Serialization in Hadoop

- Serialization is the process of turning objects into a byte stream
- Deserialization is the reverse process of turning a byte stream back into a series of objects.
- Serialization is used in two quite distinct areas of distributed data processing
  - for interprocess communication
  - for persistent storage
- In Hadoop, interprocess communication between nodes in the system is implemented using remote procedure calls (RPCs). RPC protocol uses serialization/deserialization
- Hadoop uses its own serialization format called **Writables**
  - compact and fast
  - not so easy to extend or use from languages other than Java
  - There are other serialization frameworks supported in Hadoop, such as Avro, Thrift, Protobuffers, but Writables are by far the most used in Hadoop

Writable Inferfaces

```
package org.apache.hadoop.io;

import java.io.DataOutput;
import java.io.DataInput;
import java.io.IOException;

public interface Writable
{
   void write(DataOutput out) throws IOException;

   void readFields(DataInput in) throws IOException;
}
```

```
package org.apache.hadoop.io;

public interface WritableComparable<T> extends Writable, Comparable<T>
{
}
```

IntWritable Example (I)

```
package org.apache.hadoop.io;

import java.io.*;

public class IntWritable implements WritableComparable {

  private int value;

  public IntWritable() {}
  public IntWritable(int value) { set(value); }

  public void set(int value) {
    this.value = value;
  }

  public int get() {
    return value;
  }

  public void readFields(DataInput in) throws IOException {
    value = in.readInt();
  }

  public void write(DataOutput out) throws IOException {
    out.writeInt(value);
  }
```

IntWritable Example (II)

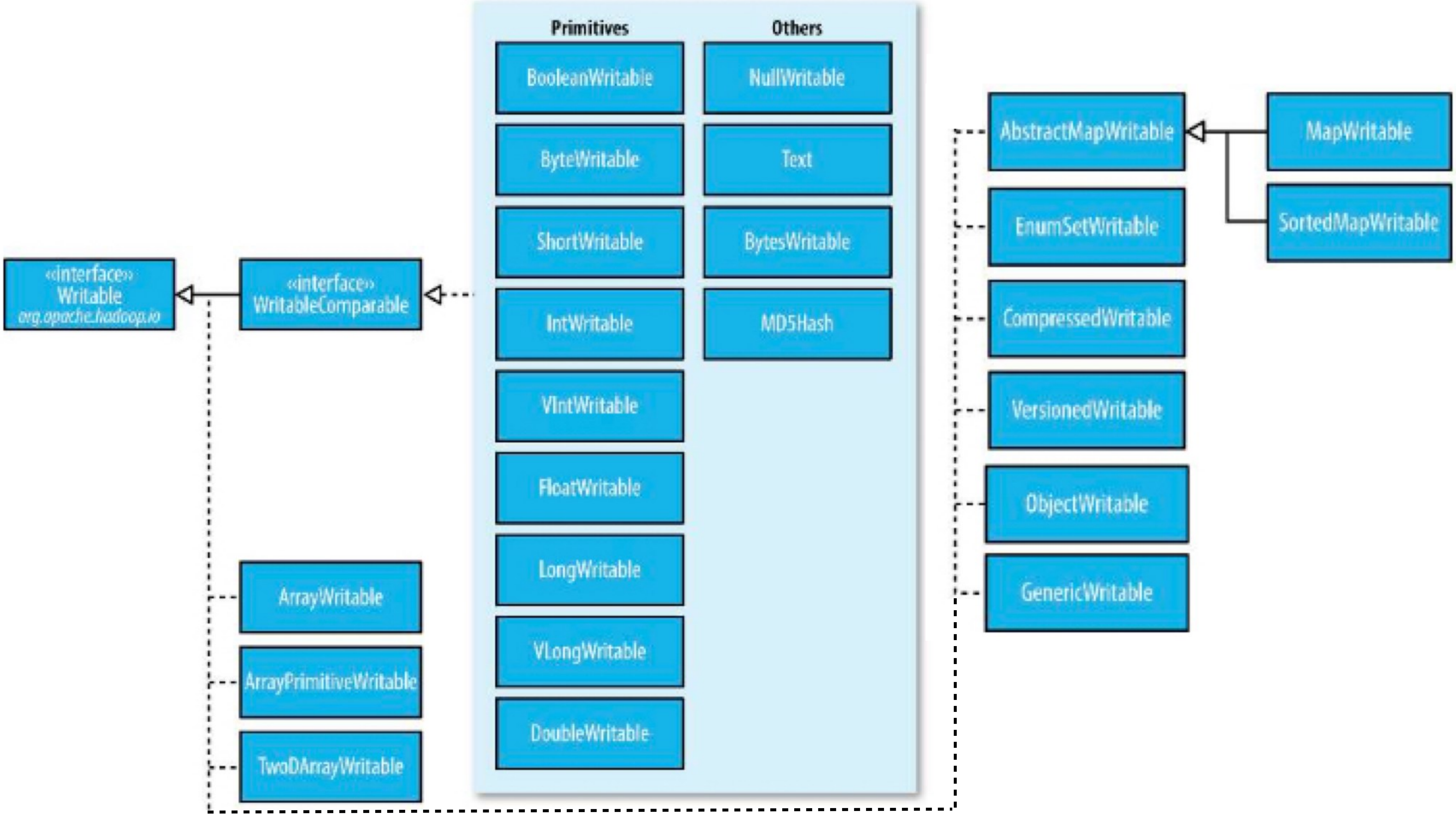
```
public boolean equals(Object o) {
  if (!(o instanceof IntWritable))
    return false;

  IntWritable other = (IntWritable)o;
  return this.value == other.value;
}

public int hashCode() {
  return value;
}

public int compareTo(Object o) {
  int thisValue = this.value;
  int thatValue = ((IntWritable)o).value;
  return (thisValue<thatValue ? -1 : (thisValue==thatValue ? 0 : 1));
}

public String toString() {
  return Integer.toString(value);
}
```

Writable Wrappers

Java primitive	Writable implementation	Serialized size (bytes)
boolean	BooleanWritable	1
byte	ByteWritable	1
short	ShortWritable	2
int	IntWritable	4
	VIntWritable	1–5
float	FloatWritable	4
long	LongWritable	8
	VLongWritable	1–9
double	DoubleWritable	8

Source: Tom White, "*Hadoop - The Definitive Guide (4 ed)*", O'Reilly, 2015

Writable Class Hierarchy



Source: Tom White, "*Hadoop - The Definitive Guide (4 ed)*", O'Reilly, 2015

Hadoop Input (I)

- An input split is a portion of the input that is processed by a single map task
- Each split is divided into records, and the map task processes each record – a key-value pair – in turn
- Splits and records are <u>logical</u>
  - Not required to be files, although commonly they are.
  - In a database context, a split might correspond to a range of rows from a table, and a record to a row in that range
  - Input splits are represented by the class InputSplit
- An InputSplit has a length in bytes and a set of storage locations (i.e., hostname strings)
  - A split doesn't contain the input data
  - A split is just a reference to the data
  - The storage locations are used by Hadoop to place map tasks as close to the split's data as possible
  - The size is used to order the splits so that the largest get processed first, to minimize the job runtime

