# 分布式计算

## The Hadoop Distributed File System(HDFS)

Weixiong Rao 饶卫雄
Tongji University 同济大学软件学院
2023 秋季
wxrao@tongji.edu.cn

# Outline

- **Introduction**
- **Architecture**

  NameNode, DataNodes, HDFS Client, CheckpointNode, BackupNode, Snapshots
- **File I/O Operations and Replica Management**

  File Read and Write, Block Placement, Replication management, Balancer,
- **FUTURE WORK**

# 安装

http://hadoop.apache.org

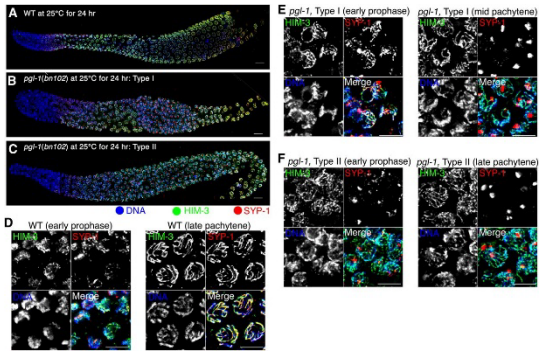
- HDFS作为Apache Hadoop子模块
- Apache Hadoop
  - 通用硬件计算机集群之上的开源软件框架: 大规模数据存储和处理
  - 由Doug Cutting 和 Mike Carafella 于2005发起.
  - Hadoop名字的启由：Cutting 儿子的玩具大象

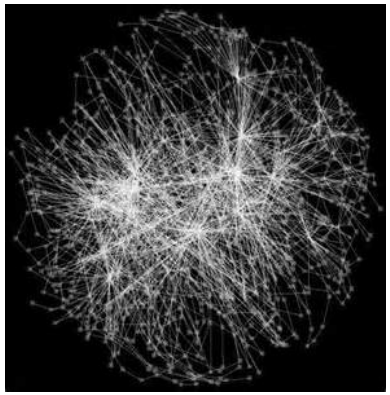
# Hadoop用户

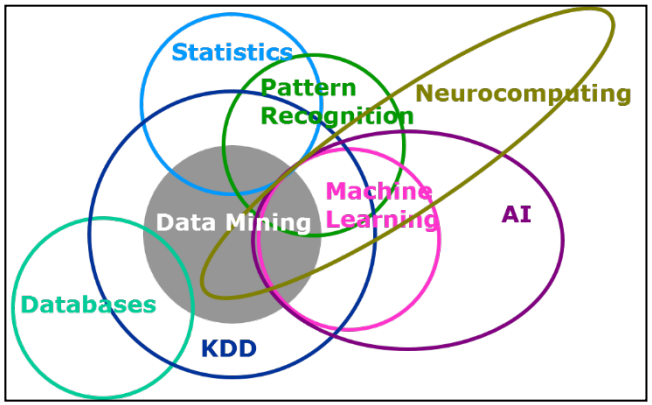

facebook
The New York Times
ebay
hp
IBM
JPMorganChase
eHarmony
twitter
intel
NETFLIX
rackspace
HOSTING
amazon.com
VISA
NING
SAMSUNG
YAHOO!

# Hadoop用途

- 文本处理: Data-intensive text processing
- 生物信息: Assembly of large genomes
- 图挖掘: Graph mining
- 机器学习/数据挖掘: Machine learning & data mining
- 社交网络: Large scale social network analysis





# Hadoop生态系统

| | |
|---|---|
| Hadoop Common | • Contains Libraries and other modules |
| HDFS | • Hadoop Distributed File System |
| Hadoop YARN | • Yet Another Resource Negotiator |
| Hadoop MapReduce | • A programming model for large scale data processing |

# Introduction

## HDFS

**The Hadoop Distributed File System (HDFS)** is the file system component of Hadoop. It is designed to

① store very large data sets ***reliably***,

② and to stream those data sets at ***high bandwidth*** to user applications.

These are achieved by **replicating file content** on multiple machines (DataNodes).

# Outline

- Introduction
- **Architecture**

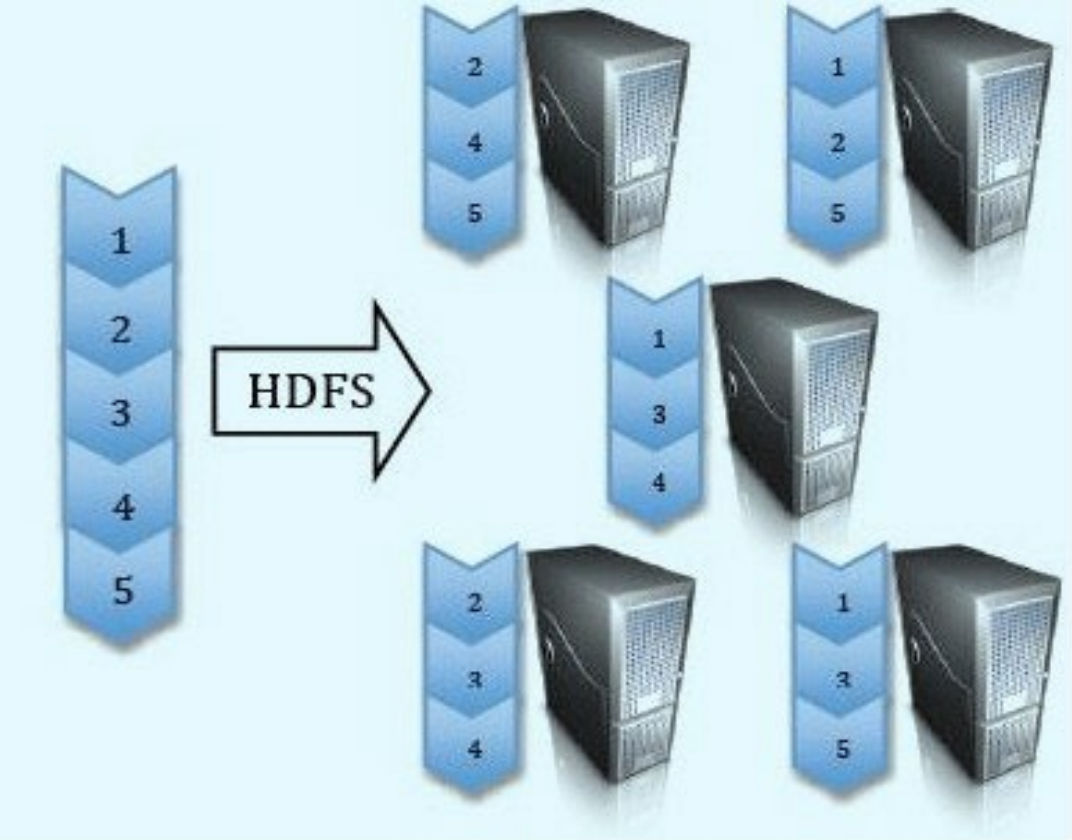
  NameNode, DataNodes, HDFS Client, CheckpointNode, BackupNode, Snapshots
- File I/O Operations and Replica Management

  File Read and Write, Block Placement, Replication management, Balancer,
- Practice at YAHoo!
- FUTURE WORK

# Architecture

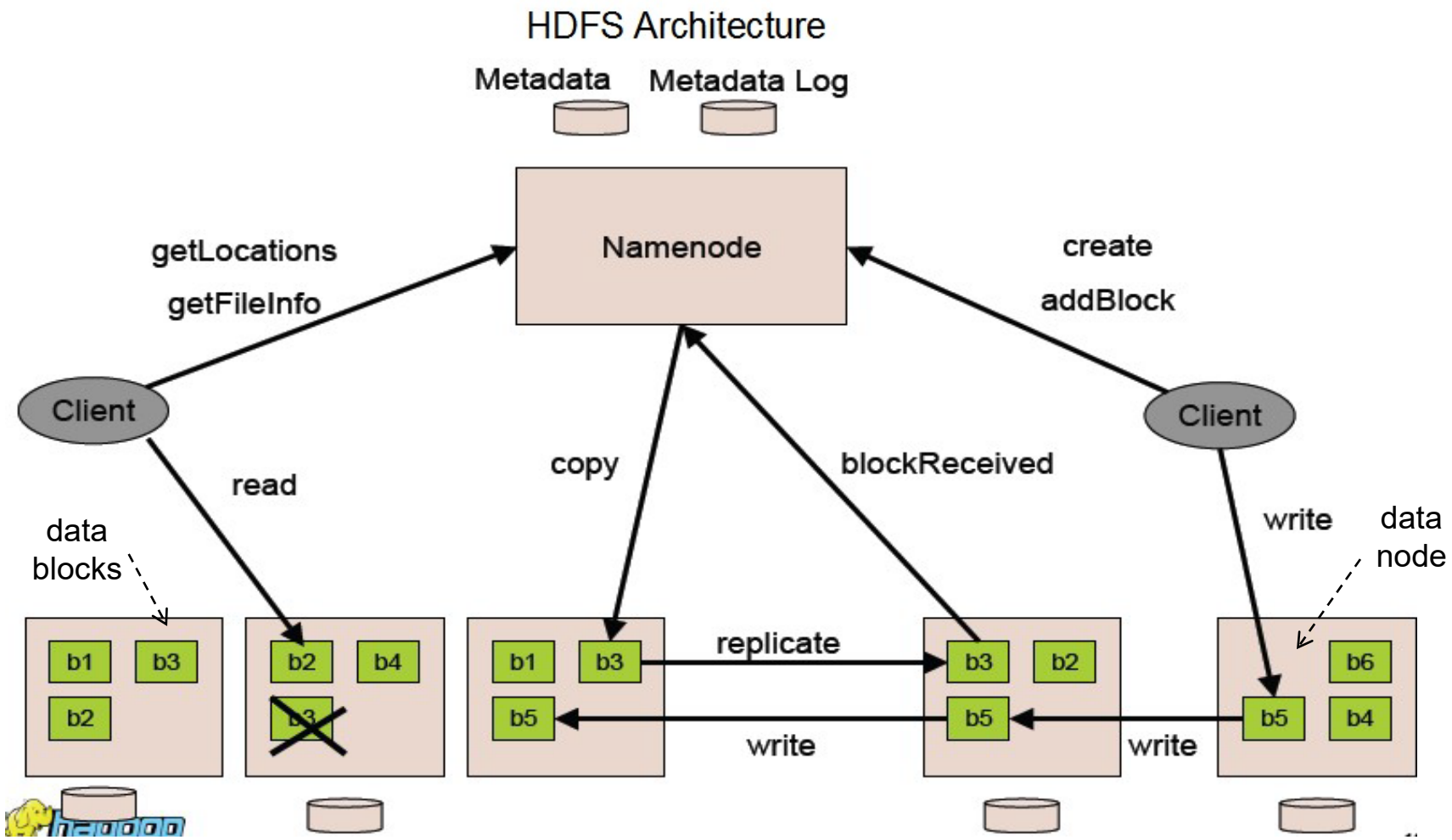
- HDFS is a block-structured file system:
  - Files broken into blocks of 128MB (per-file configurable).
- A file can be made of several blocks, and they are stored across a cluster of one or more machines with data storage capacity.
- Each block of a file is replicated across a number of machines to prevent loss of data.

