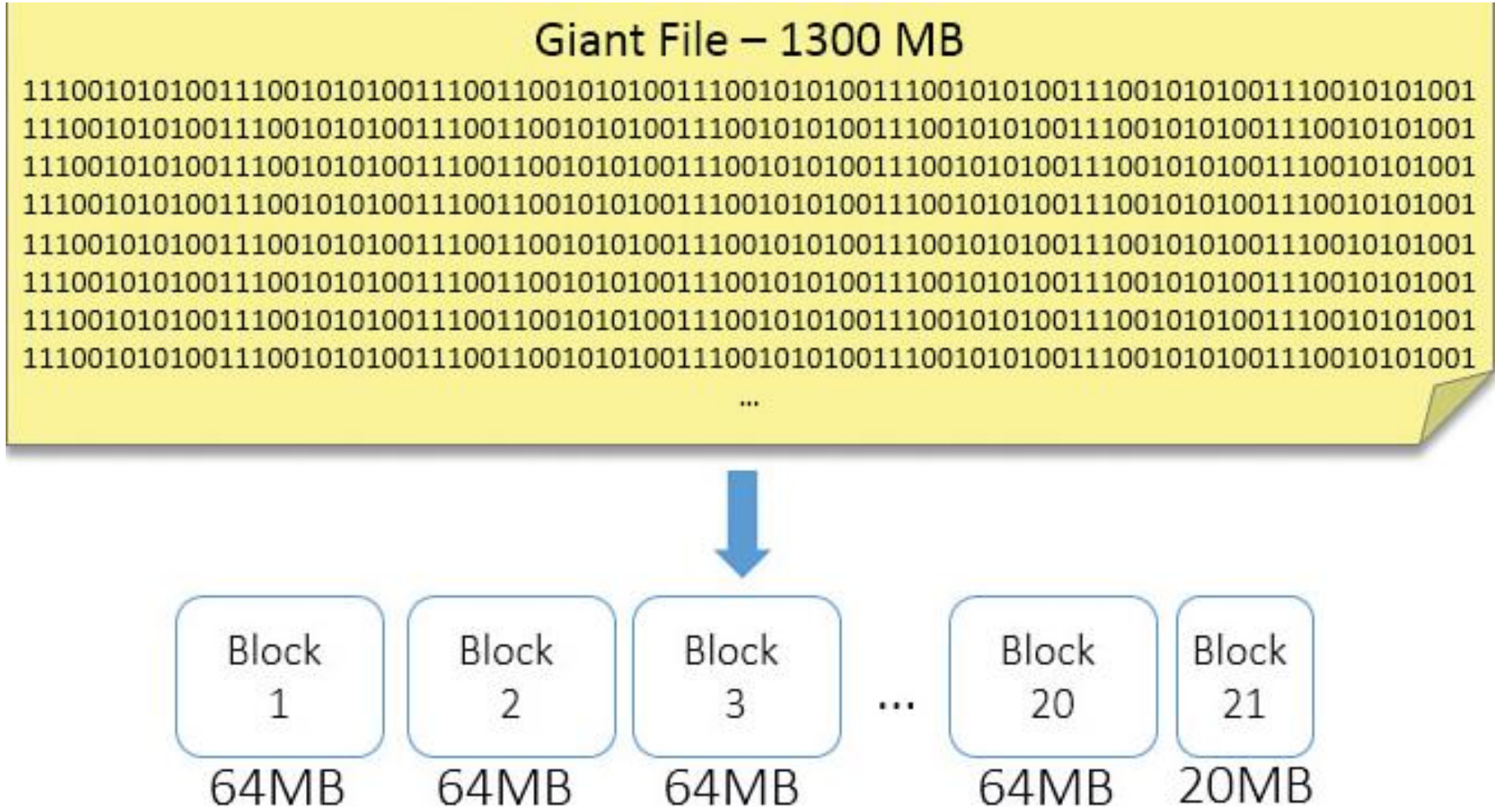



Update fsimage with Editt log.

# FILE I/O OPERATIONS AND REPLICA MANAGEMENT

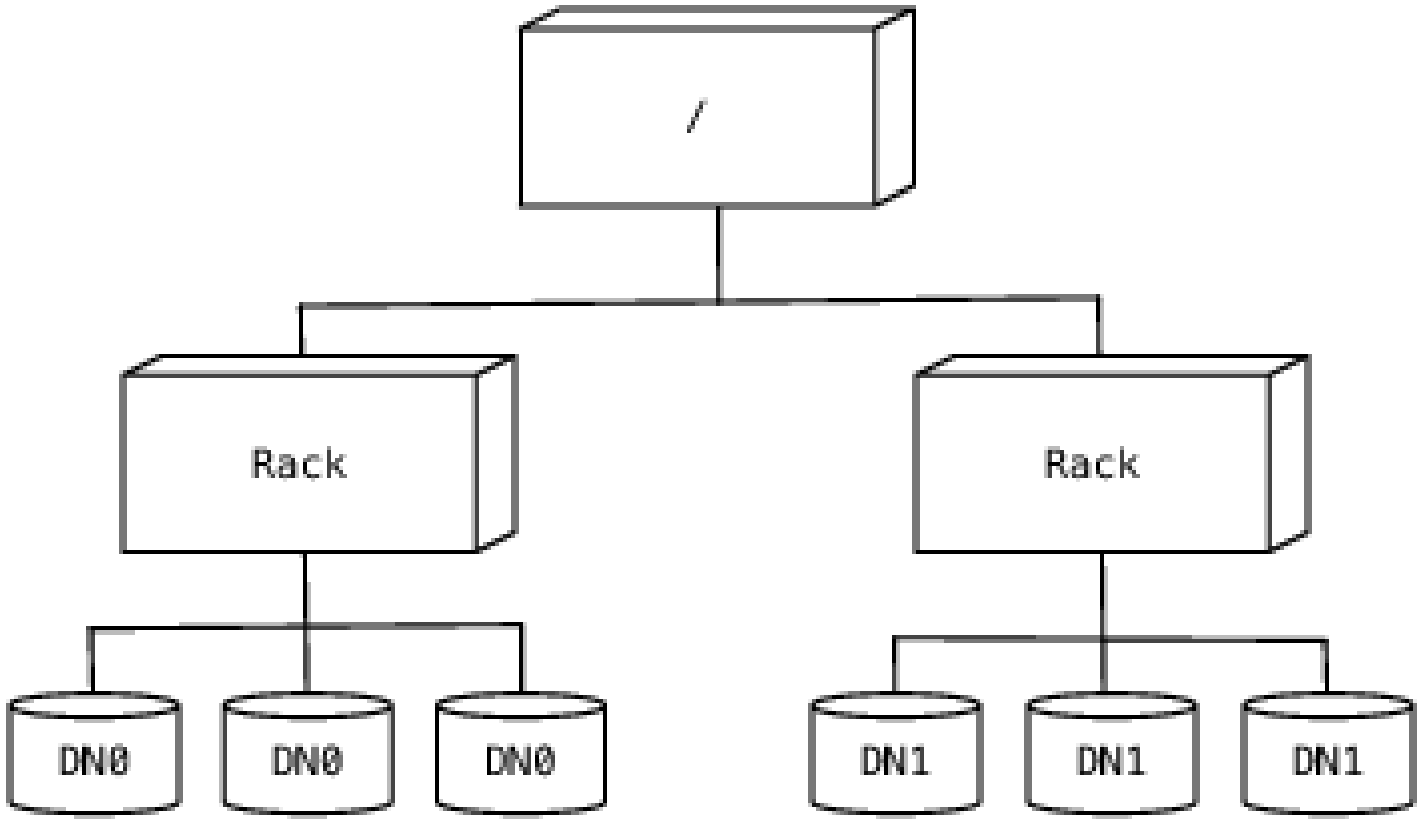
- File Split
- Block Placement and Replication in HDFS
- Writing and Reading