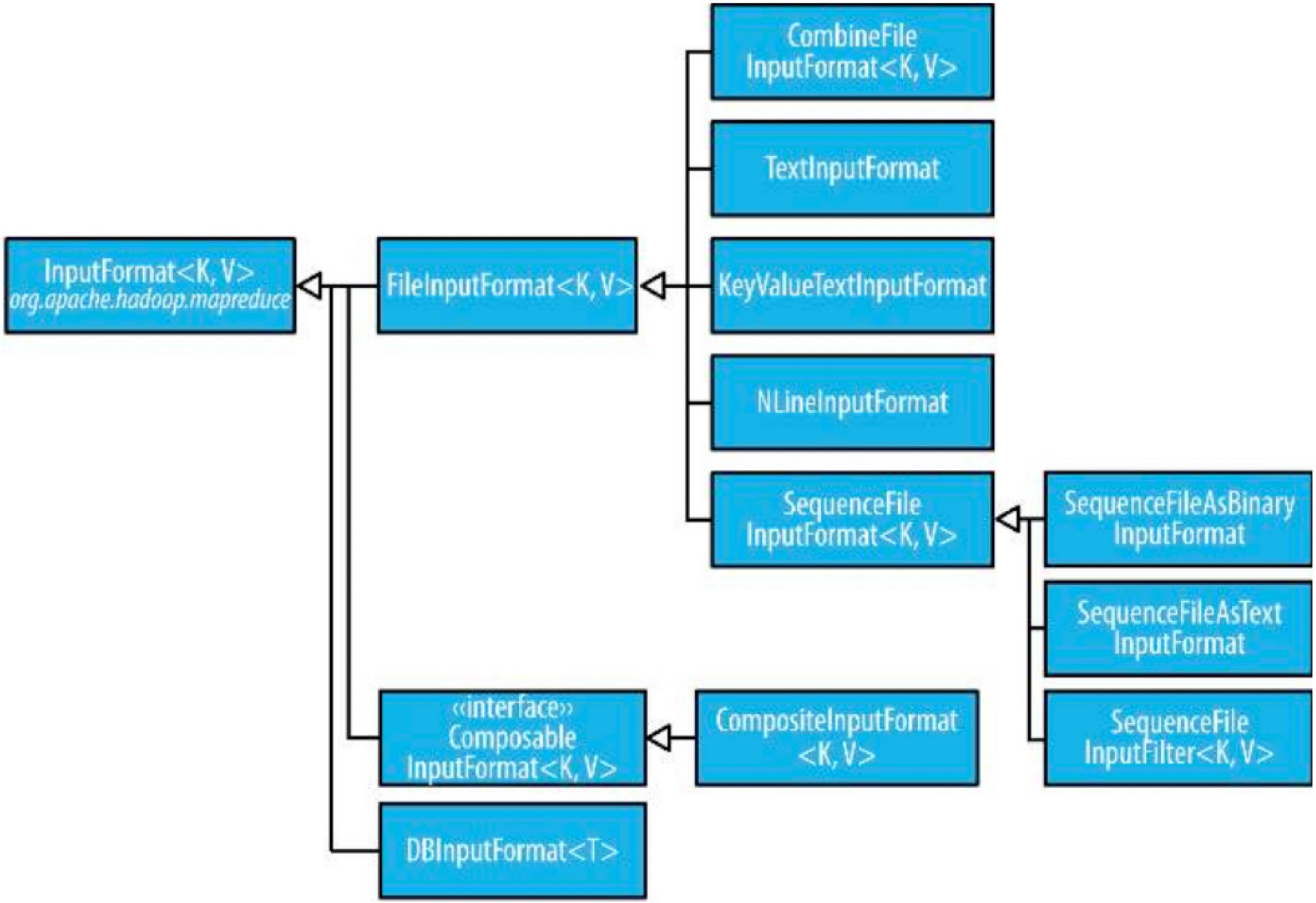
Hadoop Input (II)

- As a MapReduce application writer, you do not need to deal with InputSplits directly, as these are created by an InputFormat interface implementation

```
public abstract class InputFormat<K, V>

{

  public abstract List<InputSplit> getSplits(JobContext context)

    throws IOException, InterruptedException;

  public abstract RecordReader<K, V> createRecordReader(InputSplit split, TaskAttemptContext context)

    throws IOException, InterruptedException;

}
```

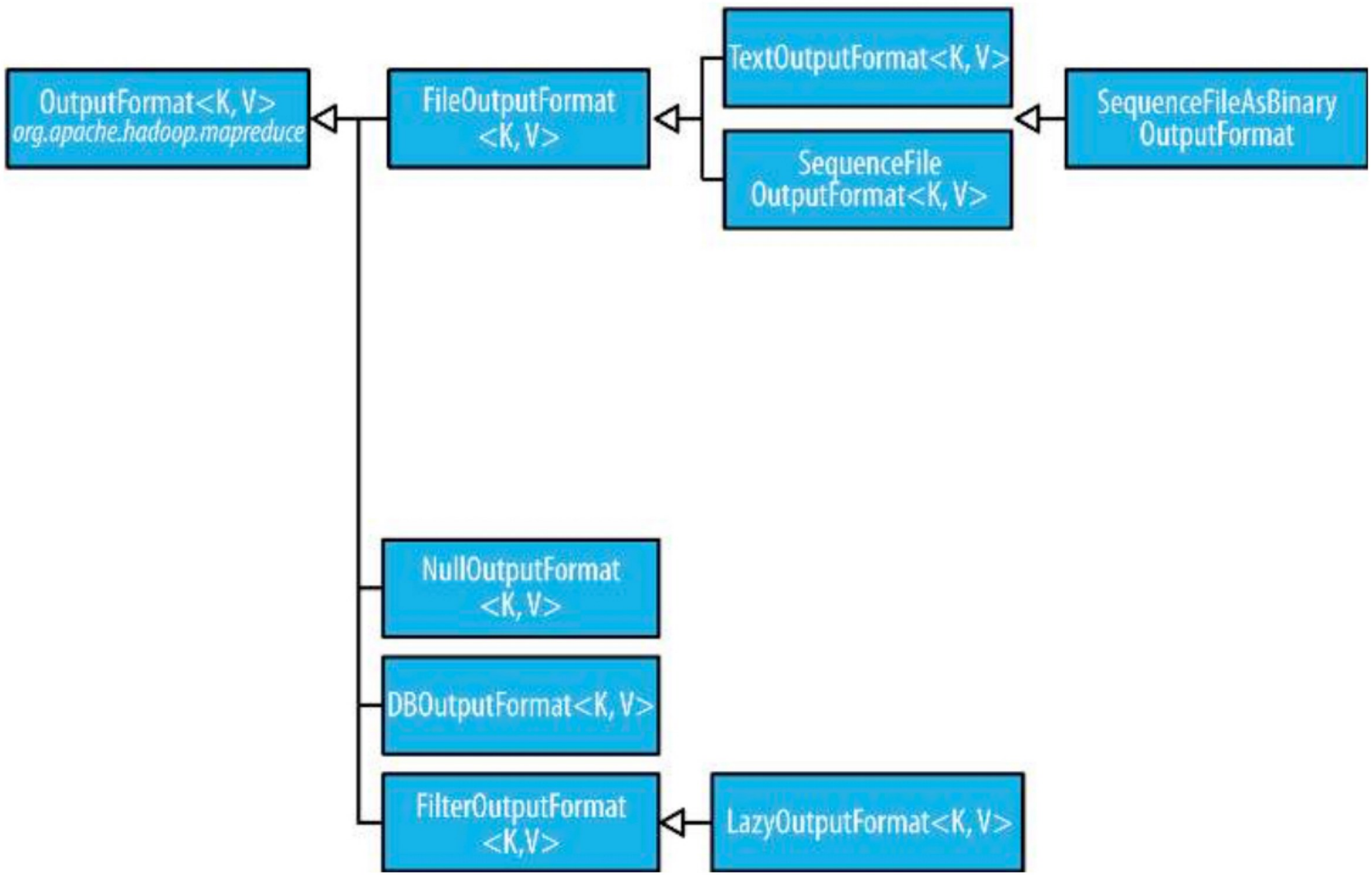
- Splits for the job are created through the getSplits() method
- Splits are then sent to the application master, which uses their storage locations to schedule map tasks that will process them on the cluster
- The map task passes the split to the createRecordReader() method on InputFormat to obtain a RecordReader for that split.
- A RecordReader is little more than an iterator over records, and the map task uses one to generate record key-value pairs, which it passes to the map method.

InputFormat Class Hierarchy



Source: Tom White, "*Hadoop - The Definitive Guide (4 ed)*", O'Reilly, 2015

OutputFormat Class Hierarchy



Source: Tom White, "*Hadoop - The Definitive Guide (4 ed)*", O'Reilly, 2015

The setup and cleanup methods

- It is common to want your Mapper or Reducer to execute some code before the map() or reduce() method is called for the first time
  - Initialize data structures
  - Read data from an external file
  - Set parameters
- The setup() method is run before the map() or reduce() method is called for the first time

```
public void setup(Context context)
```

- Similarly, you may wish to perform some action(s) after all the records have been processed by your Mapper or Reducer
- The cleanup() method is called before the Mapper or Reducer terminates

```
public void cleanup(Context context)
```

Passing parameters

```
public class MyDriverClass
{
   public int main(String[] args) throws Exception
   {
      int value = 42;
      Configuration conf = new Configuration();
      conf.setInt ("paramname", value);
      Job job = new Job(conf);
      // ...
      return job.waitForCompletion(true);
   }
}
```

```
public class MyMapper extends Mapper
{
   public void setup(Context context)
   {
      Configuration conf = context.getConfiguration();
      int myParam = conf.getInt("paramname", 0);
      // ...
   }

   public void map...
}
```

Hadoop Distributed File System

Requirements/Features for a DFS

- It is a distributed file system
  - Manages storage across a network of machines in a cluster
- Designed to run on clusters of commodity hardware
  - Does not require expensive, highly-reliable hardware
  - Commonly available, low-cost hardware
- Highly fault-tolerant
  - Failures are the norm rather than the exception

Organization of a DFS

- Files are divided into **chunks** (or blocks)
  - typically 64/128 megabytes in size
- Blocks are replicated at different compute nodes (usually 3+)
- Nodes holding copies of one block are located on <u>different racks</u>
- Block size and the degree of replication can be decided by the user
- A special node (the master node) stores, for each file, the positions of its blocks
- The master node is itself replicated