# Architecture

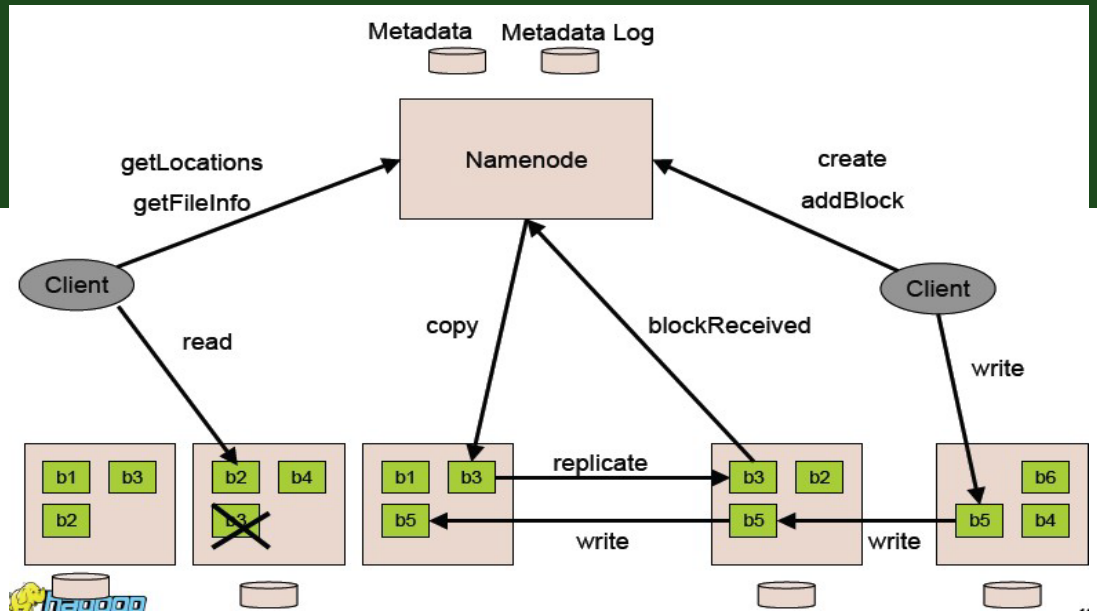



What is the benefit of replication?

# Architecture



# Architecture




## NameNode
## DataNodes

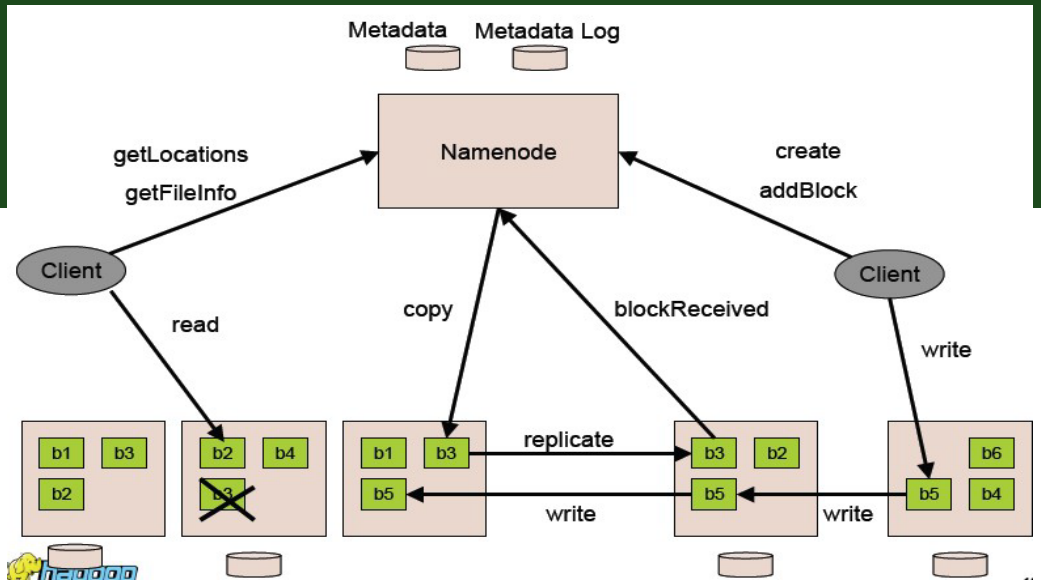
- HDFS stores file system metadata and application data separately.
- **Metadata** refers to file metadata(attributes such as permissions, modification, access times, namespace and disk space quotas.) called “inodes”+list of blocks belong to the file.

- HDFS stores metadata on a dedicated server, called the NameNode.(Master)
- Application data are stored on other servers called DataNodes.(Slaves)

- All servers are fully connected and communicate with each other using TCP-based protocols.(RPC)

# Architecture

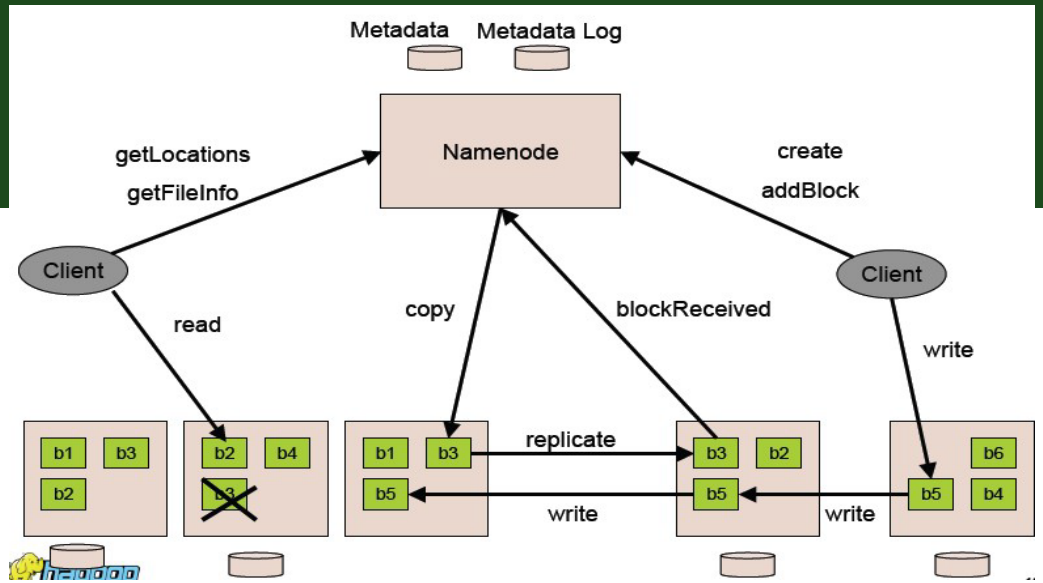
## Single Namenode




- Maintain the namespace tree (a hierarchy of files and directories) operations like opening, closing, and renaming files and directories.
- Determine the mapping of file blocks to DataNodes (the physical location of file data).
- File metadata (i.e. “inode”) .
- Authorization and authentication.
- Collect block reports from Datanodes on block locations.
- Replicate missing blocks.

- HDFS keeps the entire namespace in RAM, allowing fast access to the metadata.

# Architecture

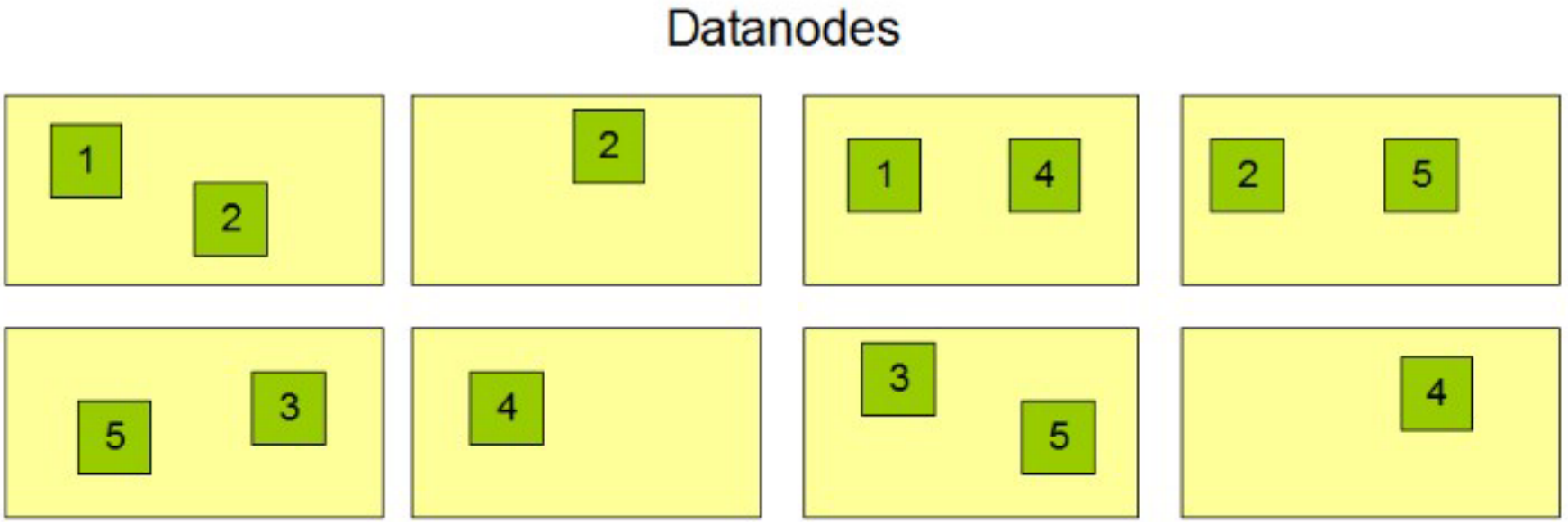



## DataNodes

- The DataNodes are responsible for serving read and write requests from the file system's clients.
- The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode.
- Data nodes periodically send block reports to Namenode.

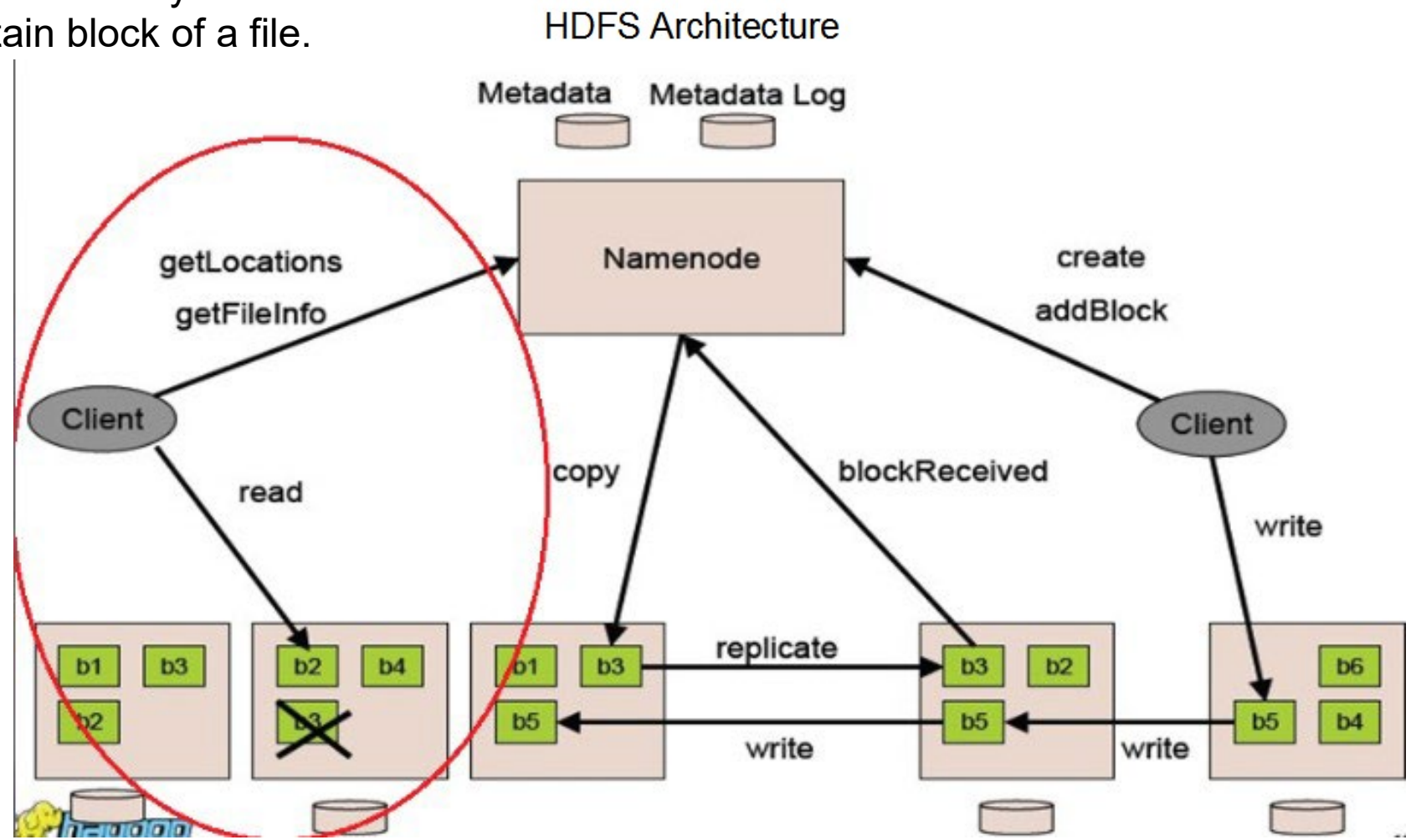
# Architecture

Namenode (Filename, numReplicas, block-ids, …)
/users/sameerp/data/part-0, r:2, {1,3}, …
/users/sameerp/data/part-1, r:3, {2,4,5}, …