# File Split in HDFS



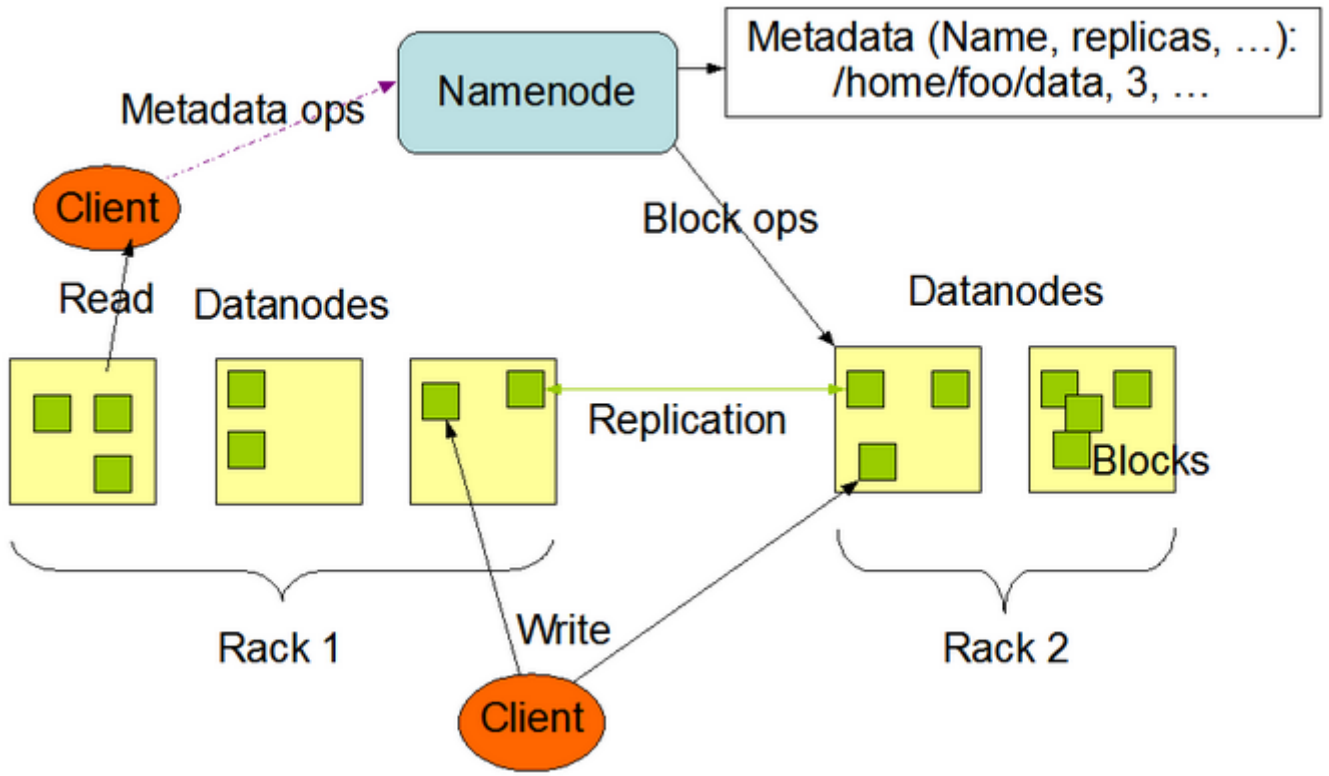

File split process when writing to HDFS.

# Block Placement and Replication in HDFS




Block Placement

# Block Placement and Replication in HDFS

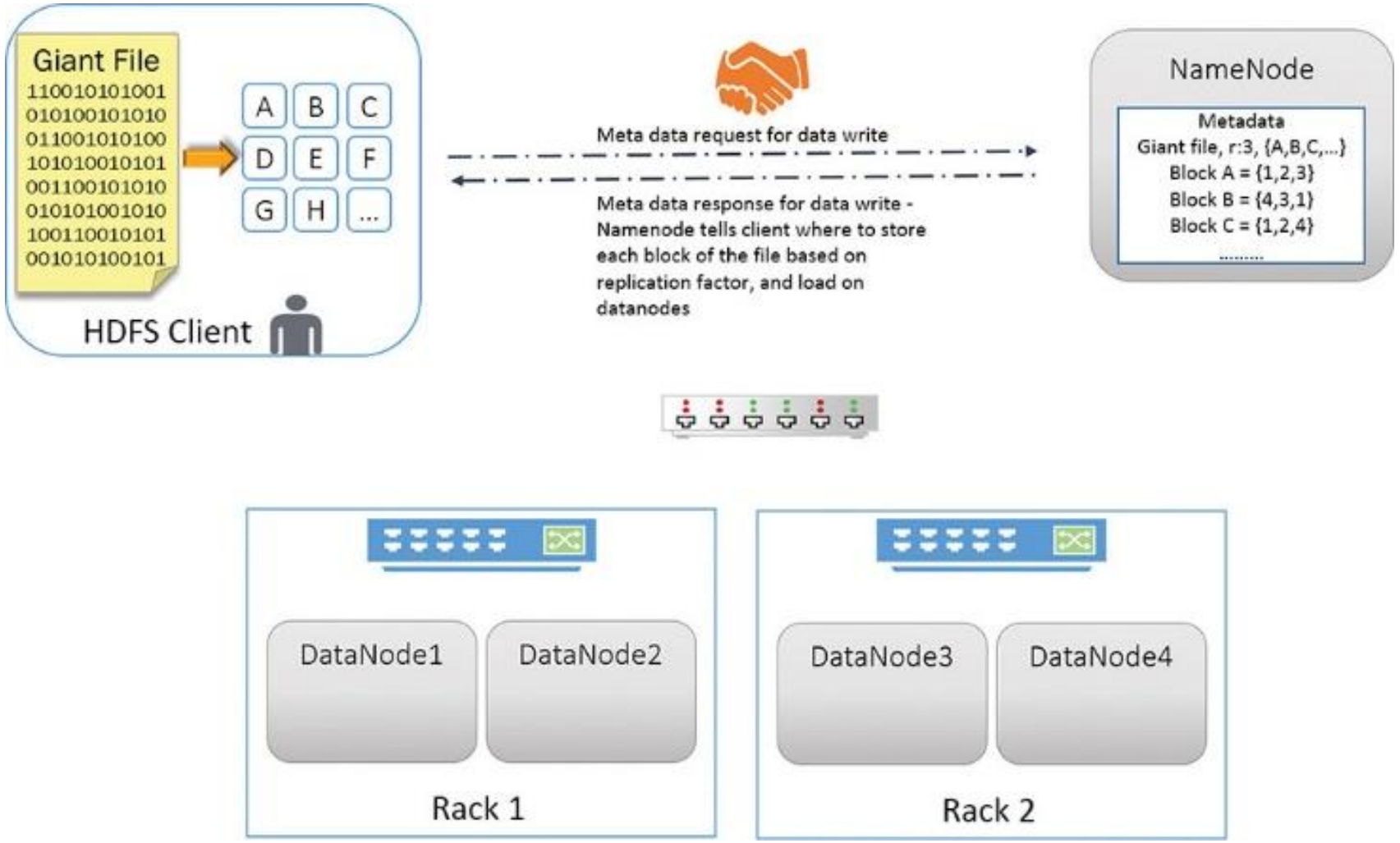



Block Placement

# Block Placement and Replication in HDFS

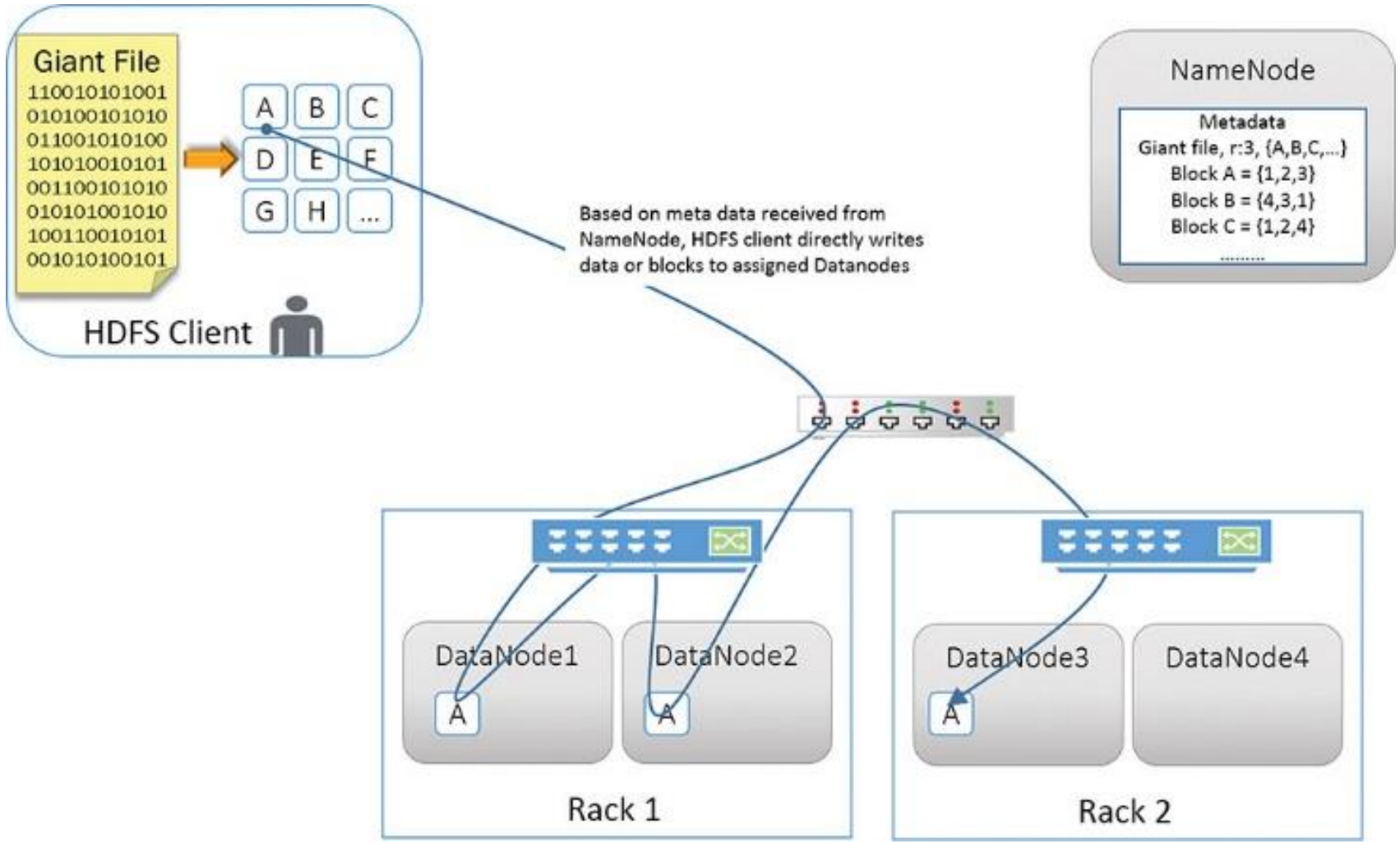
- HDFS has a balanced default block placement policy. Its objective is to have a properly load-balanced, fast-access, fault-tolerance file system:
- The first replica is written to the data node creating the file.
- The second replica is written to another data node within the same rack.
- The third replica is written to a data node in a different rack.