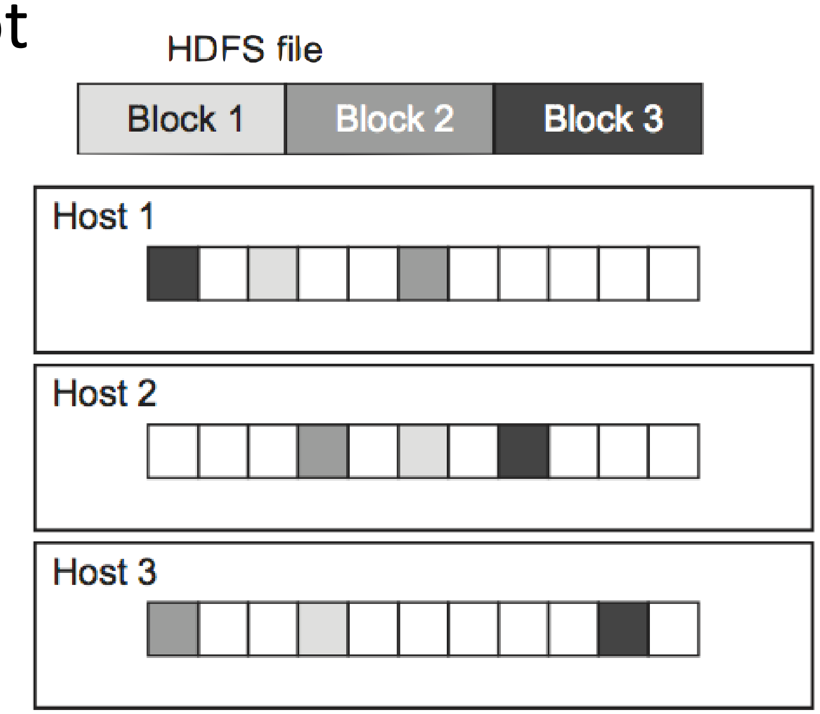
Blocks

Single-disk filesystems

- Minimum amount of data that it can read or write
- File System Blocks are typically few KB
- Disk blocks are normally 512 bytes

DFS

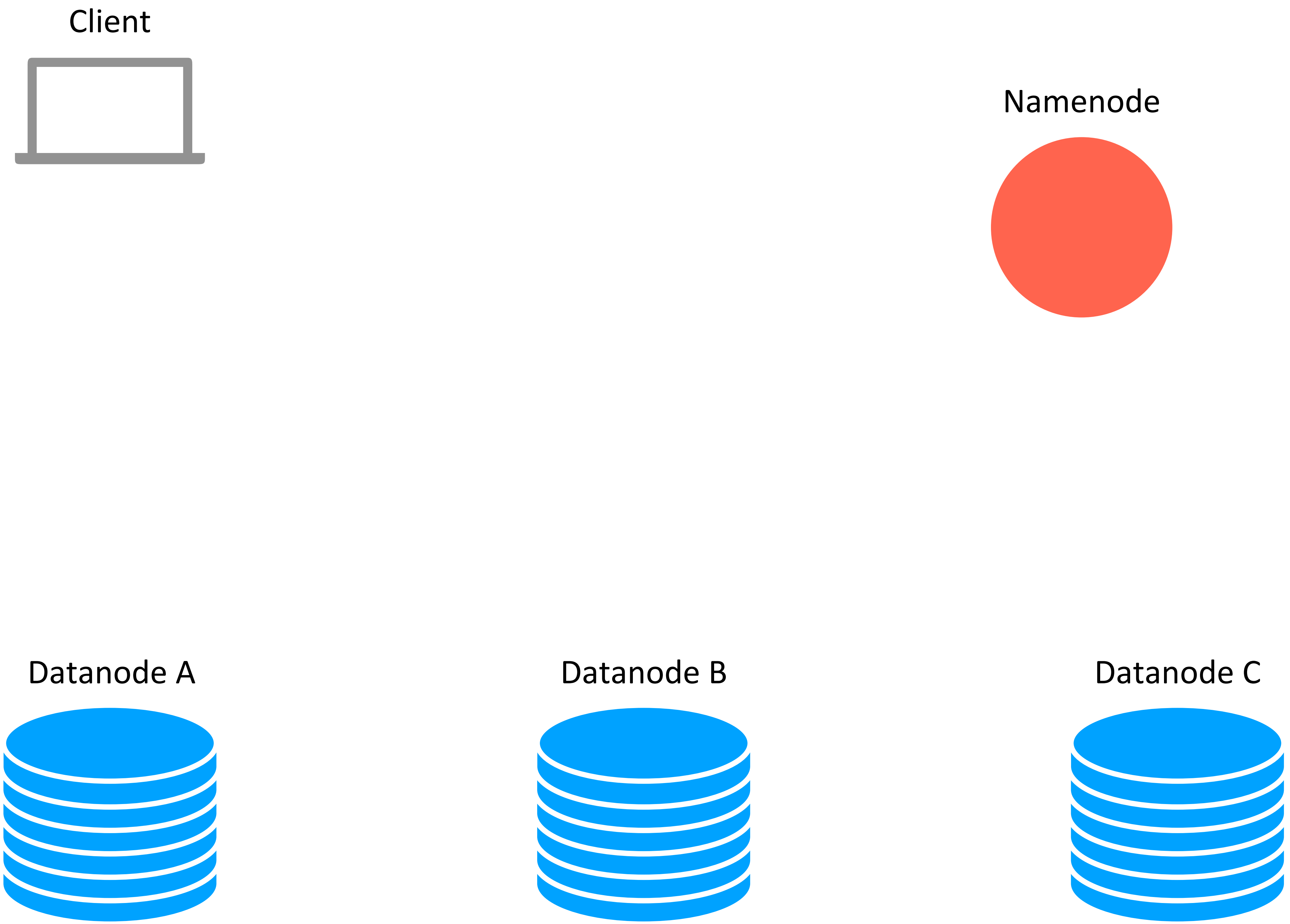
- DFS Block is much larger – e.g., 64/128 MB by default in HDFS
  - Unlike single-disk file system, the smaller file does not occupy the full 64MB block size
  - Large to minimize the cost of seeks
- Block abstractions allows
  - A file can be larger than any single disk
  - Using blocks as units of abstraction simplifies the storage subsystem (e.g., fixed size)