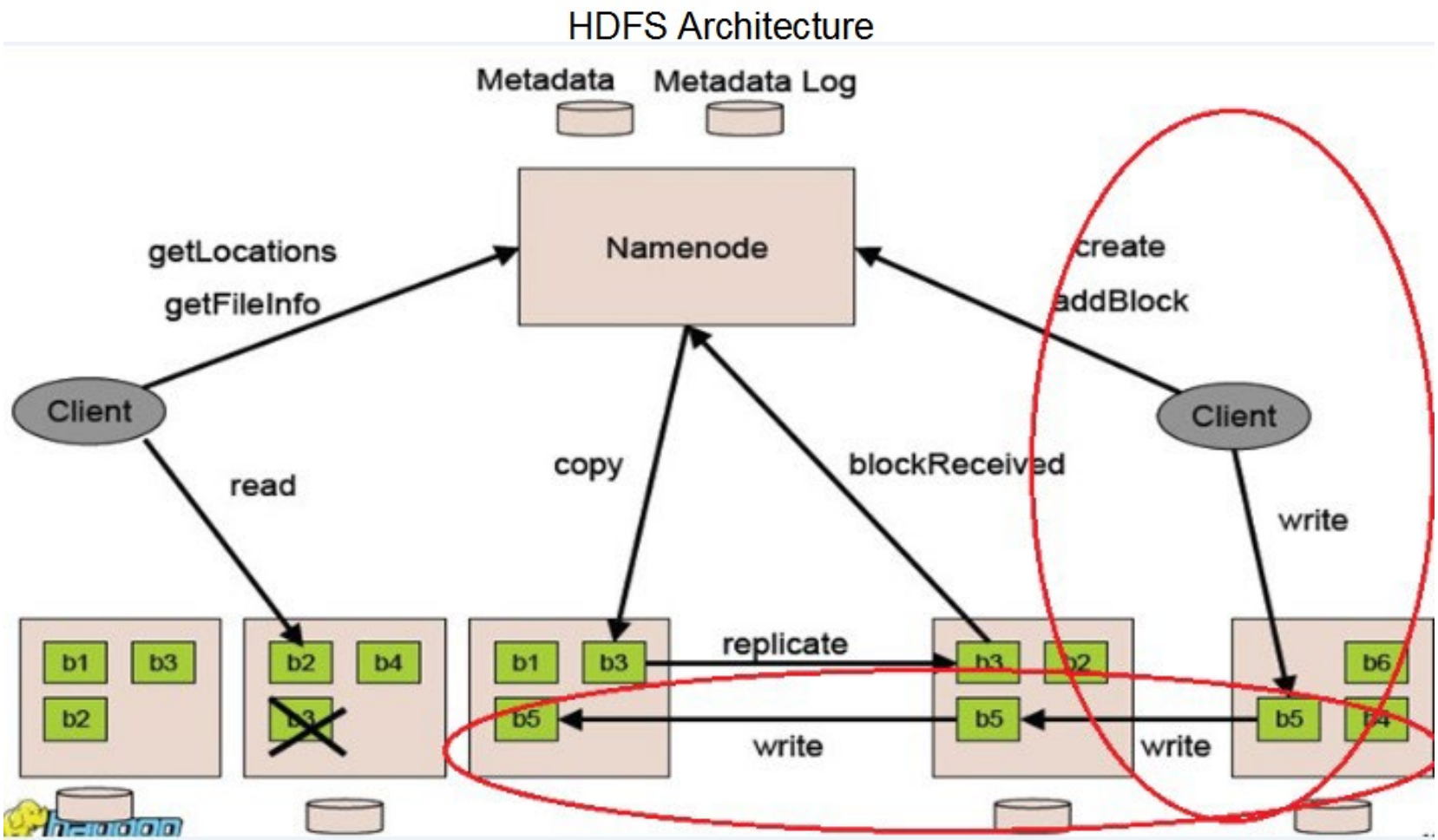


# Architecture

**READ**:
User has directly access to certain block of a file.

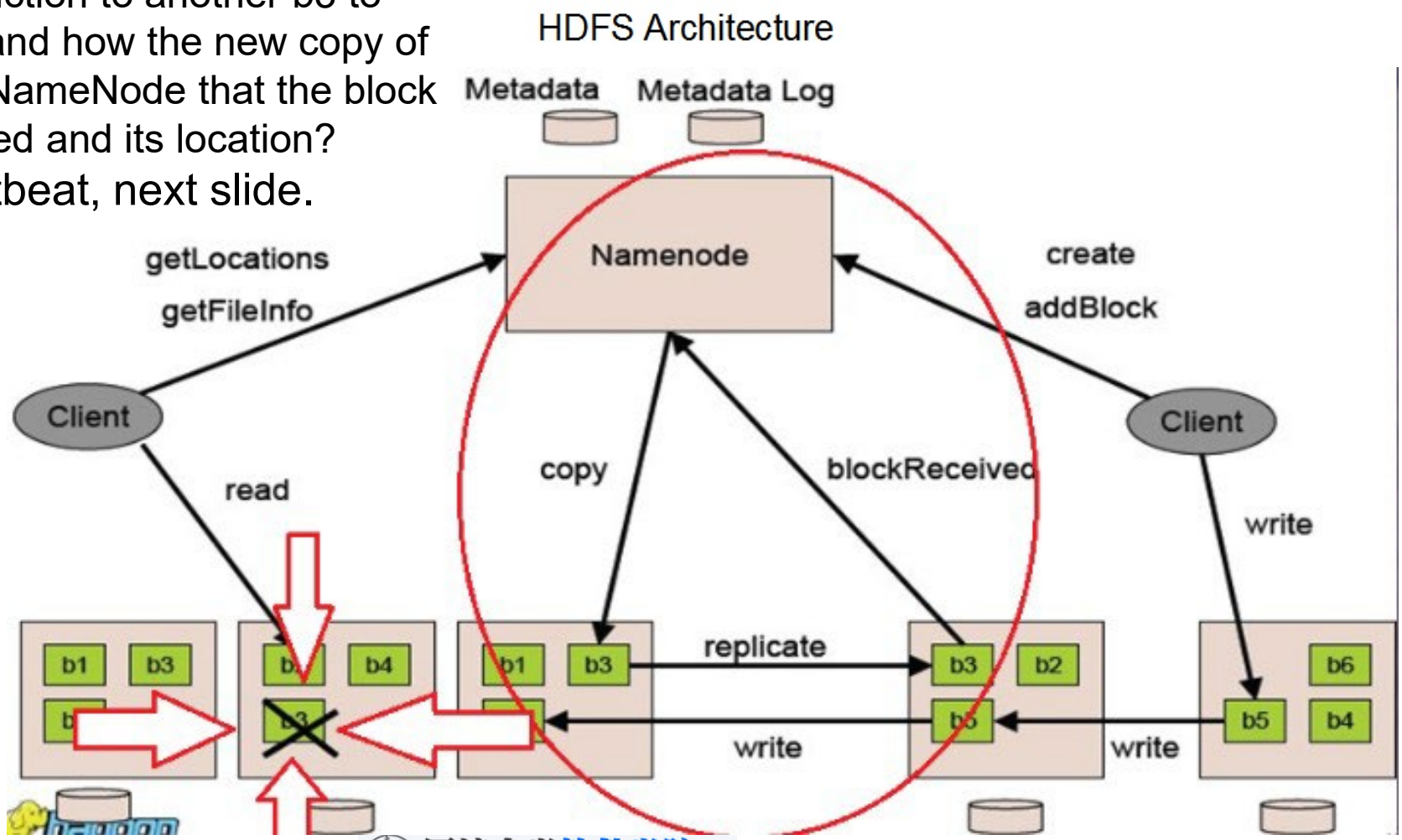


# Architecture



# Architecture

Question: How do NameNode know that b3 is dead so he can give instruction to another b3 to replicate and how the new copy of b3 notify NameNode that the block is replicated and its location?
---- Heartbeat, next slide.

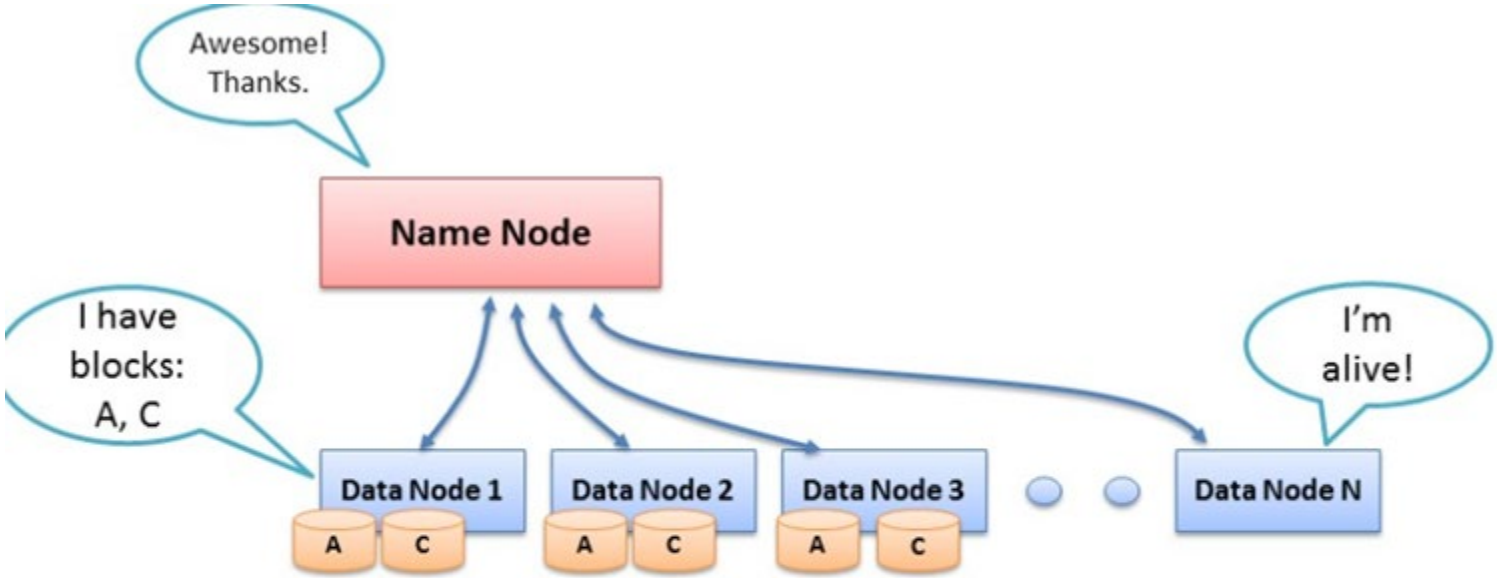


# Architecture

- NameNode and DataNode communication: *Heartbeats.*

- DataNodes send *heartbeats* to the NameNode to confirm that **the DataNode is operating** and **the block replicas it hosts are available.**

# Architecture




- Data Node sends Heartbeats
- Every 10th heartbeat is a Block report
- Name Node builds metadata from Block reports
- TCP – every 3 seconds