# Writing to HDFS



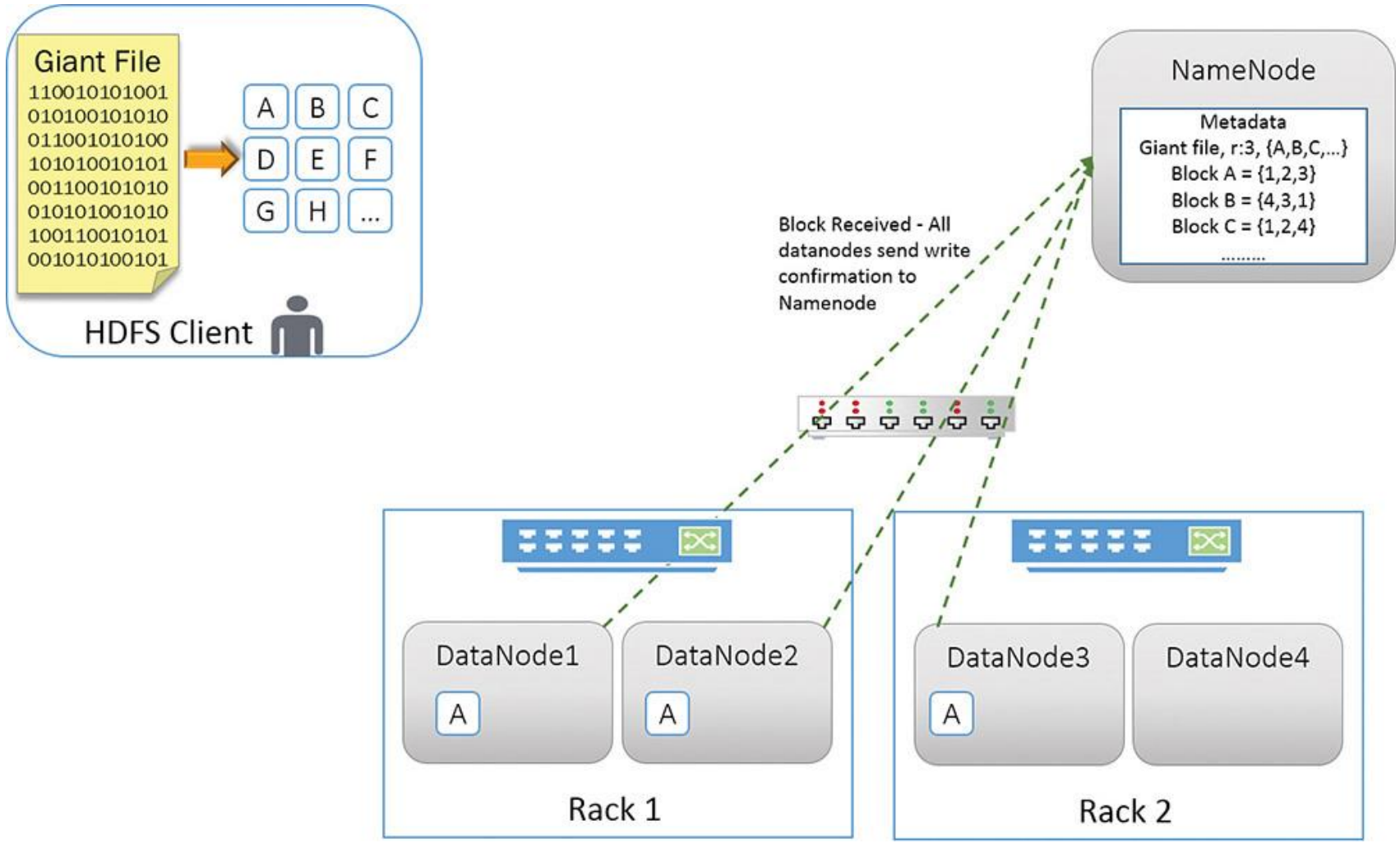

The client talks to the name node for metadata to specify where to place the data blocks.

# Writing to HDFS



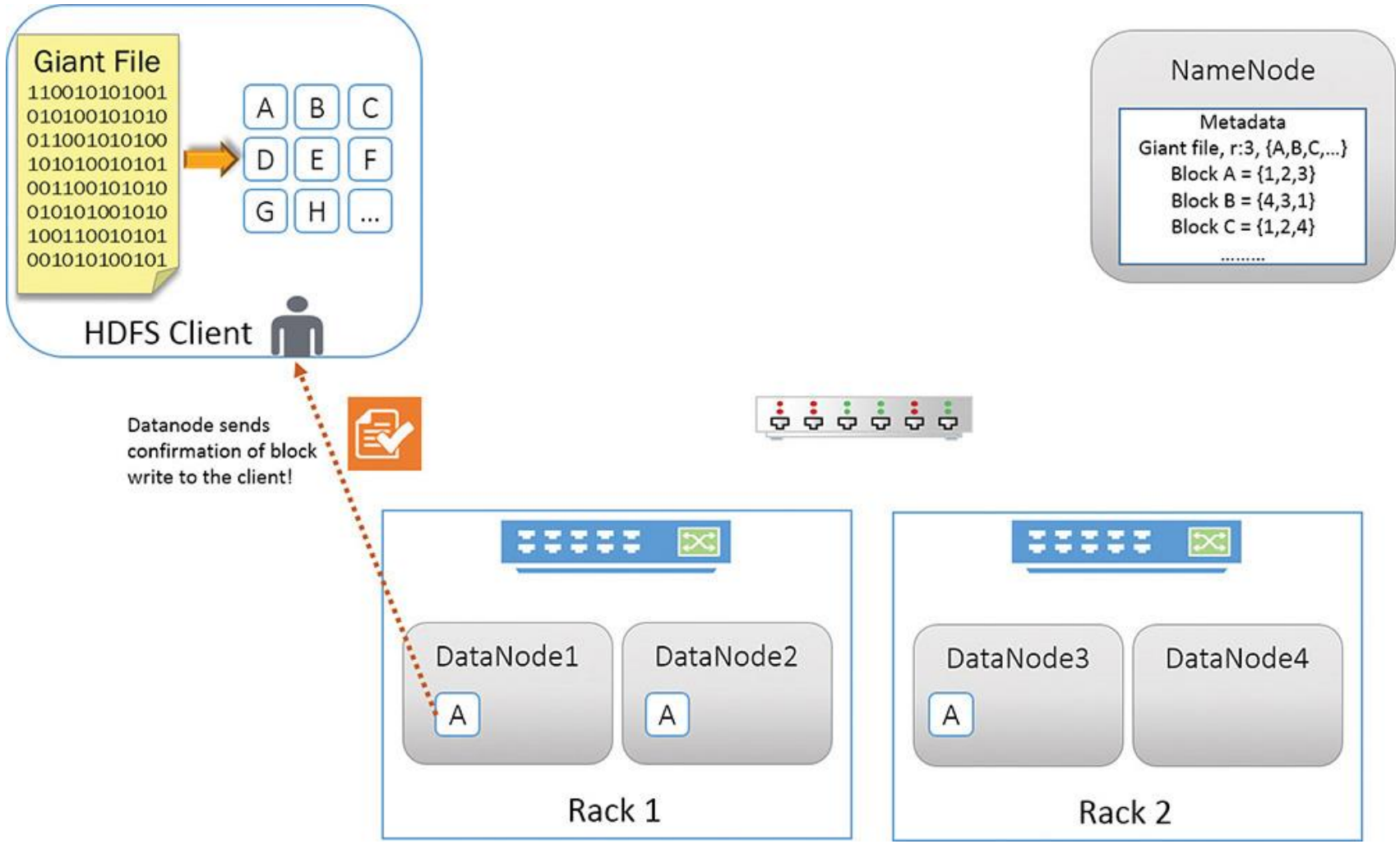

The client sends data blocks to identified data nodes.

# Writing to HDFS




Data nodes update the name node about receipt of the data blocks.