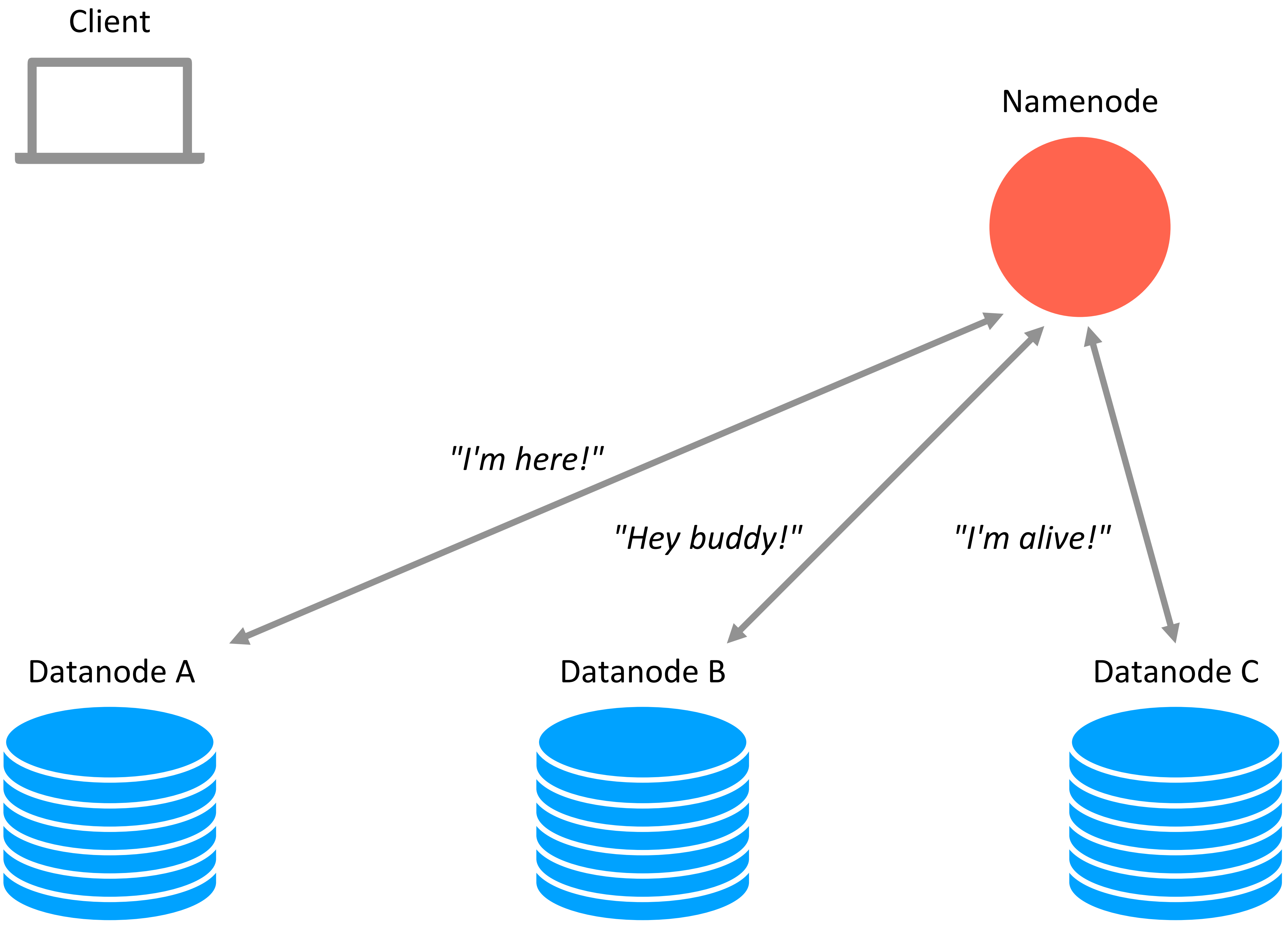
Namenodes & Datanodes

- Master/slave architecture
- HDFS cluster consists of a single Namenode, a master server that manages the file system namespace and regulates access to files by clients.
  - Maintains filesystem tree
  - Files metadata
  - File-to-block mapping
  - Location of blocks (i.e., on which datanodes)
  - Access permissions
- There are a number of Datanodes, usually one per node in a cluster. Datanodes store and manage the actual data blocks.
  - A file is split into one or more blocks, and blocks are stored in Datanodes.
  - The Datanodes manage storage attached to the nodes that they run on.
  - Datanodes serve read/write requests, perform block creation, deletion, and replication upon instruction from Namenode.
  - Each Datanode sends a heartbeat message periodically to the Namenode.