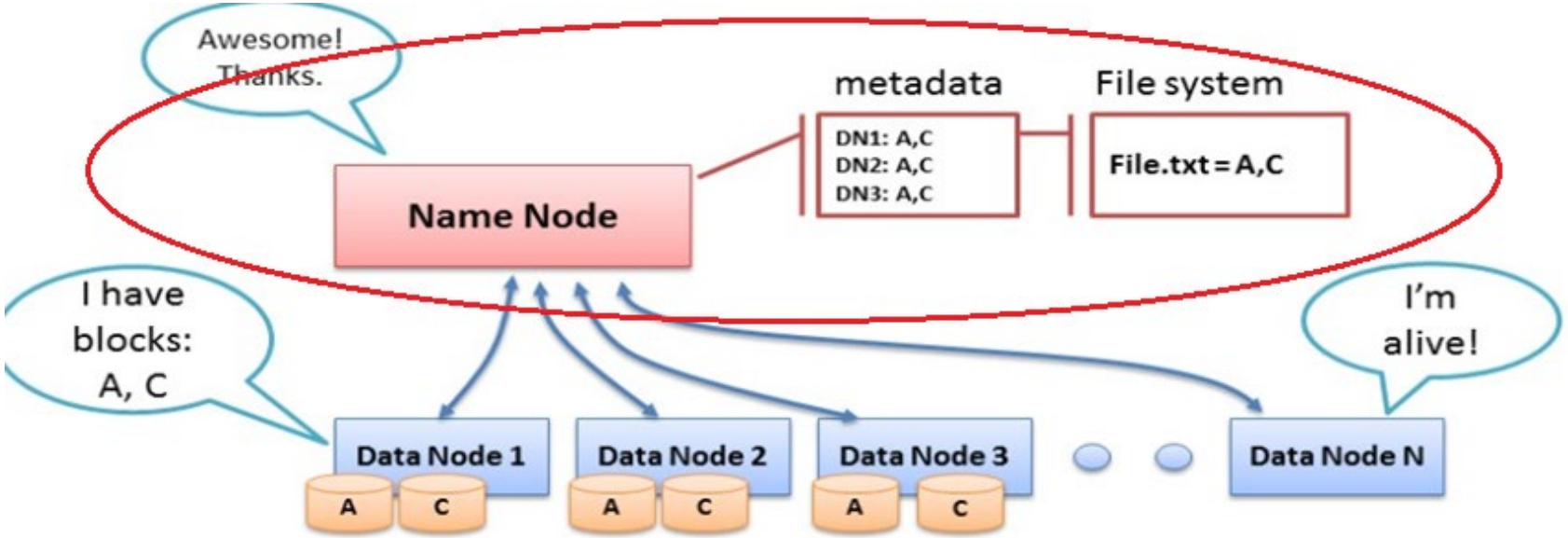
# Architecture

## Blockreports

- A DataNode identifies block replicas in its possession to the NameNode by sending a *block report*. A block report contains the ***block id***, the ***generation stamp*** and the **length for each block replica** the server hosts.

- Blockreports provide the NameNode *with an up-to-date view* of where *block replicas* are located on the cluster and nameNode constructs and maintains latest metadata from blockreports.

# Architecture



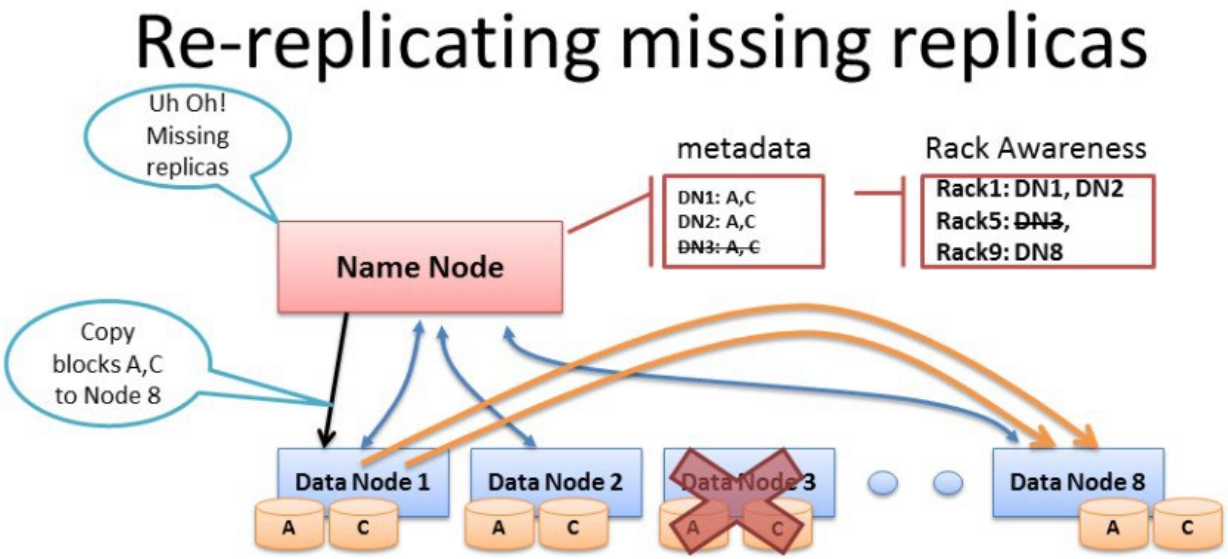

- Data Node sends Heartbeats
- Every 10th heartbeat is a Block report
- Name Node builds metadata from Block reports
- TCP – every 3 seconds

# Architecture

## Failure recovery

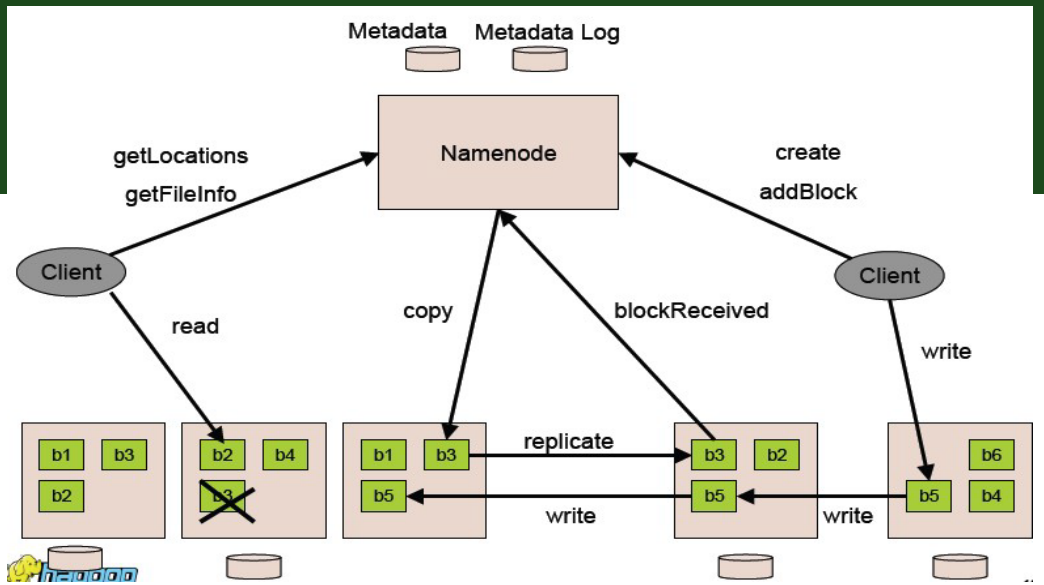
- The NameNode does not directly call DataNodes. It uses replies to heartbeats to send instructions to the DataNodes.
- The instructions include commands to:
  - replicate blocks to other nodes:
    - DataNode died.
    - copy data to local.
  - remove local block replicas;
  - re-register or to shut down the node;

# Architecture




- Missing Heartbeats signify lost Nodes
- Name Node consults metadata, finds affected data
- Name Node consults Rack Awareness script
- Name Node tells a Data Node to re-replicate

# Architecture




## Failure recovery

- So when dataNode died, NameNode will notice and instruct other dataNode to replicate data to new dataNode.
- What if NameNode died?

# Architecture

## Failure recovery

- Keep journal (the modification log of metadata).
- Checkpoint: The persistent record of the metadata stored in the local host's native files system.

*For example:*

During restart, the NameNode initializes the namespace image from the checkpoint, and then replays changes from the journal until the image is up-to-date with the last state of the file system.

# Architecture

## Failure recovery

- **CheckpointNode** and **BackupNode--**two other roles of NameNode

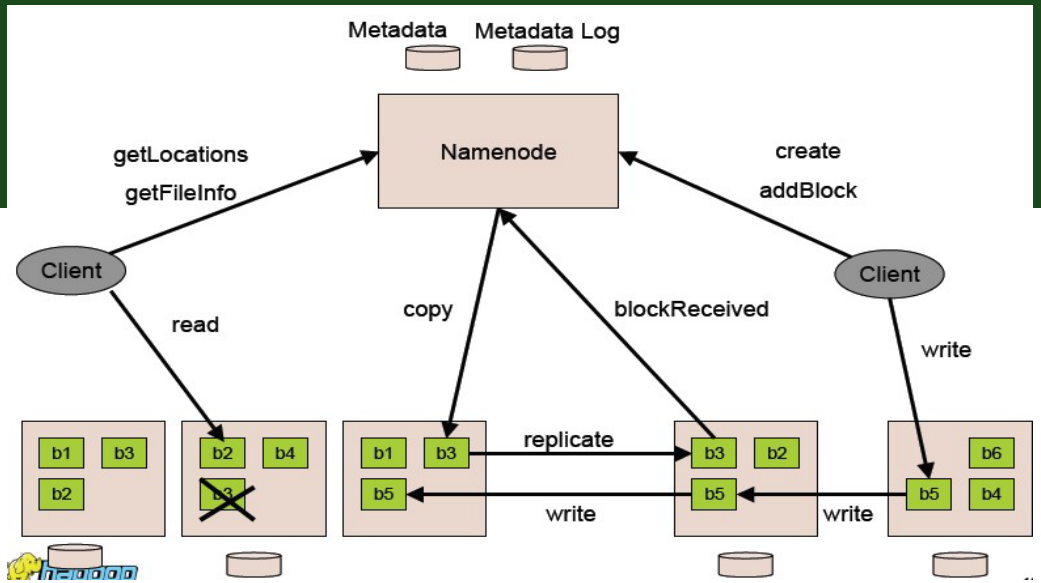
- **CheckpointNode:**
- When journal becomes too long, checkpointNode 合并 **combines** the existing checkpoint and journal to create a new checkpoint and an empty journal.

# Architecture

## Failure recovery

- Upgrades, File System Snapshots
- **The purpose of creating snapshots** in HDFS is to minimize potential damage to the data stored in the system during upgrades. During software upgrades the possibility of corrupting the system due to software bugs or human mistakes increases.
- The snapshot mechanism lets administrators **persistently save the current state of the file system(both data and metadata)**, so that if the upgrade results in data loss or corruption, it is possible to **rollback the upgrade** and return HDFS to the namespace and storage state as they were at the time of the snapshot.

# Architecture




## Failure recovery

- **CheckpointNode** and **BackupNode**--two other roles of NameNode
- **BackupNode:** A read-only NameNode
- it maintains an *in-memory*, up-to-date image of the file system namespace that is always synchronized with the state of the NameNode.
- If the NameNode fails, the BackupNode's image in memory and the checkpoint on disk is a record of the latest namespace state.