# Writing to HDFS

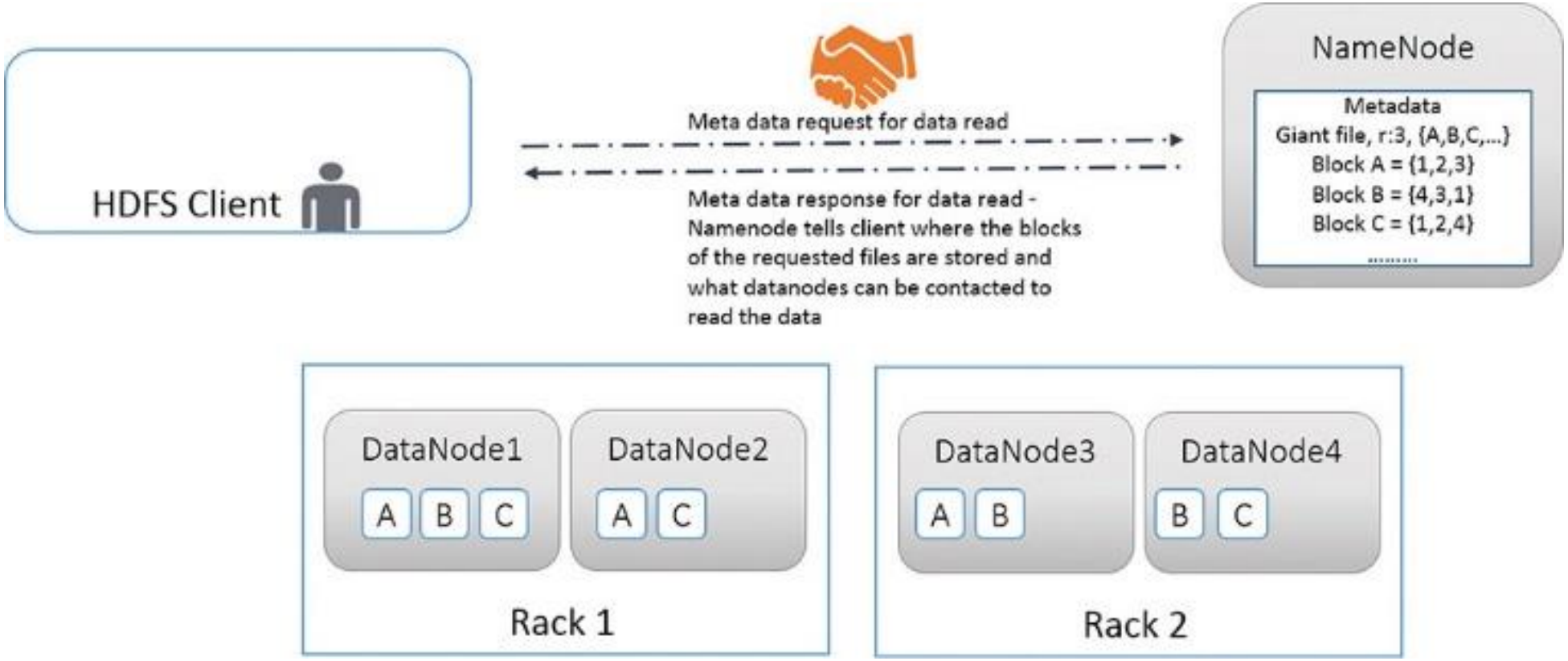



The first data node sends an acknowledgment back to the client.

# Writing to HDFS

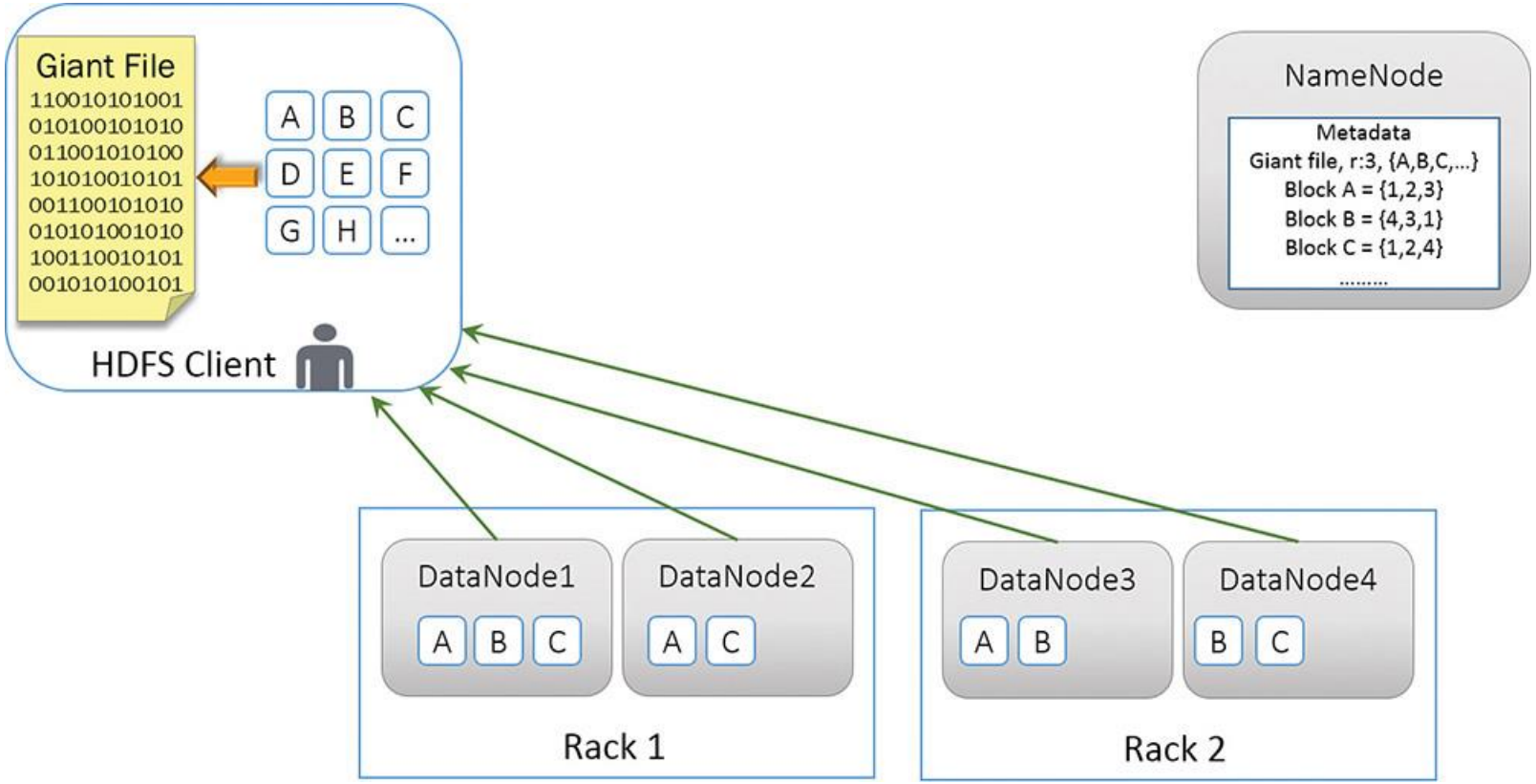
- For simplicity, we demonstrated how one block from the client is written to different data nodes. But the whole process is actually repeated for each block of the file, and data transfer happens in parallel for faster write of blocks.
- All communication from clients to the name node, clients to data nodes, data nodes to the name node, and name node to the data nodes happens over Transmission Control Protocol/Internet Protocol (TCP/IP).

# Reading from HDFS



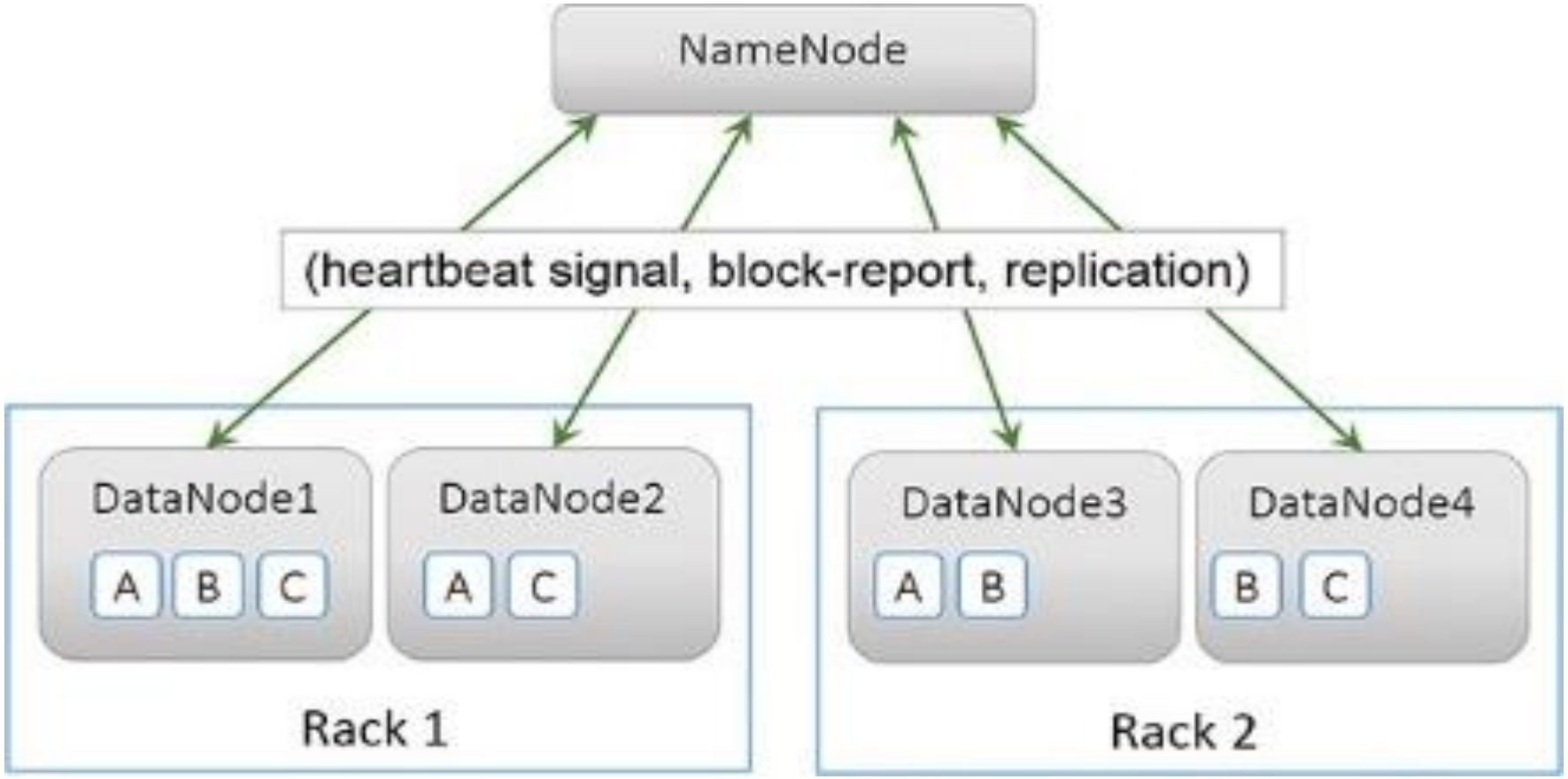

The client talks to the name node to get metadata about the file it wants to read.