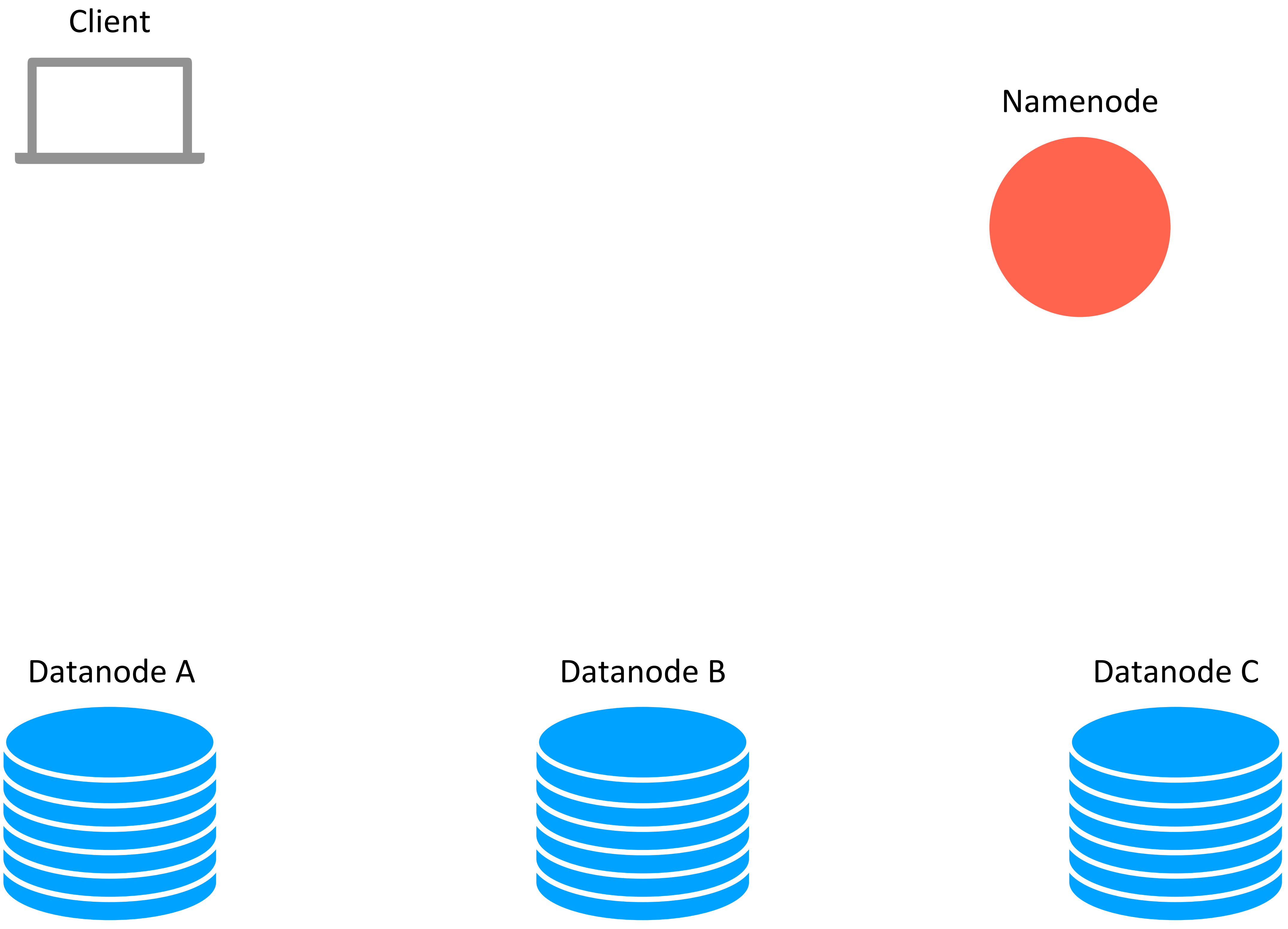
HDFS Architecture



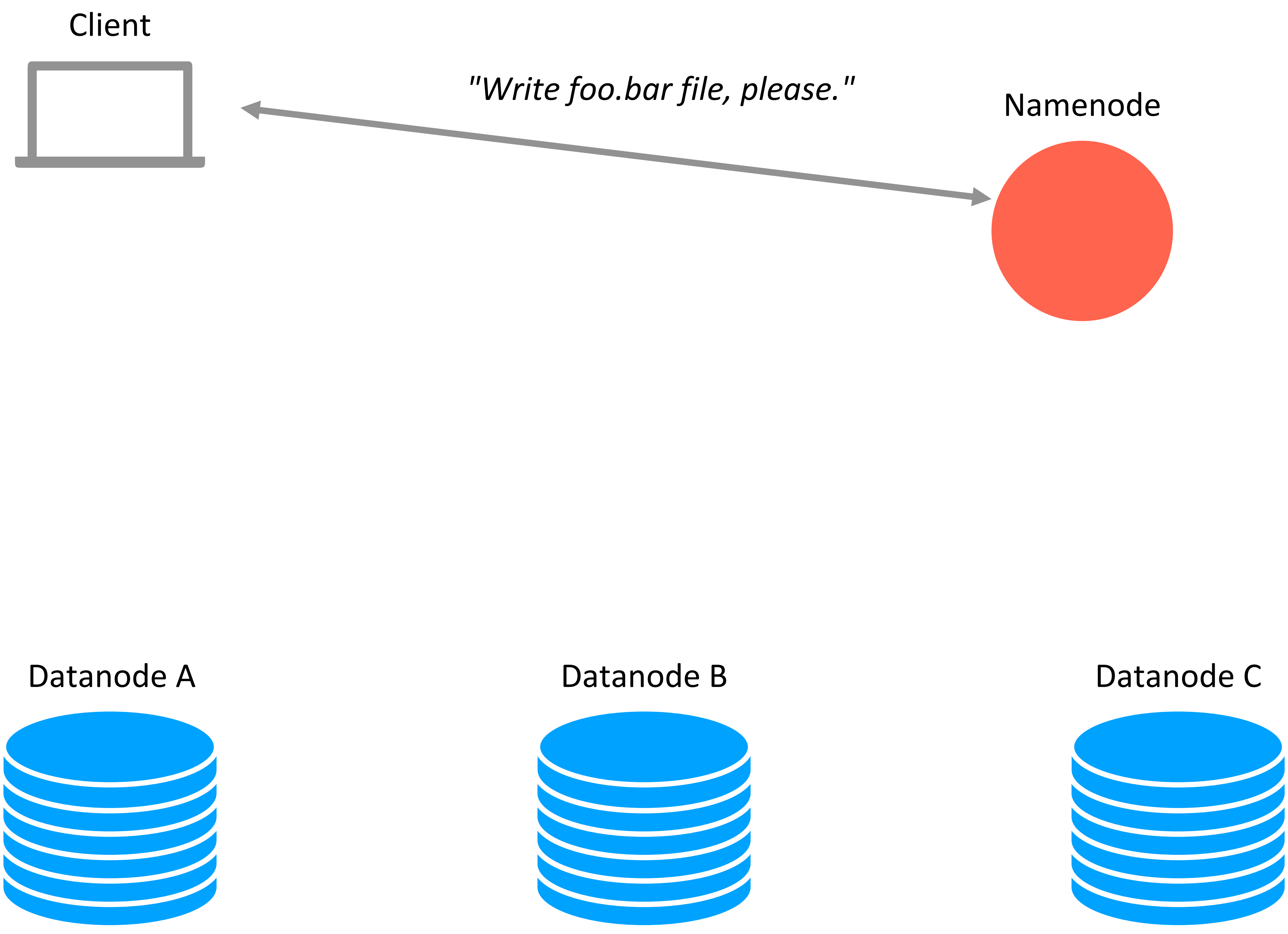
HDFS Architecture



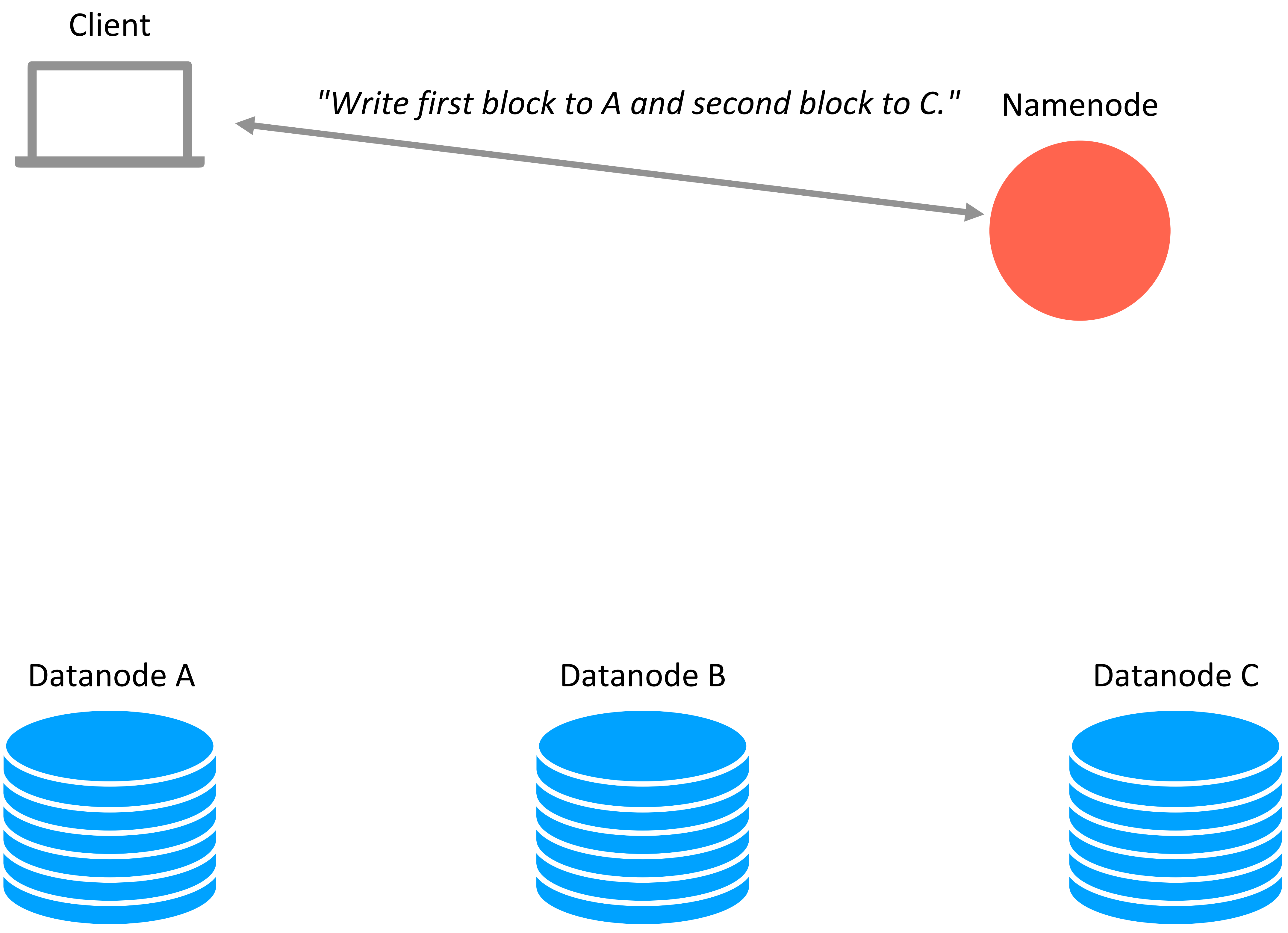
HDFS Architecture



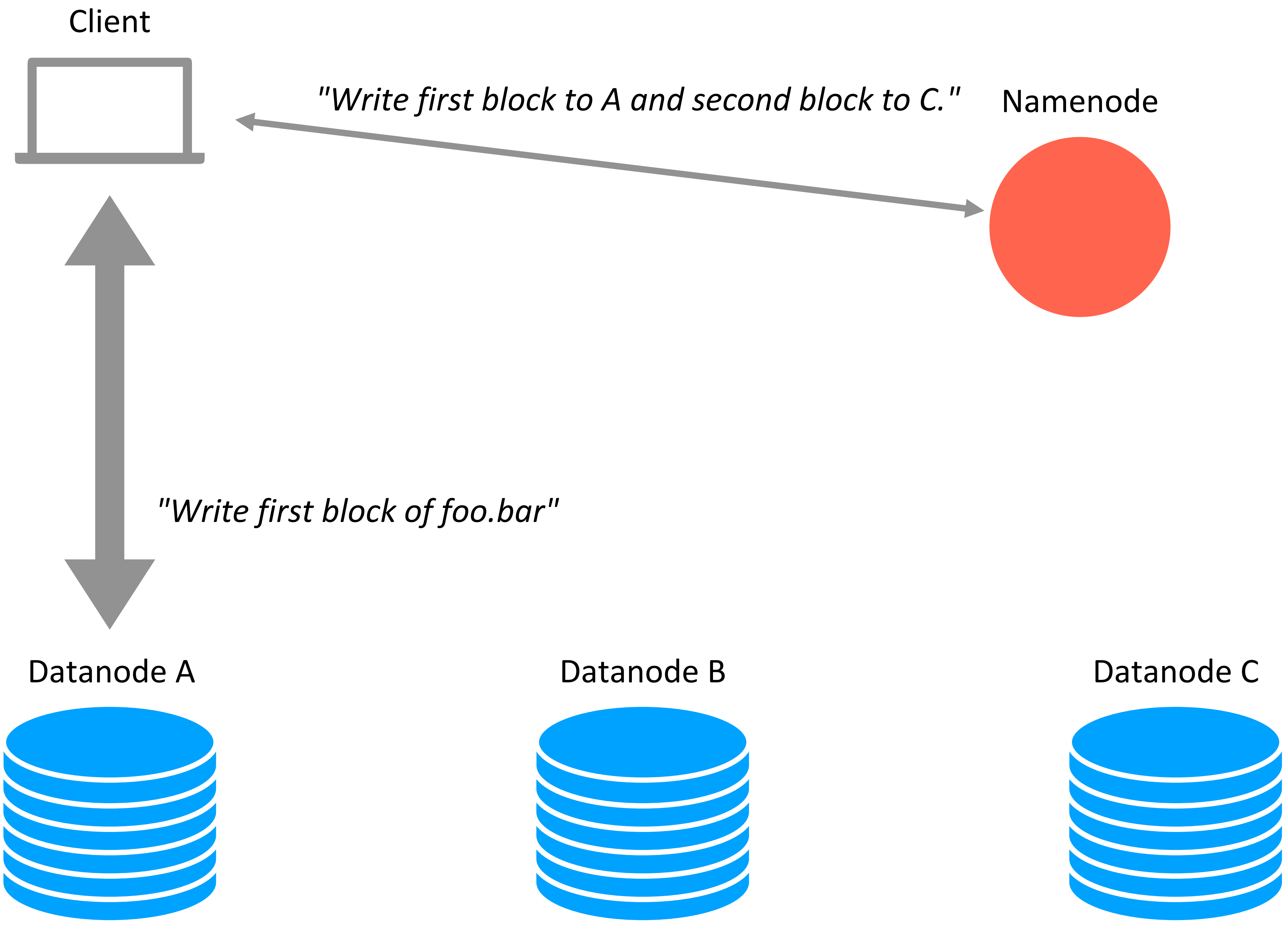
HDFS Architecture



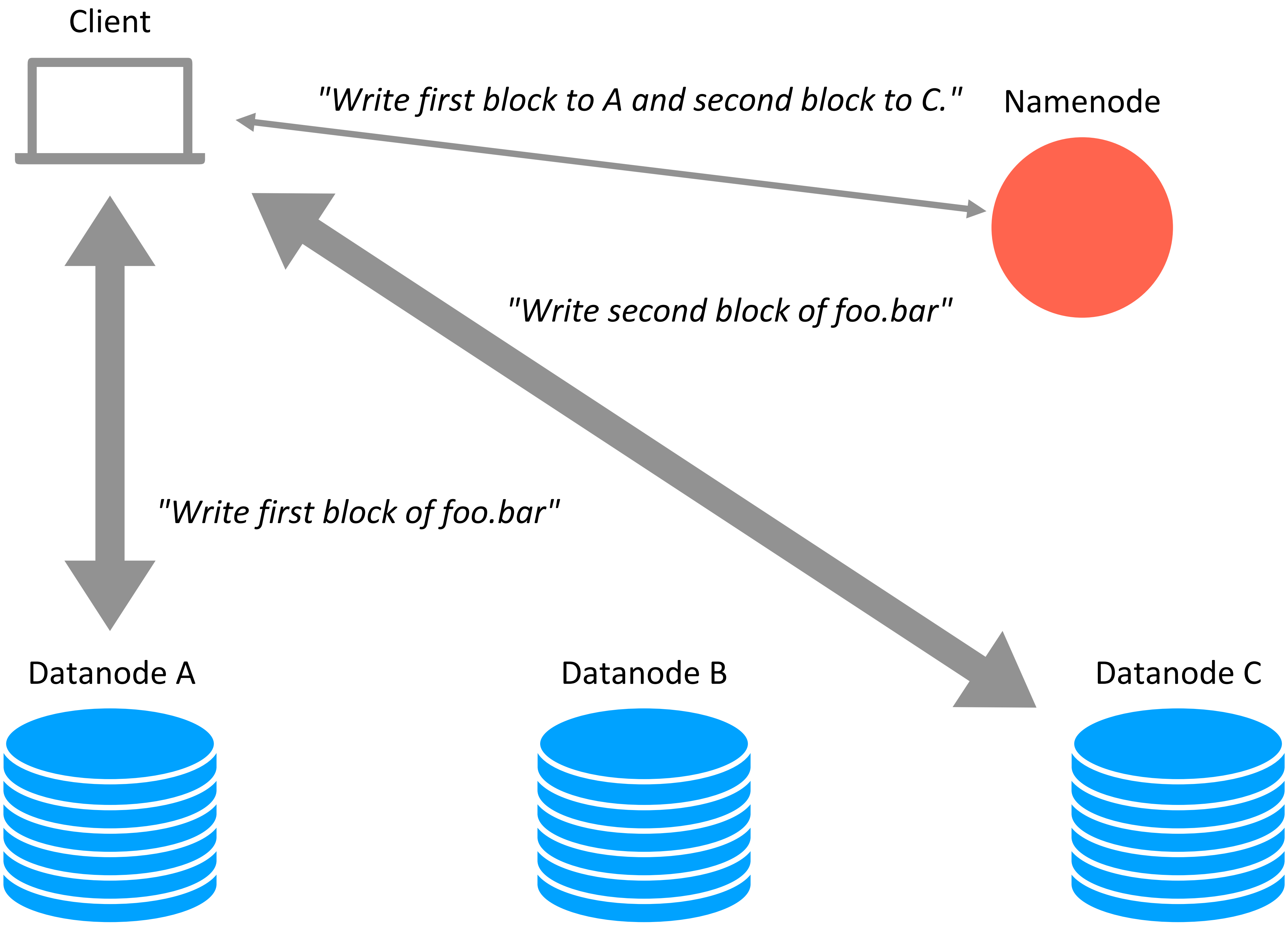
HDFS Architecture



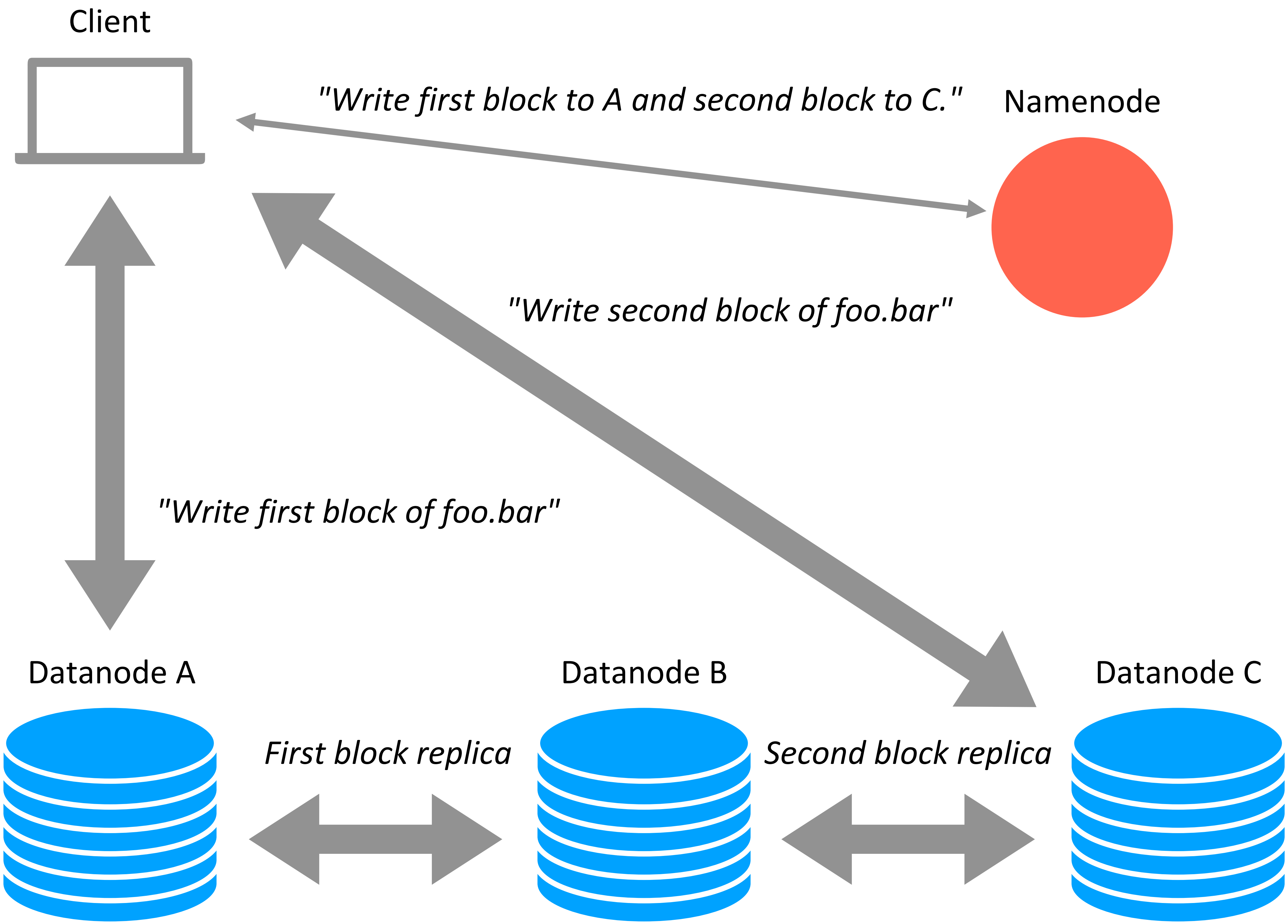
HDFS Architecture



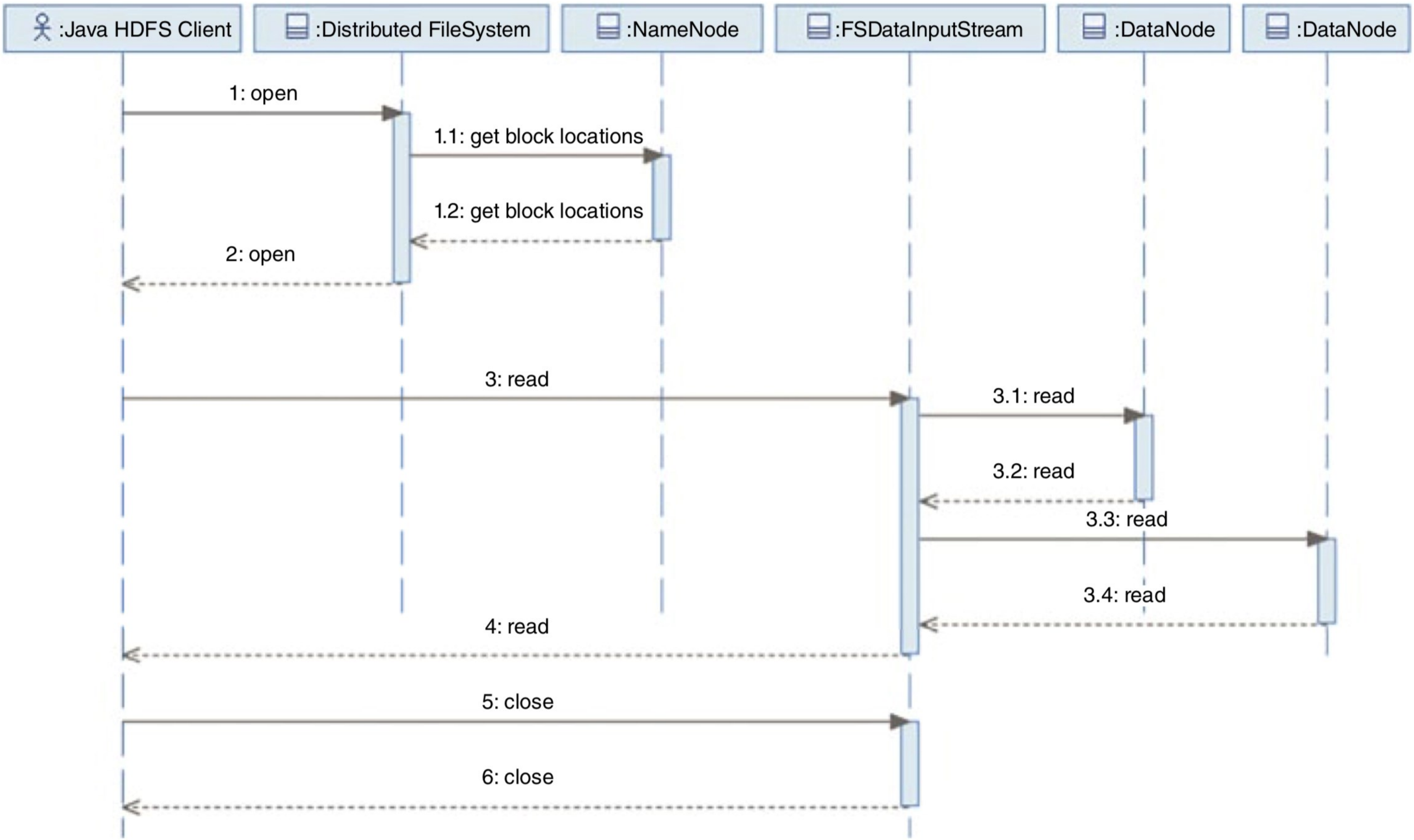
HDFS Architecture



HDFS Architecture

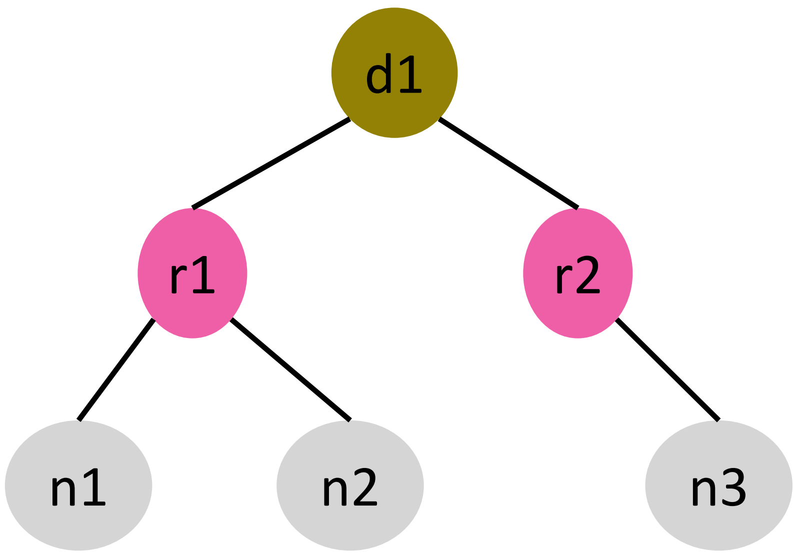


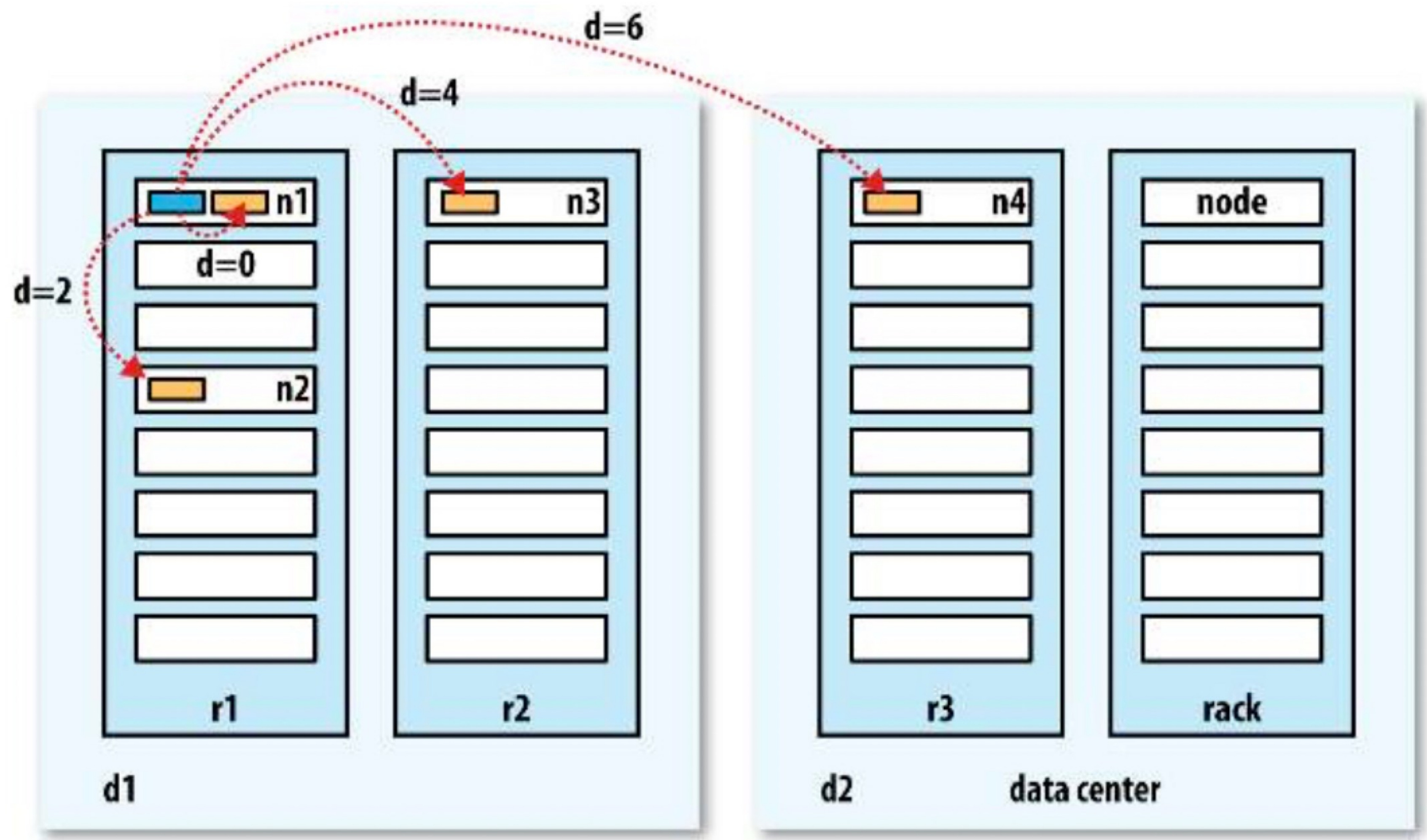
Anatomy of a read



Network Topology and Hadoop

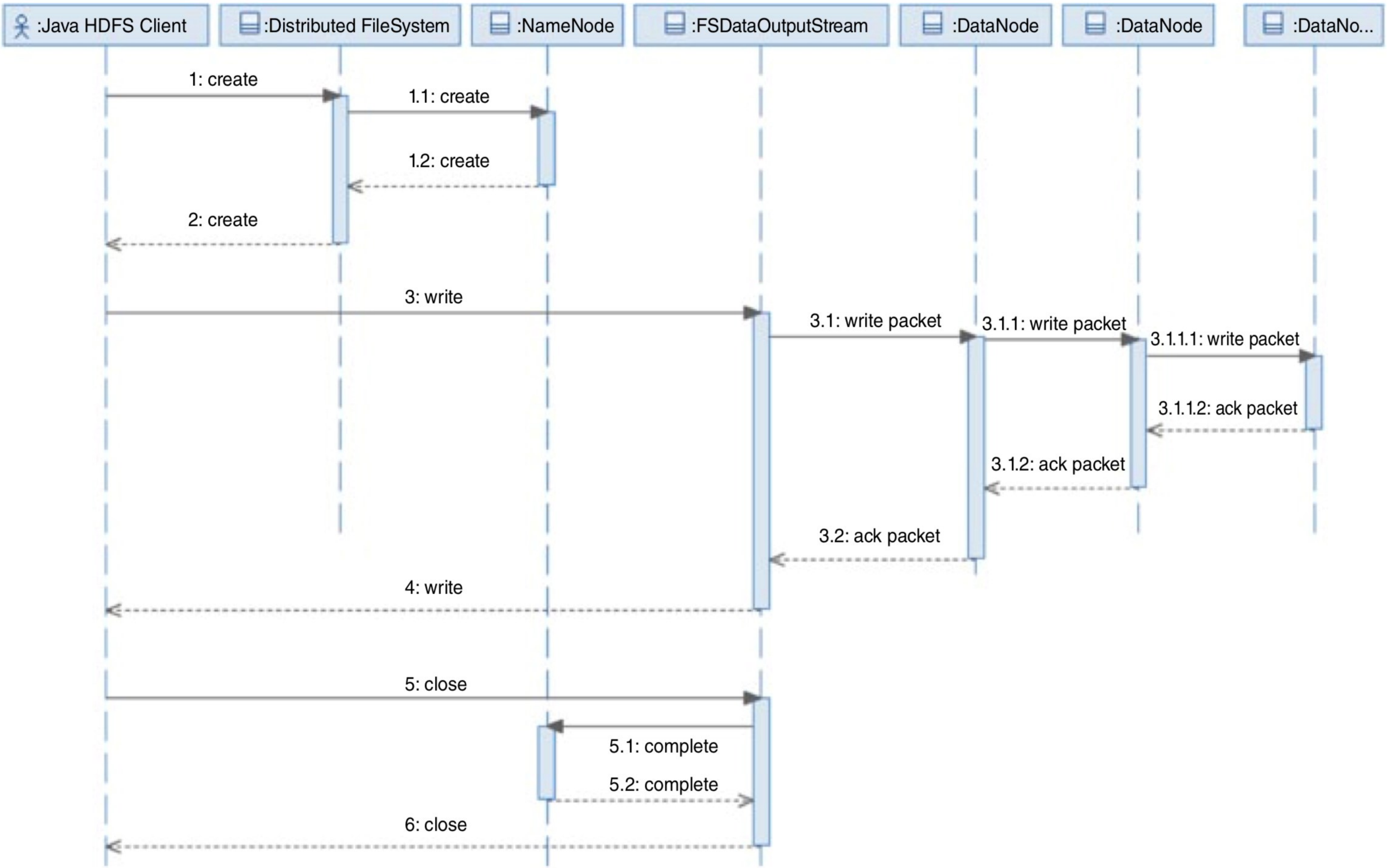
Hadoop approach – The network is represented as a tree and the distance between two nodes is the sum of their distances to their closest common ancestor.





Source: Tom White, "*Hadoop - The Definitive Guide (4 ed)*", O'Reilly, 2015

Anatomy of a write



Replication on Datanodes