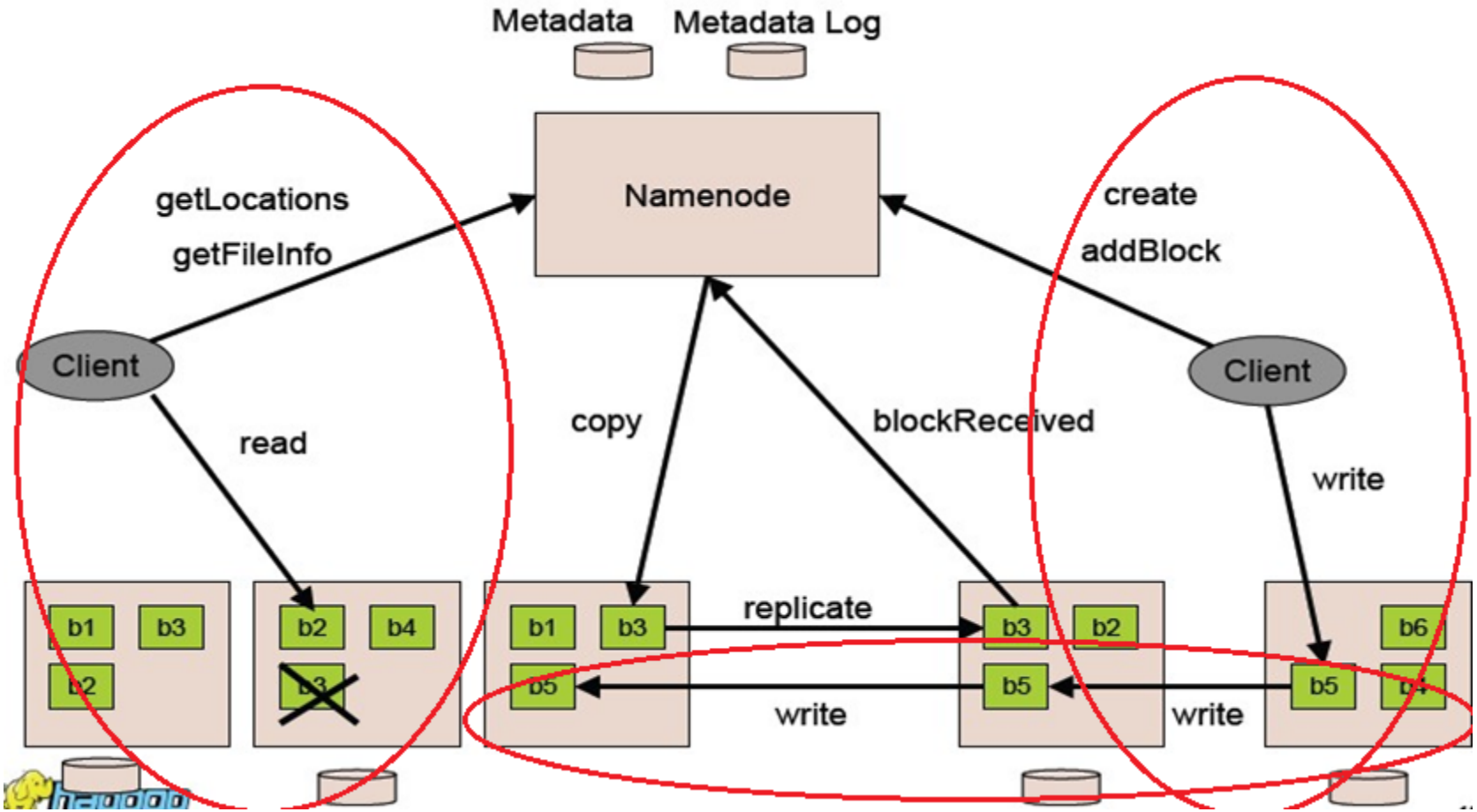
# Outline

- Introduction
- Architecture

  NameNode, DataNodes, HDFS Client, CheckpointNode, BackupNode, Snapshots
- **File I/O Operations and Replica Management**

  File Read and Write, Block Placement, Replication management, Balancer,
- Practice at YAHoo!
- FUTURE WORK

# File I/O Operations and Replica Management



# File I/O Operations and Replica Management

- Hadoop has the concept of “Rack Awareness”.

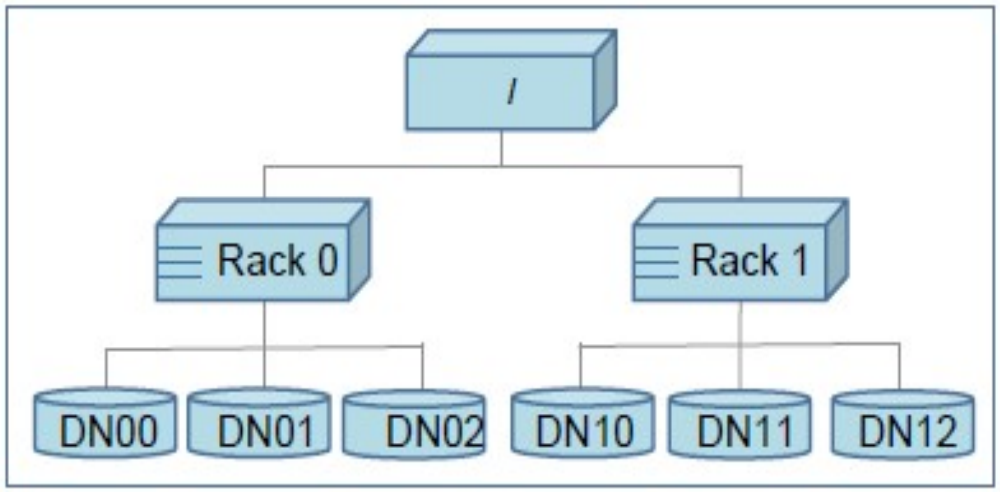



*Figure 3. Cluster topology example*

For a large cluster, it may not be practical to connect all nodes in a flat topology.

- A common practice is to spread the nodes across multiple racks.
- Nodes of a rack share a switch, and rack switches are connected by one or more core switches.
- Communication between two nodes in different racks has to go through multiple switches.

# File I/O Operations and Replica Management

- Hadoop has the concept of “Rack Awareness”.

- The default HDFS replica placement policy can be summarized as follows:

1. No Datanode contains more than one replica of any block.

2. No rack contains more than two replicas of the same block, provided there are sufficient racks on the cluster.

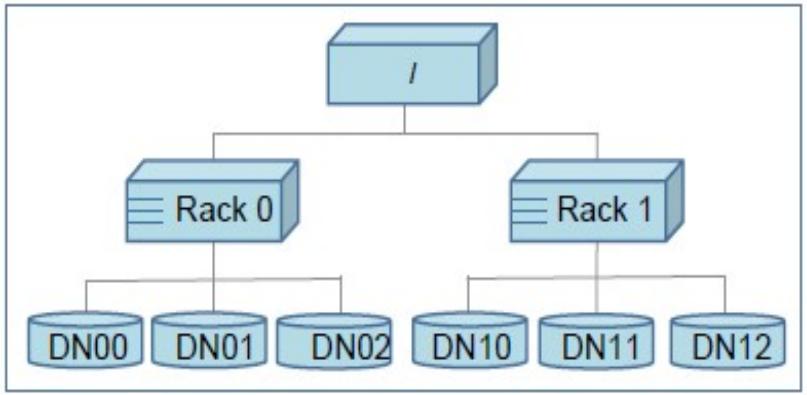
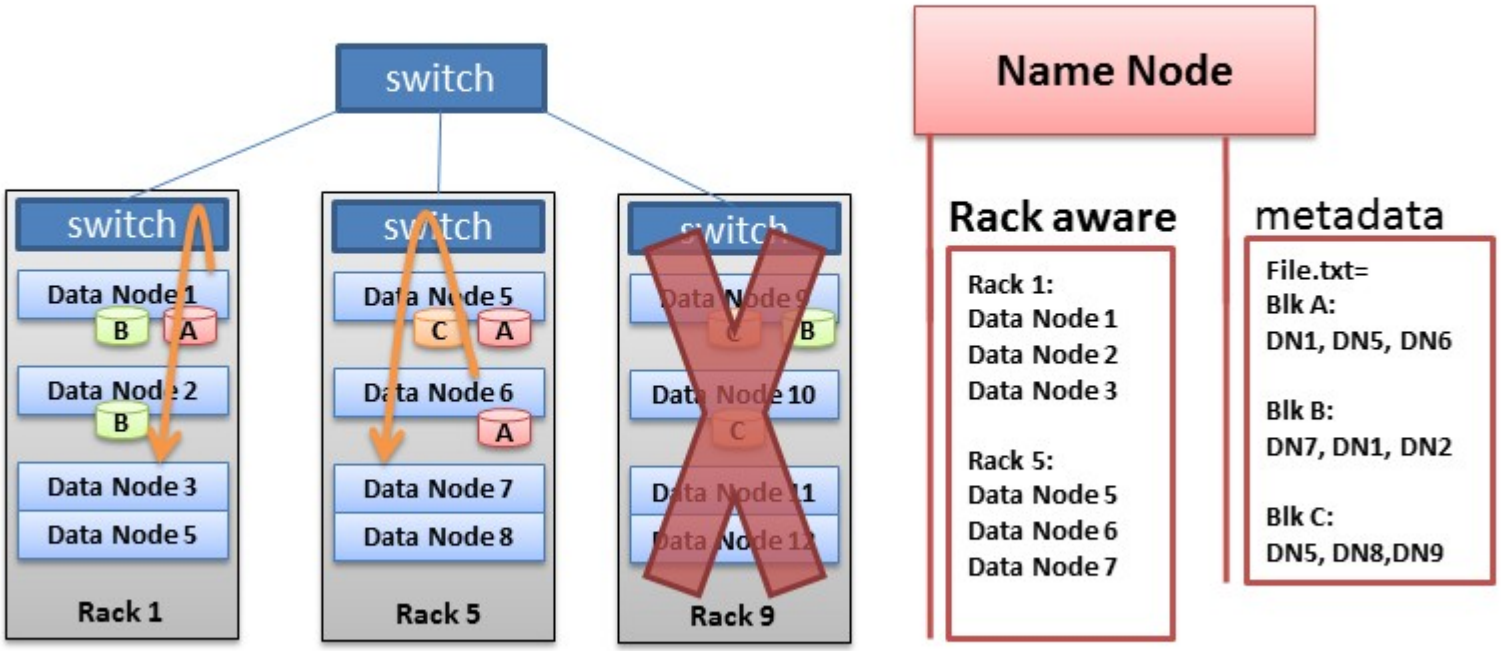