# Reading from HDFS



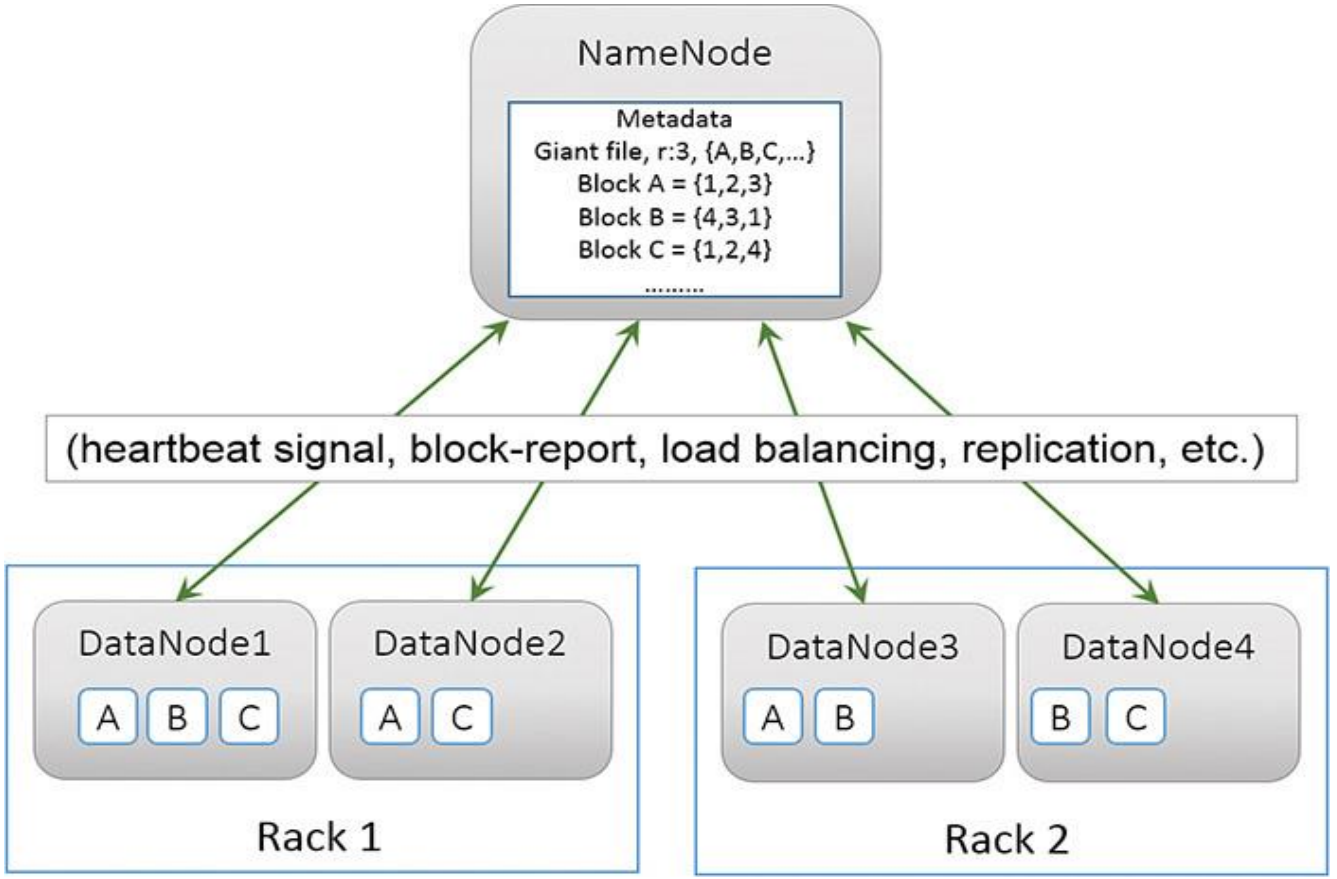

The client starts reading data blocks of the file from the identified data nodes.

# Handling Failures



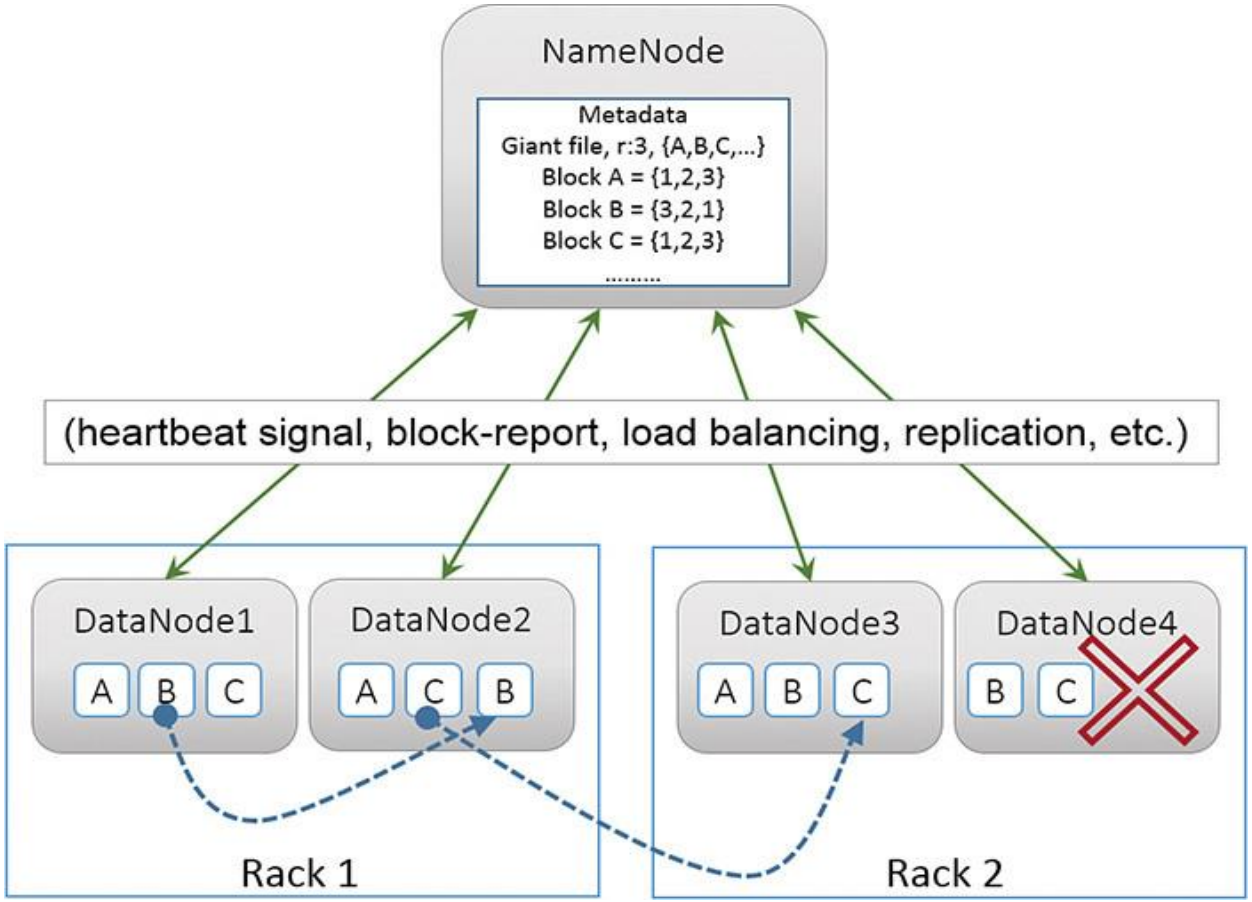

All data nodes periodically send heartbeat signals to the name node.

# Handling Failures




The name node updates its metadata based on information it receives from the data nodes.

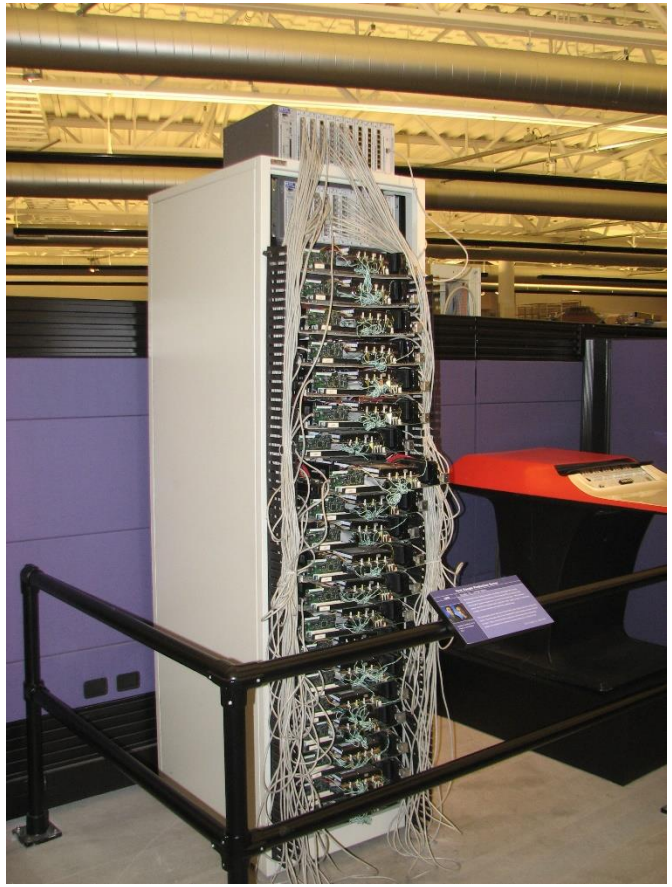
# Handling Failures




Handling a data node failure transparently.