How does the namenode choose which datanodes to store the replicas on?

- 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

Resource management in Hadoop versions

Hadoop 1.0

Map Reduce
(data processing)

HDFS
(data storage)

Hadoop 2.0, 3.0

Map Reduce
(data processing)

YARN
(cluster resource management)

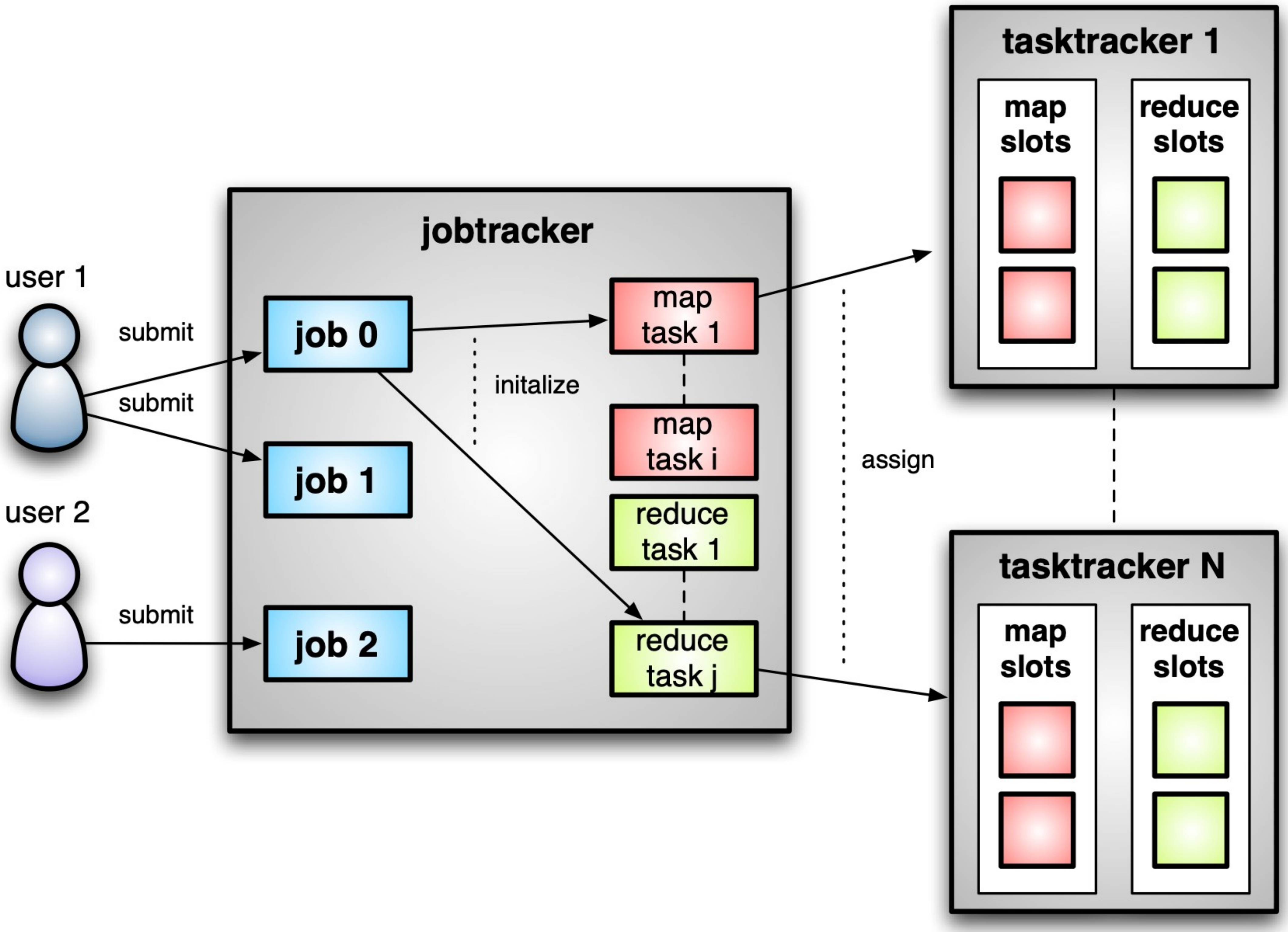
HDFS
(data storage)

Hadoop 1.0

- Job
  - Unit of work that the client wants to be performed
- Task
  - Unit of work that Hadoop schedules and runs run on nodes in the cluster (map & reduce)
- Slot
  - Processing element for tasks (map & reduce slots)
- Job Tracker
  - Accepts jobs submitted by users
  - Creates tasks
  - Assigns map and reduce tasks to Task Trackers
  - Monitors tasks and Task Trackers status, keeping a record of progress for each job and re-executing tasks upon failure
- Task Tracker
  - Runs map and reduce tasks upon instruction from the Job Tracker