*Figure 3. Cluster topology example*

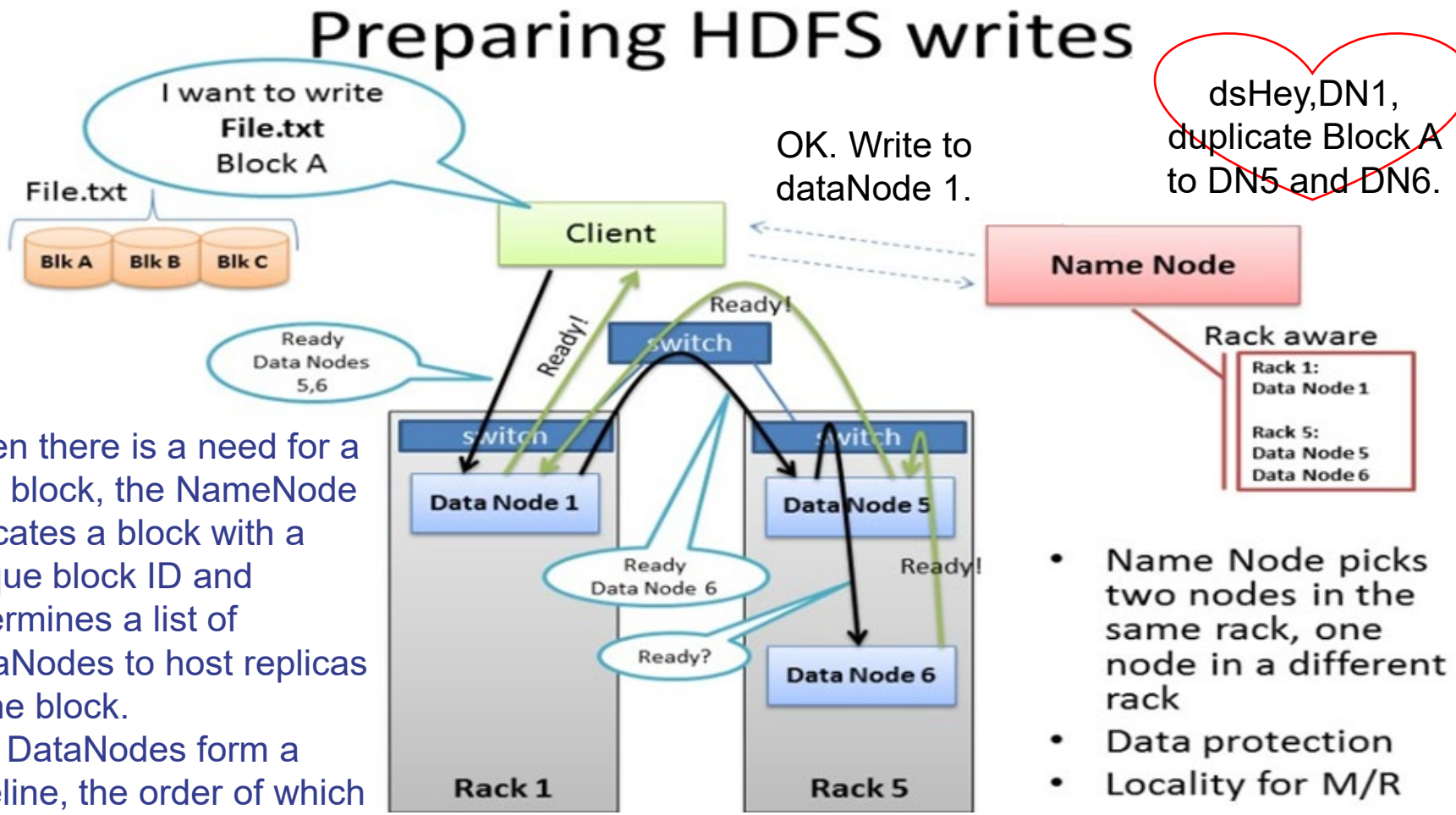
# File I/O Operations and Replica Management

## Hadoop Rack Awareness – Why?




- Never loose all data if entire rack fails

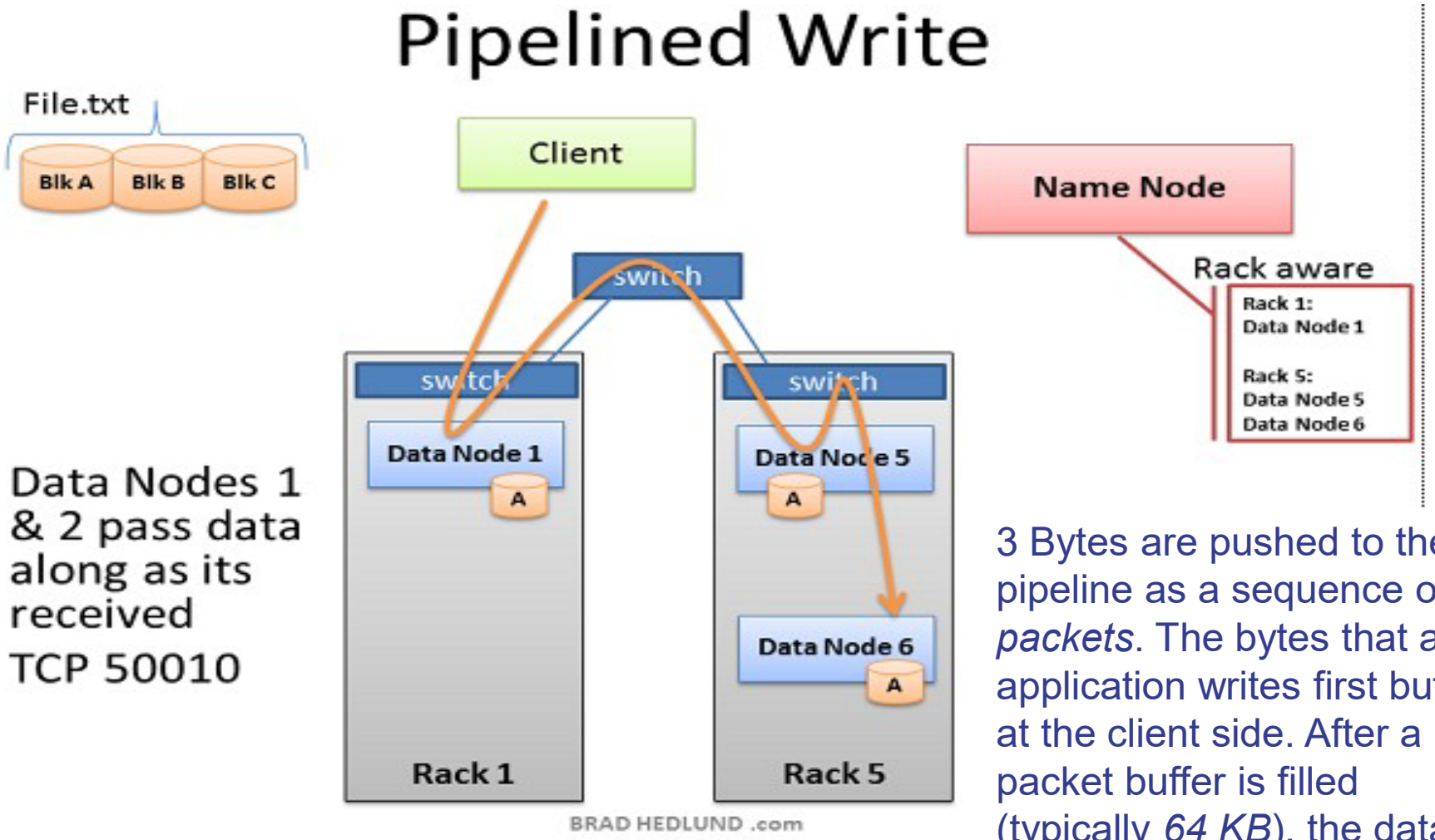
# File I/O Operations and Replica Management




1. When there is a need for a new block, the NameNode allocates a block with a unique block ID and determines a list of DataNodes to host replicas of the block.
2. The DataNodes form a pipeline, the order of which minimizes the total network distance from the client to the last DataNode.

- Name Node picks two nodes in the same rack, one node in a different rack
- Data protection
- Locality for M/R

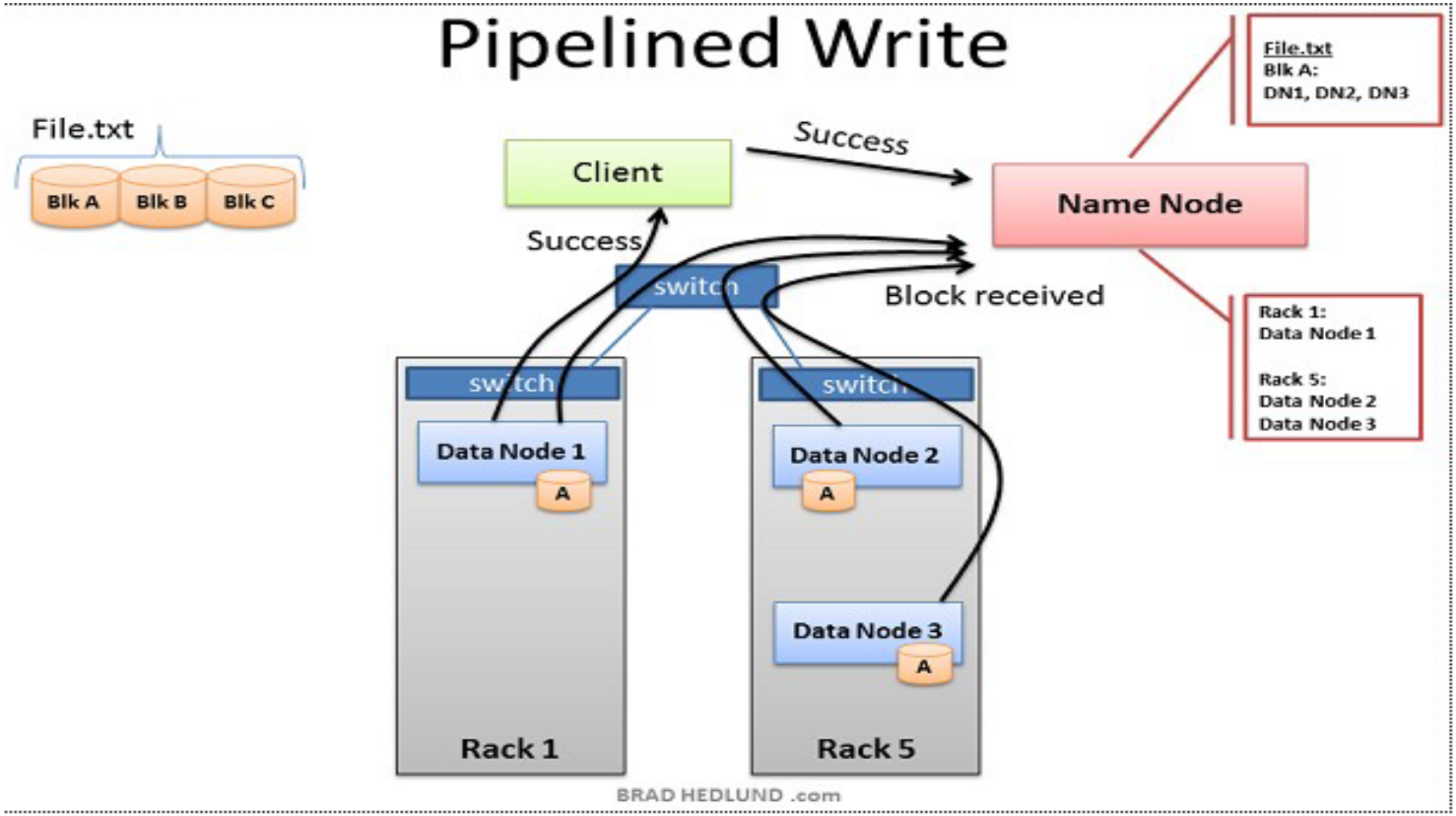
# File I/O Operations and Replica Management