# GOOGLE FILE SYSTEM

- Motivation
- Architecture
- System Interactions
- Master Operations
- Fault Tolerance



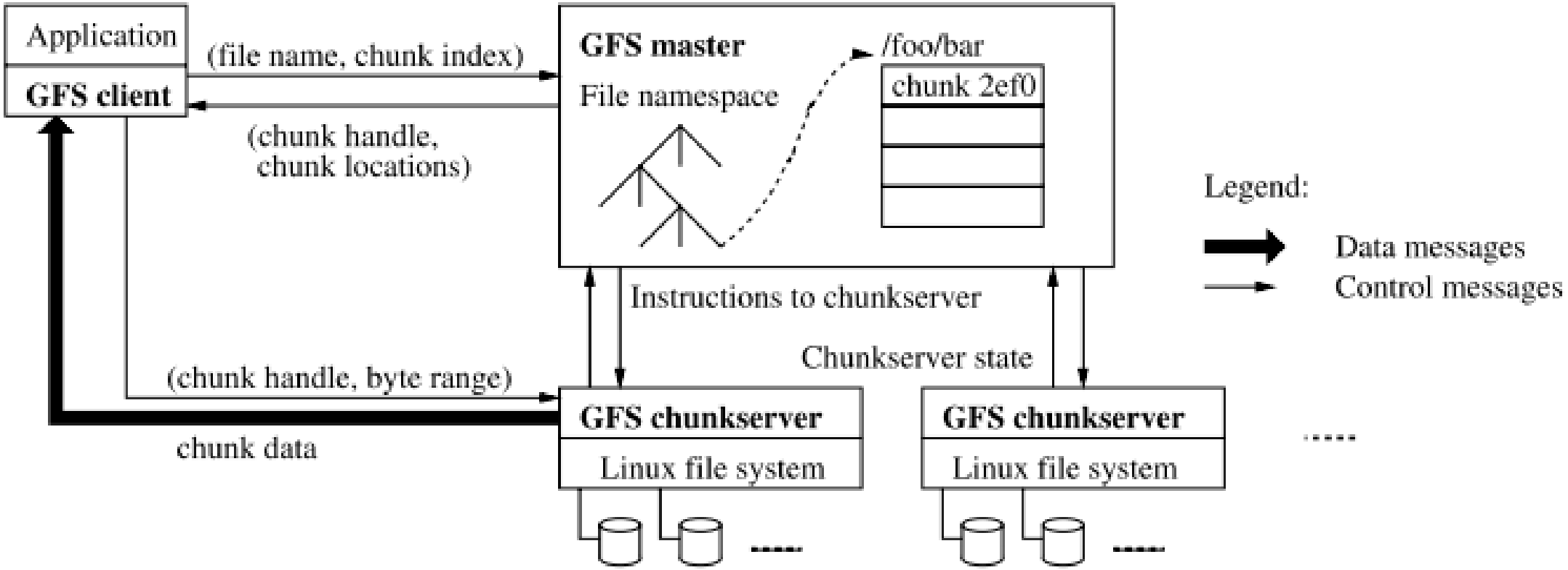
# GFS - Motivation

- Google’s unique workload and environment.
  - Component failures are the norm rather than the exception.
  - Files are huge by traditional standards.
  - Most files are mutated by appending new data rather than overwriting existing data.
  - Co-designing the applications and the file system API benefits the overall system by increasing our flexibility.

# GFS - Assumptions

- The system is built from many inexpensive commodity components that often fail.
- The  system stores a modest number of large files.
- The workloads primarily consist of two kinds of reads: large streaming reads and small random reads.
- The workloads also have many large, sequential writes that append data to files.
- The system must efficiently implement well-defined semantics for multiple clients that concurrently append to the same file.
- High sustained bandwidth is more important than low latency.

# GFS - Architecture



# GFS - Architecture

- Chunk Size: 64 MB, a key design parameter.
- Disadvantages
  - Wasted space
  - Hot spots if many clients are accessing on small files consist of a few chunks.
- Advantages
  - Reduces clients' need to interact with master
  - Reduces network overhead
  - Reduces the size of the metadata

# GFS - Architecture

- Metadata is maintained by the master
  - File and chunk namespaces
  - Mapping from files to chunks
  - Locations of each chunk's replicas
- All in memory (64 bytes / chunk)
  - Fast
  - Easily accessible
- Operation log for persistent logging of critical metadata updates
  - Persistent on local disk
  - Replicated on shadow master
  - Checkpoints for faster recovery

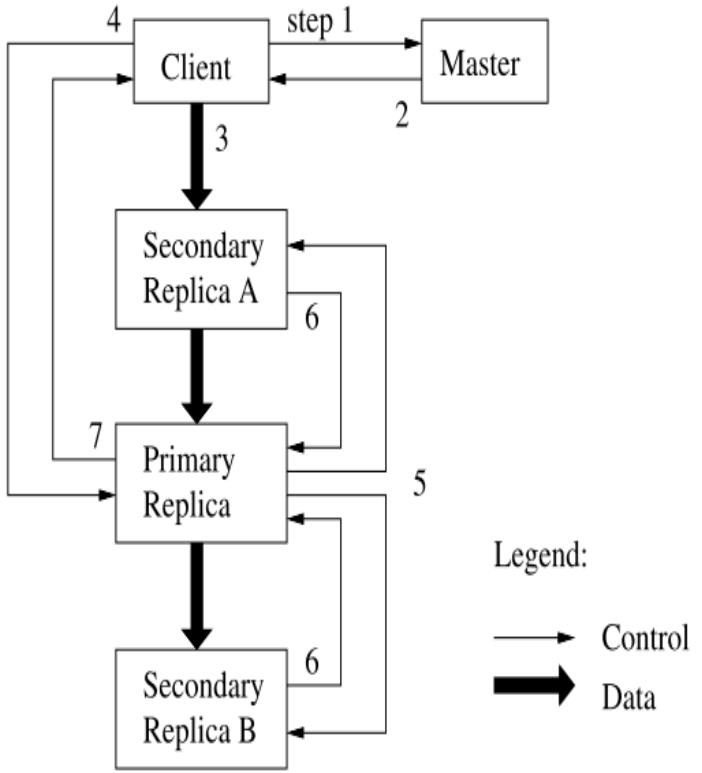
# GFS – Relaxed Consistency Model

- File namespace mutations are atomic.
- Terminology:
  - *consistent*: all clients will always see the same data, regardless of which replicas they read from
  - *defined*: same as consistent and, furthermore, clients will see what the modification is in its entirety

| | Write | Record Append |
|---|---|---|
| Serial success | *defined* | *defined* interspersed with *inconsistent* |
| Concurrent successes | *consistent* but *undefined* | |
| Failure | *inconsistent* | |

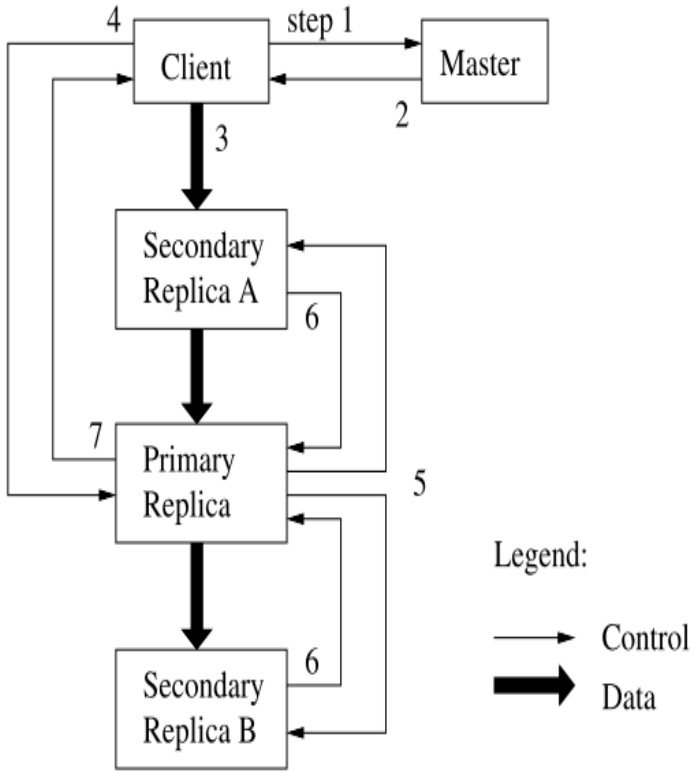
# GFS - System Interactions

- **Read Operation**
- **Write Operation**
  1. Client asks replica locations and identity of primary replica (leases holder)
  2. Master replies with the identity of the primary and the locations of the other (secondary) replicas
  3. Client pushes data to replicas
  4. Client issues update request to primary
  5. Primary forwards/performs write request
  6. Primary receives replies from replica
  7. Primary replies to client



# GFS - System Interactions

- **Record Append Operation**
  - Performed atomically
  - Append location chosen by GFS and returned to client
  - Extension to step 5:
    - If record fits in current chunk: write record and tell replicas the offset
    - If record exceeds chunk: pad the chunk, notifies secondaries to do the same, reply to client to use next chunk
- **Snapshot**
  - Makes a copy of a file or a directory tree almost instantaneously
  - minimize interruptions of ongoing mutations
  - copy-on-write with reference counts on chunks



# GFS - Master Operations

- **Namespace Management and Locking**
  - Logically a mapping from pathname to chunk list
  - Allows concurrent file creation in same directory
  - Read/write locks prevent conflicting operations
  - File deletion by renaming to a hidden name; removed during regular scan
- **Replica Placement**
  - Serves two purposes:
    - maximize data reliability and availability
    - maximize net-work bandwidth utilization
  - Spread chunk replicas across racks.