  - Manages storage and transmission of intermediate output
  - Sends progress reports to the Job Tracker

These are the main problems in Hadoop 1.0

Hadoop 1.0



tasktracker 1
map slots
reduce slots
jobtracker
user 1
submit
job 0
map task 1
initalize
submit
map task i
assign
job 1
user 2
reduce task 1
tasktracker N
submit
job 2
reduce task j
map slots
reduce slots

Hadoop 1.0 Limitations

- Scalability
  - Job Tracker performs resource allocation and monitoring for all the jobs
  - No more than 4,000 nodes and 40,000 concurrent tasks (whereas with YARN, it goes up to 10,000 nodes and 100,000 tasks)
- Availability
  - Job tracker is a single point of failure
  - Any failure kills all queued and running jobs
  - Replicating the state of this component to achieve availability can be complex
- Resource Utilization
  - Due to the predefined number of map and reduce slots for each Task Tracker, utilization issues occur, e.g., a reduce task has to wait because only map slots are available in the cluster
  - Furthermore, a slot can be too big (waste of resources) or too small (which may cause a failure) for a particular task

YARN Components

- YARN provides its core services via two types of long-running daemons
  - a Resource Manager (one per cluster) to manage the use of resources across the cluster
  - Node Managers (one per node in the cluster) to launch containers and monitor usage of container resources, reporting stats to resource manager
- A container is a set of computer resources allocated to run an application-specific process (e.g., a map or reduce task).
- 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 (e.g., allocate container on a node where there is a replica of the HDFS block)
- If the locality constraint cannot be met
  - no allocation is made or
  - the constraint can be loosened (e.g., on another node in the same rack)