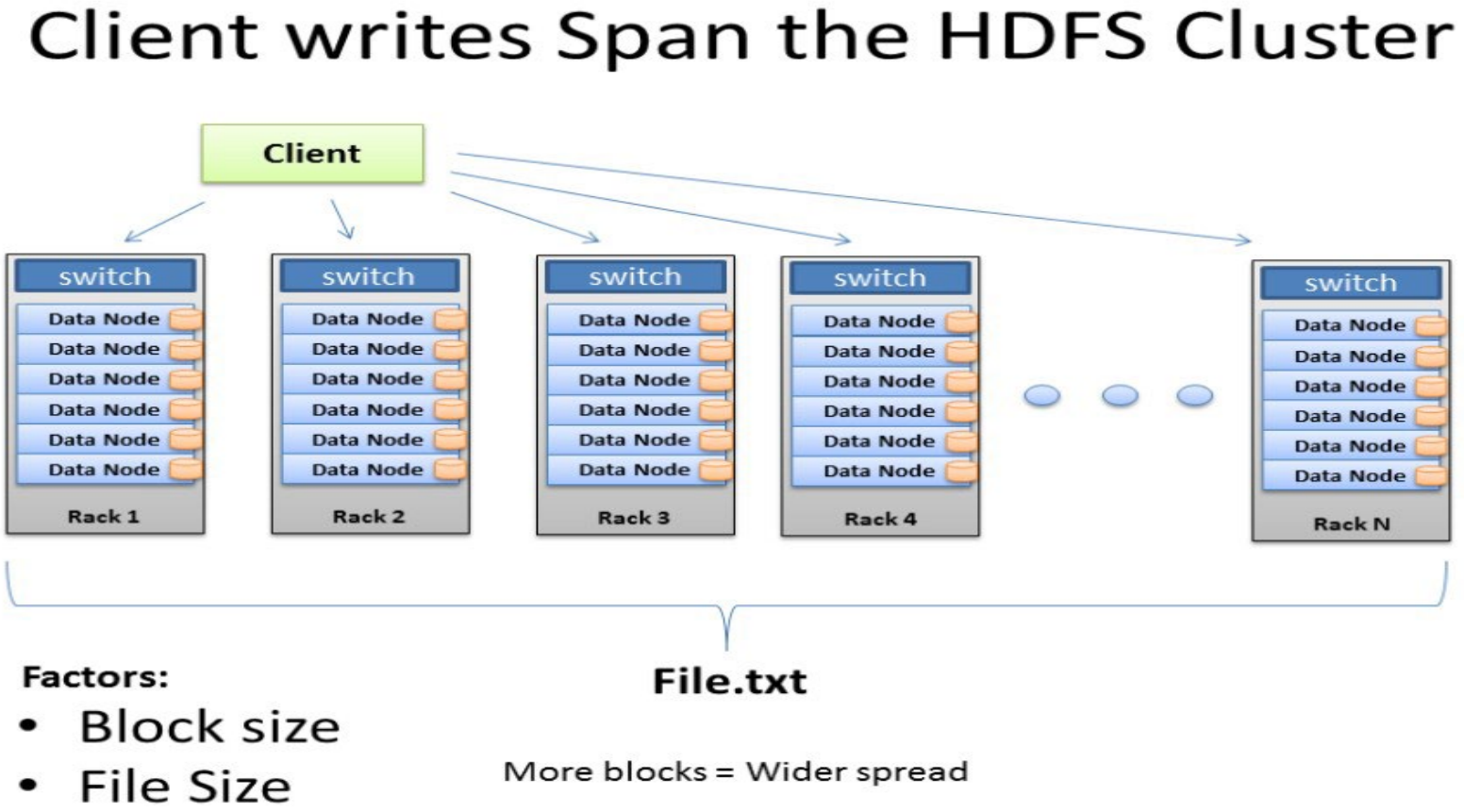

- Data Nodes 1 & 2 pass data along as its received
- TCP 50010

3 Bytes are pushed to the pipeline as a sequence of *packets*. The bytes that an application writes first buffer at the client side. After a packet buffer is filled (typically *64 KB*), the data are pushed to the pipeline.

# File I/O Operations and Replica Management

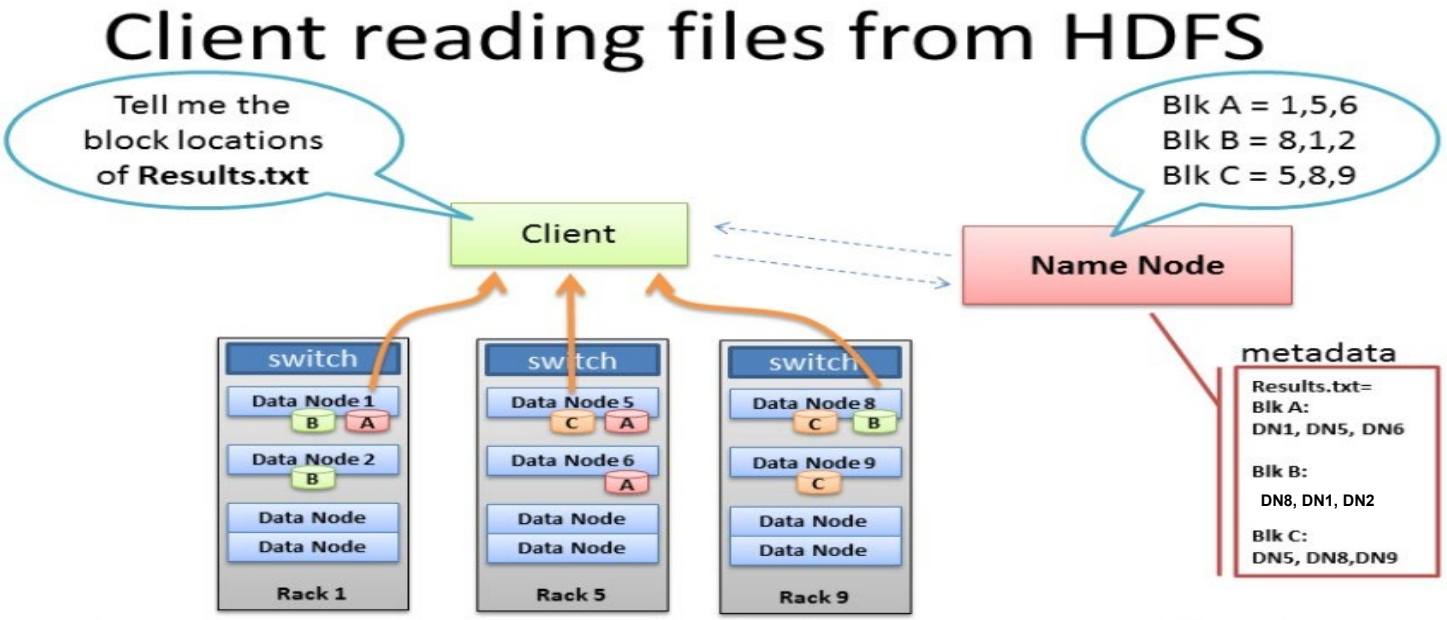


# File I/O Operations and Replica Management




The more blocks that make up a file, the more machines the data can potentially spread. The more CPU cores and disk drives that have a piece of my data mean more parallel processing power and faster results. This is the motivation behind building large, **wide** clusters. To process more data, faster.

# File I/O Operations and Replica Management

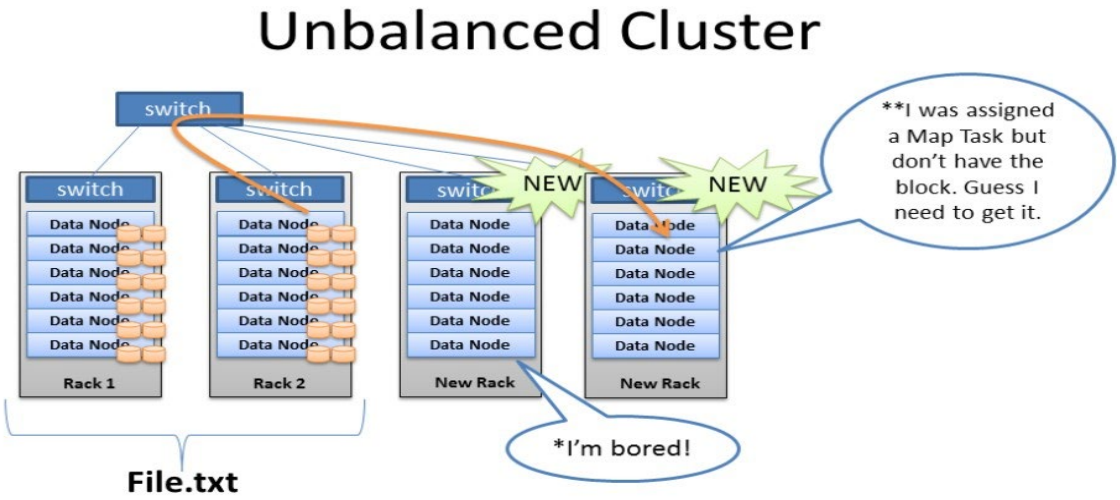



- Client receives Data Node list for each block
- Client picks first Data Node for each block
- Client reads blocks sequentially

In this graph, client just fetches the first of the block choices NameNode provided,(1,8,5) actually fetched in the order of the distance from the client.

# File I/O Operations and Replica Management

- Balancer



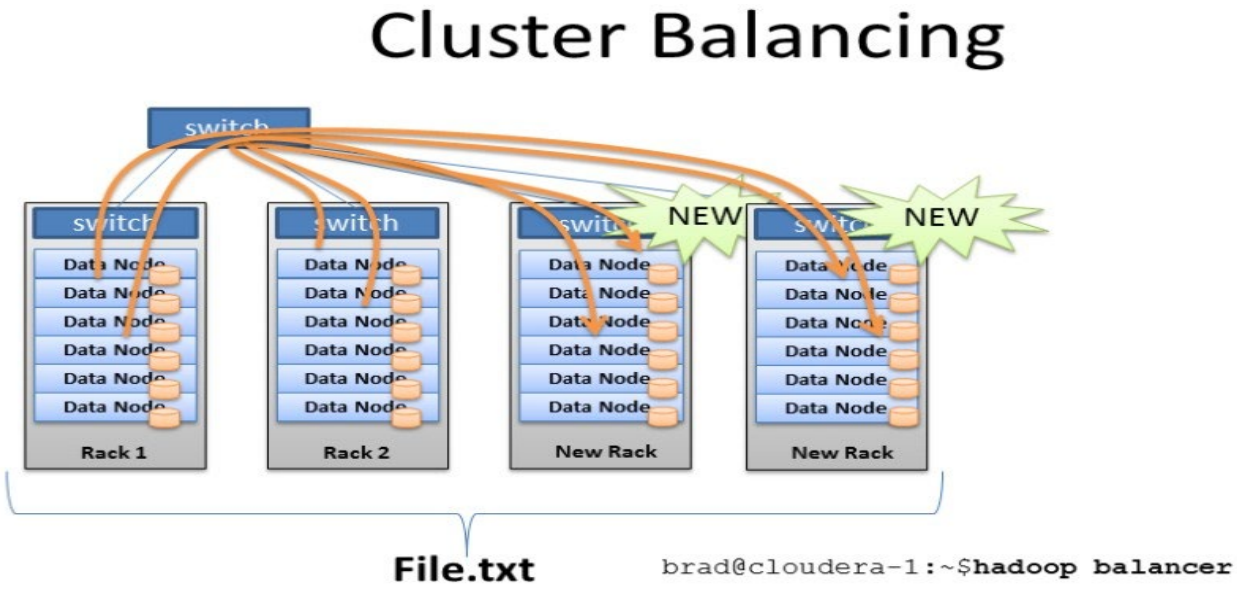

- Hadoop prefers local processing if possible
- New servers underutilized for Map Reduce, HDFS*
- More network bandwidth, slower job times**

- When we add new racks full of servers and network to an existing Hadoop cluster we can end up in a situation where our cluster is unbalanced. In this case, Racks 1 & 2 were my existing racks containing File.txt and running my Map Reduce jobs on that data. When I added two new racks to the cluster, my File.txt data doesn't auto-magically start spreading over to the new racks. All the data stays where it is.

1. The new servers are sitting idle with no data, until I start loading new data into the cluster.
2. Furthermore, if the servers in Racks 1 & 2 are really busy, the Job Tracker may have no other choice but to assign Map tasks on File.txt to the new servers which have no local data.
3. The new servers need to go grab the data over the network. As result you may see more network traffic and slower job completion times.

# File I/O Operations and Replica Management

- Balancer




- Balancer utility (if used) runs in the background
- Does not interfere with Map Reduce or HDFS
- Default rate limit 1 MB/s

- Balancer looks at the difference in available storage between nodes and attempts to provide balance to a certain threshold.
- New nodes with lots of free disk space will be detected and balancer can begin copying block data off nodes with less available space to the new nodes.

# Outline

- Introduction
- Architecture

  NameNode, DataNodes, HDFS Client, CheckpointNode, BackupNode, Snapshots
- File I/O Operations and Replica Management

  File Read and Write, Block Placement, Replication management, Balancer,
- **Practice at YAHoo!**
- FUTURE WORK

# Practice at YAHoo!

- **HDFS clusters at Yahoo! include about 3500 nodes**
- **A typical cluster node has:**

· 2 quad core Xeon processors @ 2.5ghz
· Red Hat Enterprise Linux Server Release 5.1
· Sun Java JDK 1.6.0_13-b03
· 4 directly attached SATA drives (one terabyte each)
· 16G RAM
· 1-gigabit Ethernet

Yahoo! Adopted Hadoop for internal use at the end of 2006. So the data is a little bit out of date.