# GFS - Master Operations

- **Creation, Re-replication, Rebalancing**
  - Creation: place the initially empty replicas
  - Re-replication: the number of available replicas falls below goal.
  - Rebalancing: periodically for better disk space and balancing.
- **Garbage Collection**
  - Does not reclaim the available physical storage at time of file deletion.
  - Master identifies orphaned chunks in a similar regular scan of the chunk namespace
  - “HeartBeat” message: chunkserver is free to delete.
- **Stale Replica Detection**
  - Version number to each chunk/replica.
  - Stale replicas removed as part of garbage collection

# GFS - Fault Tolerance

- **High Availability**
  - Fast Recovery
  - Chunk Replication
  - Master Replication
- **Data Integrity**
  - A responsibility of chunkservers
  - 32 bit checksum for every 64 KB block of data
  - Verify checksum before returning data

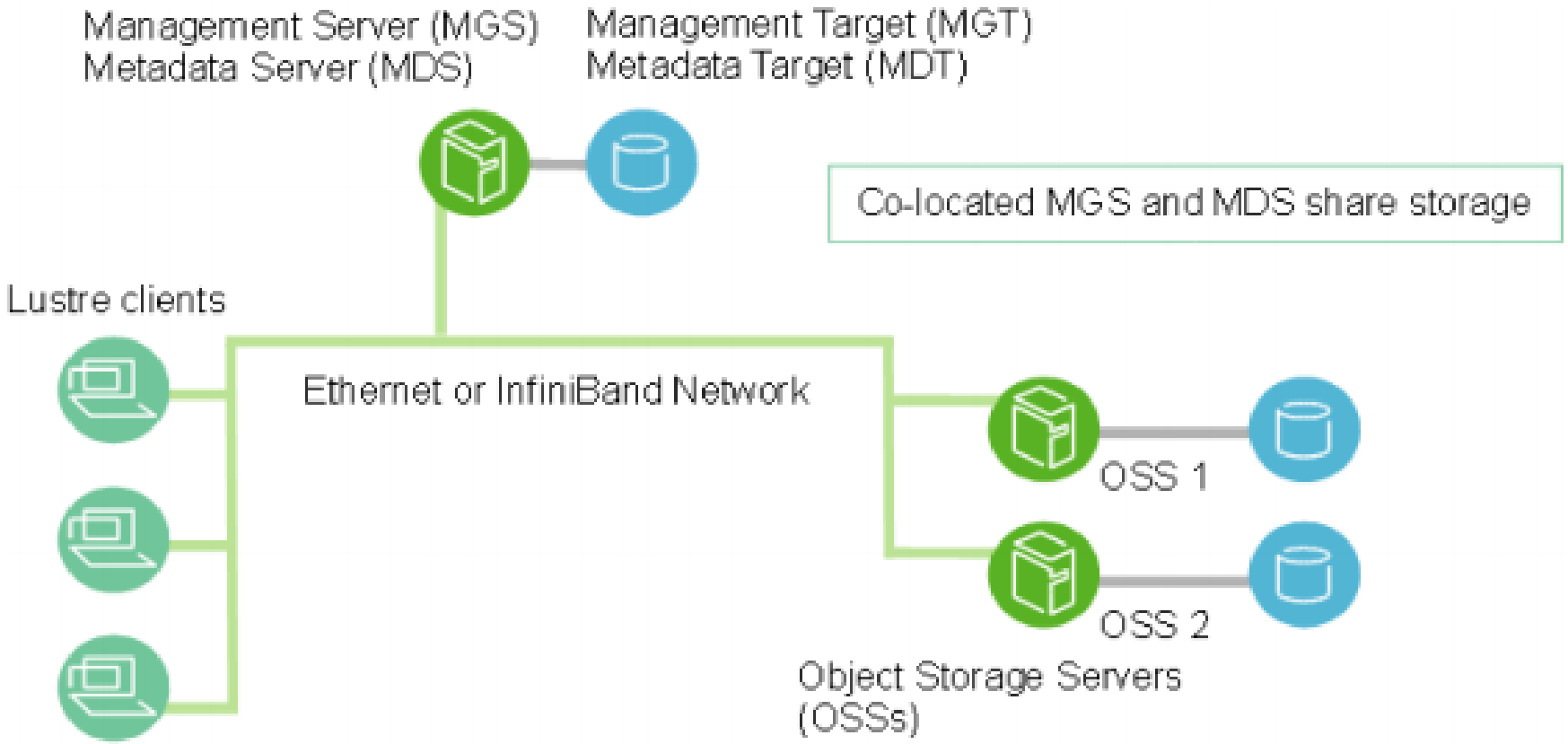
# LUSTRE

- Introduction
- Basic Components
- Read Write Process
- File Striping
- Failover

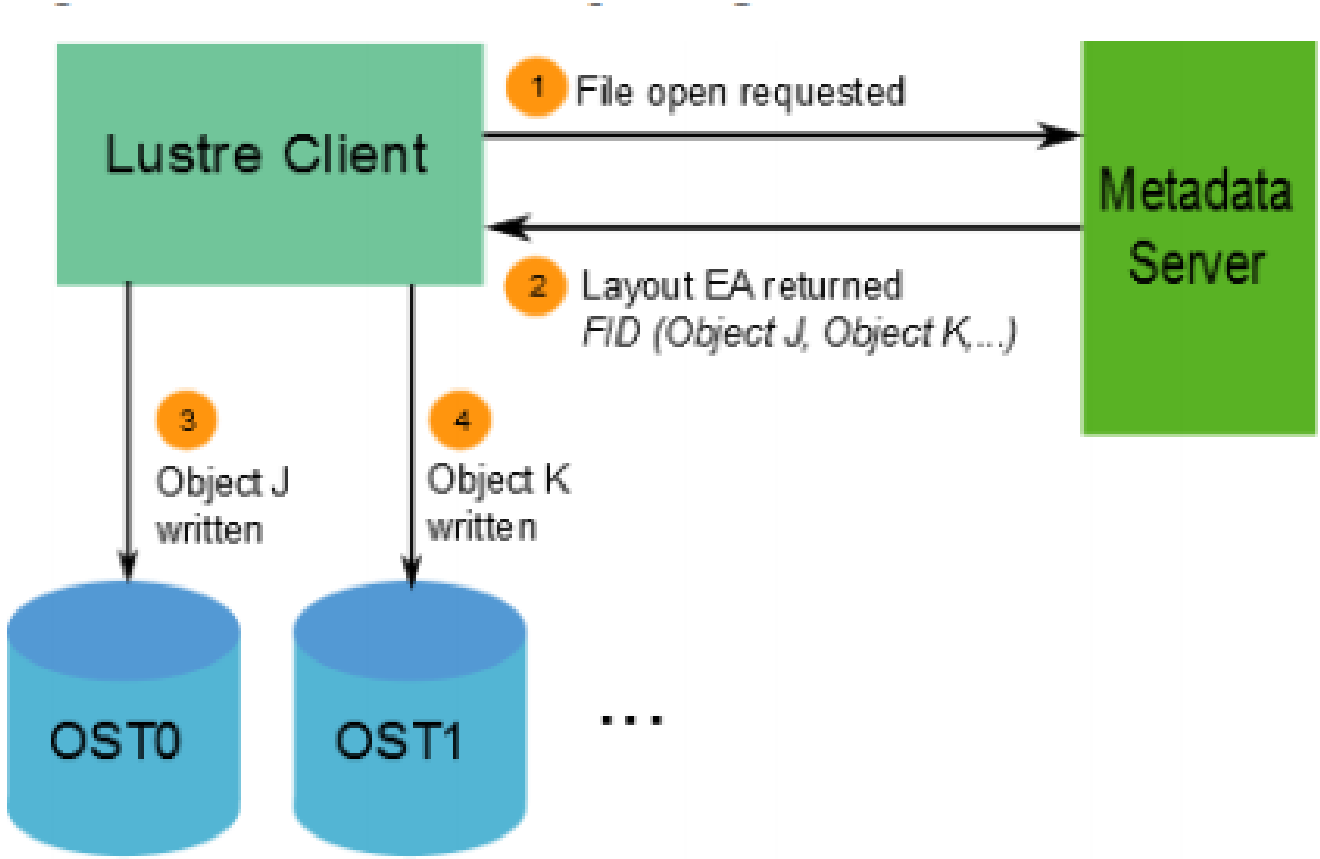
# Introduction

- **Lustre** is a type of parallel distributed file system, generally used for large-scale cluster computing.
- Lustre file systems are scalable and can be part of multiple computer clusters with tens of thousands of client nodes, tens of petabytes (PB) of storage on hundreds of servers, and more than a terabyte per second (TB/s) of aggregate I/O throughput