# Practice at YAHoo!

- 70 percent of the disk space is allocated to HDFS. The remainder is reserved for the operating system (Red Hat Linux), logs, and space to spill the *output of map tasks*. **(MapReduce intermediate data are not stored in HDFS.)**
- For each cluster, the NameNode and the BackupNode hosts are specially provisioned with up to 64GB RAM; application tasks are never assigned to those hosts.
- In total, a cluster of 3500 nodes has 9.8 PB of storage available as blocks that are replicated three times yielding a net 3.3 PB of storage for user applications. As a convenient approximation, one thousand nodes represent one PB of application storage.

Why MapReduce intermediate data are not stored in HDFS?
1. HDFS read and write is expensive.
2. If intermediate data is lost, computation can be done on the corresponding dataNode again to get intermediate data. It is determinstic.

# Practice at YAHoo!

## Durability of Data

- **Uncorrelated node failures**

  Replication of data *three times* is a robust guard against loss of data due to uncorrelated node failures.

- **Correlated node failures, the failure of a rack or core switch.**

HDFS can tolerate losing a rack switch (each block has a replica on some other rack).

- **Loss of electrical power to the cluster**

a large cluster will lose a handful of blocks during a power-on restart.

# Practice at YAHoo!

- Benchmarks

| Bytes (TB) | Nodes | Maps | Reduces | Time | HDFS I/0 Bytes/s | |
|---|---|---|---|---|---|---|
| | | | | | Aggregate (GB) | Per Node (MB) |
| 1 | 1460 | 8000 | 2700 | 62 s | 32 | 22.1 |
| 1000 | 3658 | 80 000 | 20 000 | 58 500 s | 34.2 | 9.35 |

*Table 2. Sort benchmark for one terabyte and one petabyte of data. Each data record is 100 bytes with a 10-byte key. The test program is a general sorting procedure that is not specialized for the record size. In the terabyte sort, the block replication factor was set to one, a modest advantage for a short test. In the petabyte sort, the replication factor was set to two so that the test would confidently complete in case of a (not unexpected) node failure.*

- For row two, failure is no long uncommon,
- because, for example, if one machine dies in a thousand machines, when ten thousand machines, ten machines may die.
- This is a lot. So replica factor being two is necessary.

# Practice at YAHoo!

- Benchmarks

Ops/s is operations per second

| Operation | Throughput (ops/s) |
|---|---|
| Open file for read | 126 100 |
| Create file | 5600 |
| Rename file | 8300 |
| Delete file | 20 700 |
| DataNode Heartbeat | 300 000 |
| Blocks report (blocks/s) | 639 700 |

NameNode Throughput benchmark

# FUTURE WORK

- Automated failover

**plan:** Zookeeper, Yahoo's distributed consensus technology to build an automated failover solution

- Scalability of the NameNode

**Solution:** Our near-term solution to scalability is to allow multiple namespaces (and NameNodes) to share the physical storage within a cluster.

**Drawbacks:** The main drawback of multiple independent namespaces is the cost of managing them.

# 分布式计算

## HDFS使用及编程

Weixiong Rao 饶卫雄
Tongji University 同济大学软件学院
2023 秋季
wxrao@tongji.edu.cn

# 安装及启动

- Single Node Setup → Cluster Setup
  - 安装Java
  - 安装ssh
  - 安装Hadoop 单节点

http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/SingleCluster.html

- 启动
  - Local (Standalone) Mode 本地模式
  - Pseudo-Distributed Mode 伪分布式
  - Fully-Distributed Mode 真分布式

# Pseudo-Distributed Mode 伪分布式

- 在单机节点上运行伪分布式Hadoop
  - 每个Hadoop daemon运行在一个独立的Java进程
- 配置
  - etc/hadoop/core-site.xml
  - etc/hadoop/hdfs-site.xml
- 设置passphraseless ssh
- 启动
  - 格式化文件系统 $ bin/hdfs namenode -format
  - 启动NameNode和DataNode daemon: $ sbin/start-dfs.sh
  - Web 接面访问 http://localhost:9870/

# Hadoop示例代码

- HDFS文件目录准备

```
$ bin/hdfs dfs -mkdir /user
$ bin/hdfs dfs -mkdir /user/<username>
```

- 输入文件准备

```
$ bin/hdfs dfs -mkdir input
$ bin/hdfs dfs -put etc/hadoop/*.xml input
```

- 示例程序

```
$ bin/hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-3.0.0-beta1.jar grep input output 'dfs[a-z.]+'
```

- 输出文件

```
$ bin/hdfs dfs -get output output
$ cat output/*
```

```
$ bin/hdfs dfs -cat output/*
```

- 停止Hadoop daemon

```
$ sbin/stop-dfs.sh
```

# Spark的安装与部署

Spark参考文档
https://yq.aliyun.com/articles/120247

- Spark主要使用HDFS充当持久化层，完整地使用Spark需要预先安装Hadoop
- **在Linux集群上安装与配置Spark**
  1. 安装JDK
  2. 安装Scala
  3. 配置SSH免密码登录
  4. 安装Hadoop
  5. 安装Spark

# HDFS交互操作

- Web界面操作
  - http://namenode-name:9870
- 命令行操作
  - bin/hdfs dfs -help
  - bin/hdfs dfs -help command-name
- **DFSAdmin 命令**
  - bin/hdfs dfsadmin –help
  - -report: 报告HDFS的基本统计情况
  - -printTopology: 打印集群拓扑情况, 展示racks和datanodes以及关联NameNode的树形结构.

# HDFS的Balancer

- HDFS未必能够总是保证数据均匀的放到DataNodes之上, 如新的DataNode加入
- 总体来看DataNodes采用多个策略取得数据负载平衡
  - 写数据节点存入一份数据块副本
  - 需一份数据块副本放至同一个机架其他节点
  - 数据块副本需跨多个机架
  - 将HDFS数据块均匀放至集群节点

**Rack Awareness**

- 上述策略可能存在冲突, 导致负载不均匀

```
hdfs balancer
        [-policy <policy>]
        [-threshold <threshold>]
        [-exclude [-f <hosts-file> | <comma-separated list of hosts>]]
        [-include [-f <hosts-file> | <comma-separated list of hosts>]]
        [-source [-f <hosts-file> | <comma-separated list of hosts>]]
        [-blockpools <comma-separated list of blockpool ids>]
        [-idleiterations <idleiterations>]
        [-runDuringUpgrade]
```

# HDFS Erasure Coding纠删编码

- 数据副本导致额外的成本开销
  - 3*倍的存储开销
  - 热数据、冷数据，所需的资源均一致
- 更优的方法：纠删编码
  - 替换数据副本一样的容错功能，但空间开销更少: 仅50%
- 最常见的方法
  - Redundant Array of Inexpensive Disks (RAID)

```
hdfs ec [generic options]
    [-setPolicy -path <path> [-policy <policyName>] [-replicate]]
    [-getPolicy -path <path>]
    [-unsetPolicy -path <path>]
    [-listPolicies]
    [-addPolicies -policyFile <file>]
    [-listCodecs]
    [-enablePolicy -policy <policyName>]
    [-disablePolicy -policy <policyName>]
    [-help [cmd ...]]
```

# HDFS JAVA 读写操作API

- HDFS
  - 抽象了整个集群的存储资源，可以存放大文件。
  - 文件采用分块存储复制的设计，默认块大小是64M
  - 流式数据访问，一次写入和append操作，多次读取
- 不适合的方面
  - 低延迟的数据访问(解决方案：KVS)
  - 大量的小文件

# HDFS JAVA 读写操作API

- 通过URL访问文件

```
hdfs://localhost:9000/user/joe/TestFile.txt
URI uri=URI.create (“hdfs://host: port/path”);
```

- 配置文件conf/core-site.xml需包括上述host和port

```
<property><name>fs.default.name</name><value>hdfs://localhost:9000</value></property>
```

- 访问路径

```
Path path=new Path (uri); //It constitute URI
```

- 创建配置 载入core-site and core-default.xml系统配置信息

```
Configuration conf = new Configuration ();
```

- 文件系统

```
public static FileSystem get(Configuration conf)
public static FileSystem get(URI uri, Configuration conf)
public static FileSystem get(URI uri, Configuration conf, String user)
```

# HDFS JAVA 读写操作API

- **FSDataInputStream**

缺省的buffer size 4096 byte i.e. 4KB

```
URI uri = URI.create ( “hdfs://host: port/file path” );
Configuration conf = new Configuration ();
FileSystem file = FileSystem.get (uri, conf);
FSDataInputStream in = file.open(new Path(uri));
```

```
public interface Seekable {
  void seek(long pos) throws IOException;
  long getPos() throws IOException;
 boolean seekToNewSource(long targetPos) throws IOException;
}
```

```
FileSystem file = FileSystem.get (uri, conf);
FSDataInputStream in = file.open(new Path(uri));
byte[] btbuffer = new byte[5];
in.seek(5); // sent to 5th position
Assert.assertEquals(5, in.getPos());
in.read(btbuffer, 0, 5);//read 5 byte in byte array from offset 0
System.out.println(new String(btbuffer));//print 5 character from 5th position
in.read(10,btbuffer, 0, 5);// print 5 character staring from 10th position
```

# HDFS JAVA 读写操作API

- **FSDataOutputStream**

public FSDataOutputStream create(Path f) // create empty file.
public FSDataOutputStream append(Path f) // will append existing file

```
URI uri=URI.create(strURI);
FileSystem fileSystem=FileSystem.get(uri,conf);
FileStatus fileStatus=fileSystem.getFileStatus(new Path(uri));
System.out.println("AccessTime:"+fileStatus.getAccessTime());
System.out.println("AccessTime:"+fileStatus.getLen());
System.out.println("AccessTime:"+fileStatus.getModificationTime());
System.out.println("AccessTime:"+fileStatus.getPath());
```

# HDFS JAVA 读写操作API

- **FileStatus**

```
URI uri=URI.create(strURI);
        FileSystem fileSystem=FileSystem.get(uri,conf);
        FileStatus fileStatus=fileSystem.getFileStatus(new Path(uri));
        System.out.println("AccessTime:"+fileStatus.getAccessTime());
        System.out.println("AccessTime:"+fileStatus.getLen());
        System.out.println("AccessTime:"+fileStatus.getModificationTime());
        System.out.println("AccessTime:"+fileStatus.getPath());
```

```
public FileStatus[] listStatus(Path f)
```

# 示例代码

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.BlockLocation;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class hdfsexample {
        …
}
```

# 示例代码

```
public static void main(String[] args) throws IOException {
    Configuration conf = new Configuration();
    //this is import for connect to hadoop hdfs
    //or else you will get file:///, local file system
    conf.set("fs.default.name", "hdfs://namenode:9000");

    FileSystem fs = FileSystem.get(conf);
    System.out.println(fs.getUri());
    Path file = new Path("/user/hadoop/test/demo2.txt");
    if (fs.exists(file)) fs.delete(file,false);
    write(fs,file);
    read(fs,file);
    showblock(fs,file);

    fs.close();
  }
```

# 示例代码

```
static void showblock(FileSystem fs,Path file) throws IOException
{
   // show the file meta data info
   FileStatus      fileStatus = fs.getFileStatus(file);
   BlockLocation[] blocks  = fs.getFileBlockLocations(fileStatus, 0, fileStatus.getLen());
   for(BlockLocation bl:blocks)
       System.out.println(bl.toString());
}
```

# 示例代码

```
static void read(FileSystem fs,Path file) throws IOException
{
   //Reading from file
   FSDataInputStream inStream = fs.open(file);
   String data = null;
   BufferedReader br = new BufferedReader(new InputStreamReader(inStream));
   while((data = br.readLine())!=null)
       System.out.println(data);
   br.close();
}
```

# 示例代码

```
static void write(FileSystem fs,Path file) throws IOException
   {
      FSDataOutputStream outStream = null;
      outStream = fs.create(file);
      BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(outStream));
      for(int i=1;i<101;i++)
         {
            bw.write("Line" +i + " welcome to hdfs java api");
            bw.newLine();
         }
      bw.close();
   }

}
```