# Basic Components

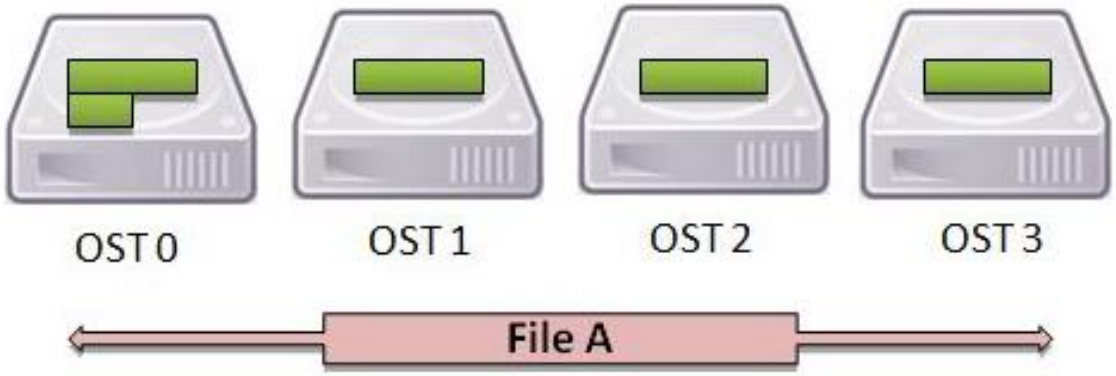


# Read Write Process



# File Striping

- A key feature of the **Lustre** file system is its ability to distribute the segments of a single file across multiple OSTs using a technique called **file striping**



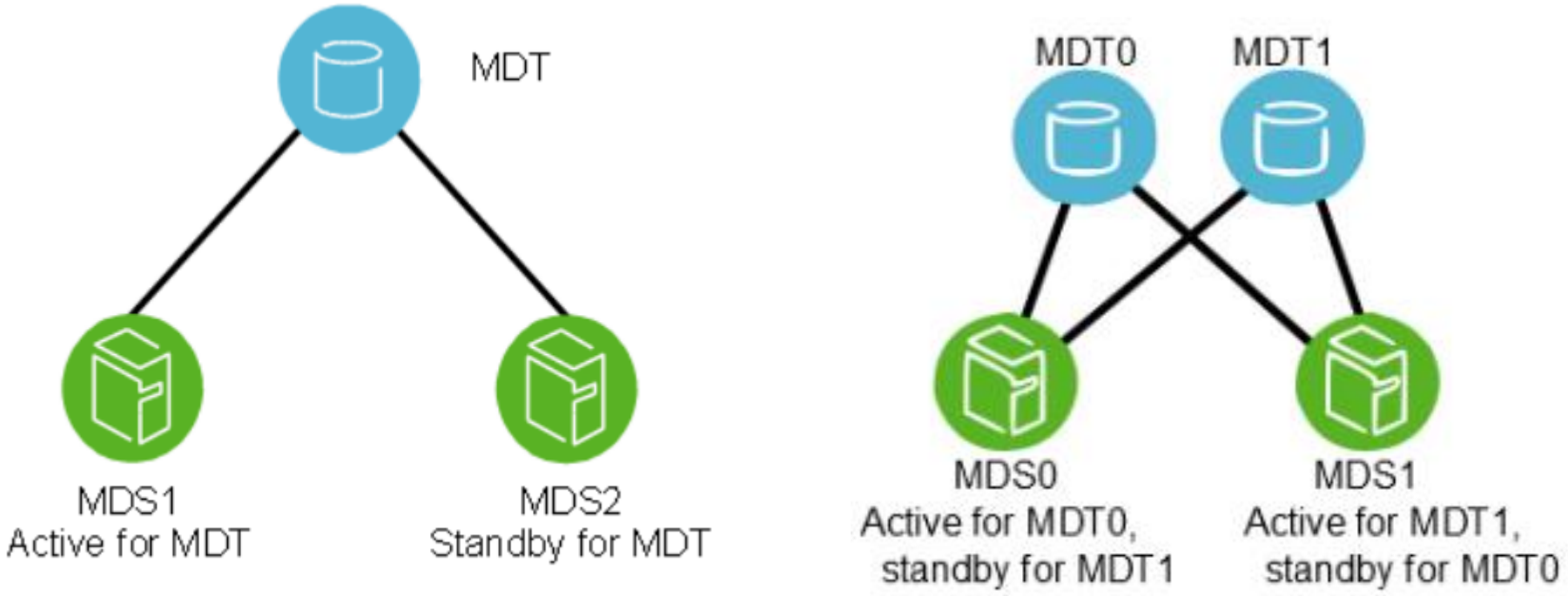
# Failover

- When a primary server fails or is unavailable, a standby server can be switched into its place to run applications and associated resources.

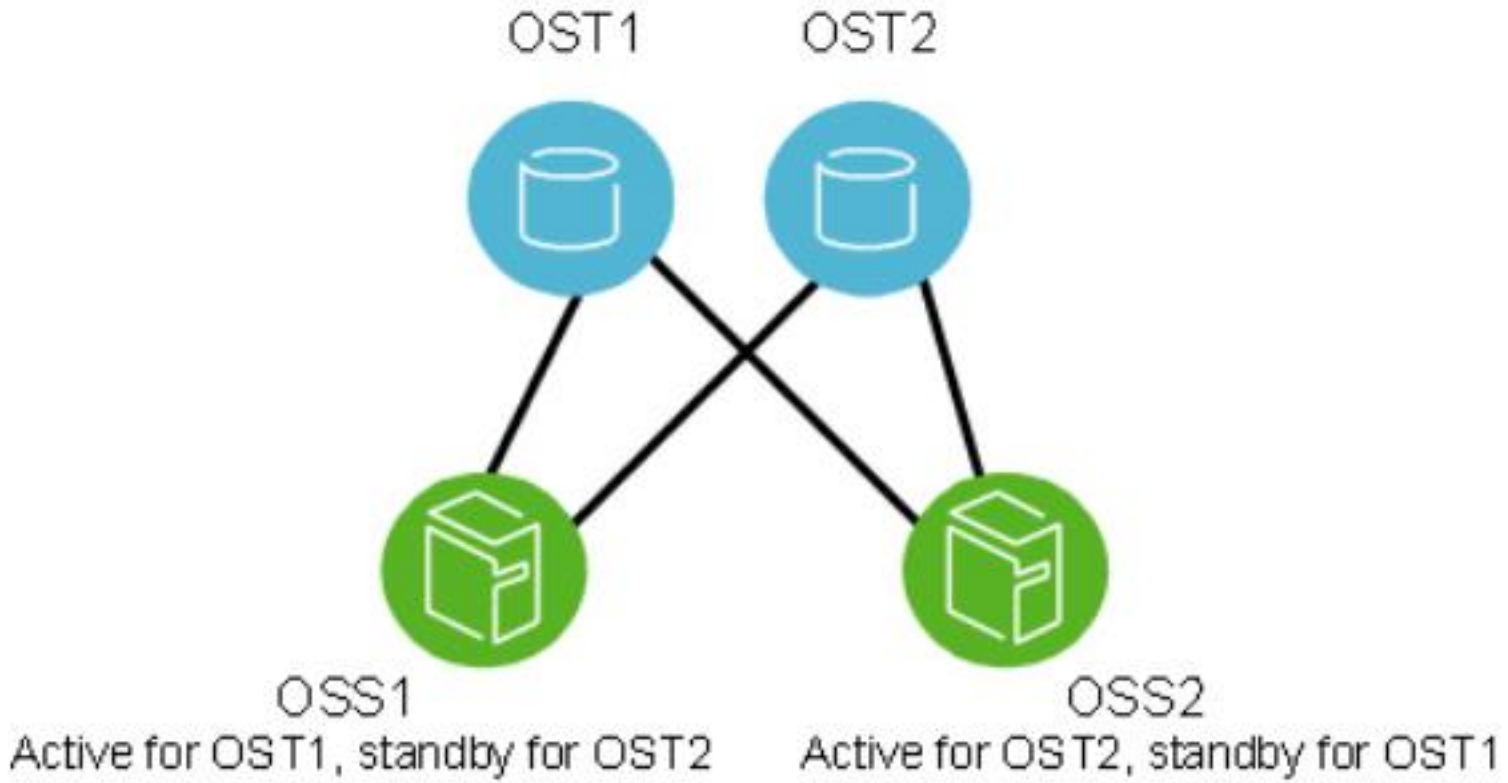
- The **Lustre** software does not provide redundancy for data. It depends exclusively on redundancy of backing storage devices.

# MDT Failover

- Two ways to configu.



# OST Failover



# Reference


- Benjamin Depardon, Ga¨el Le Mahec, Cyril S´eguin, “**Analysis of Six Distributed File Systems**”, HAL, Feb 2013.
- Manpreet Singh, Arshad Ali, “**Big Data Analytics with Microsoft HDInsight in 24 Hours”,** Nov 8, 2015
- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, “**The Google File System**”, 2003
- http://lustre.org/
- https://www.nics.tennessee.edu/computing-resources/file-systems/io-lustre-tips