Advanced Cloud Computing

Hadoop Distributed File System

Wei Wang
CSE@HKUST
Spring 2022

THE DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING
計算機科學及工程學系

Outline

HDFS overview

Architecture

Workflow

Fault tolerance

Programming APIs

What is **H**adoop **D**istributed **F**ile **S**ystem (HDFS)?

What is HDFS?

A short answer

- An *open-source* implementation of GFS

A long answer

- A **filesystem** designed for storing **very large files** with **streaming data access patterns**, running on clusters of **commodity hardware**

Brief history

Initially developed by Doug Cutting as a filesystem for Apache Nutch, a web search engine

- early name: **N**utch **D**istributed **F**ile**S**ystem (NDFS)

Moved out of Nutch and acquired by Yahoo! in 2006 as an independent project called *Hadoop*

The origin of the name

“**Hadoop**” is a made-up name, as explained by Doug Cutting:

> “The name my kid gave a stuffed yellow elephant. Short, relatively easy to spell and pronounce, meaningless, and not used elsewhere: those are my naming criteria. Kids are good at generating such. Googol is a kid’s term.”

Command-Line Interface

Basic filesystem operations

Start with **`hadoop fs`** (or **`hdfs dfs`**), followed by similar commands to Linux OS

- `% hadoop fs -ls`
- `% hadoop fs -rm`
- `% hadoop fs -mv`
- `% hadoop fs -put` **`localFile hdfsFile`**
- `% hadoop fs -get` **`hdfsFile localFile`**
- `% hadoop fs -help`

Basic filesystem operations

Also supports operations on the local filesystem (with prefix `file:///`)

- `% hadoop fs -ls file:///`**`localDir`**
- `% hadoop fs -mkdir file:///`**`localDir`**

HDFS design assumptions

Cheap, commodity machines

- Failures as a norm, rather than an exception in large clusters (e.g., 10k nodes)
- hard disk, power supply, human errors, etc.

A “modest” number of very *large* files

- a few million files each > 100MB

HDFS design assumptions

Batch processing

- Files are **write-once**, **mostly appended to** (perhaps concurrently)
- **Streaming reads**, rather than random data access
- High **sustained throughput** favored over low latency

Do these look familiar?

The design of HDFS heavily borrows from GFS

Design of HDFS

NameNode

- a single master for managing filesystem meta

DataNode (chunkserver)

- multiple DataNodes for storing and retrieving data
- reports to NameNode with list of blocks hosted

SecondaryNameNode (shadow master)

- performs checkpointing

Design of HDFS

Single namespace for the entire cluster

Data coherency: Write-once, read-many-times

Files are broken up into blocks: 128MB blocks each replicated on multiple DataNodes

Intelligent client

- Client can find locations of blocks
- Client accesses data directly from DataNodes

Demo: HDFS web interface

Noticeable differences from GFS

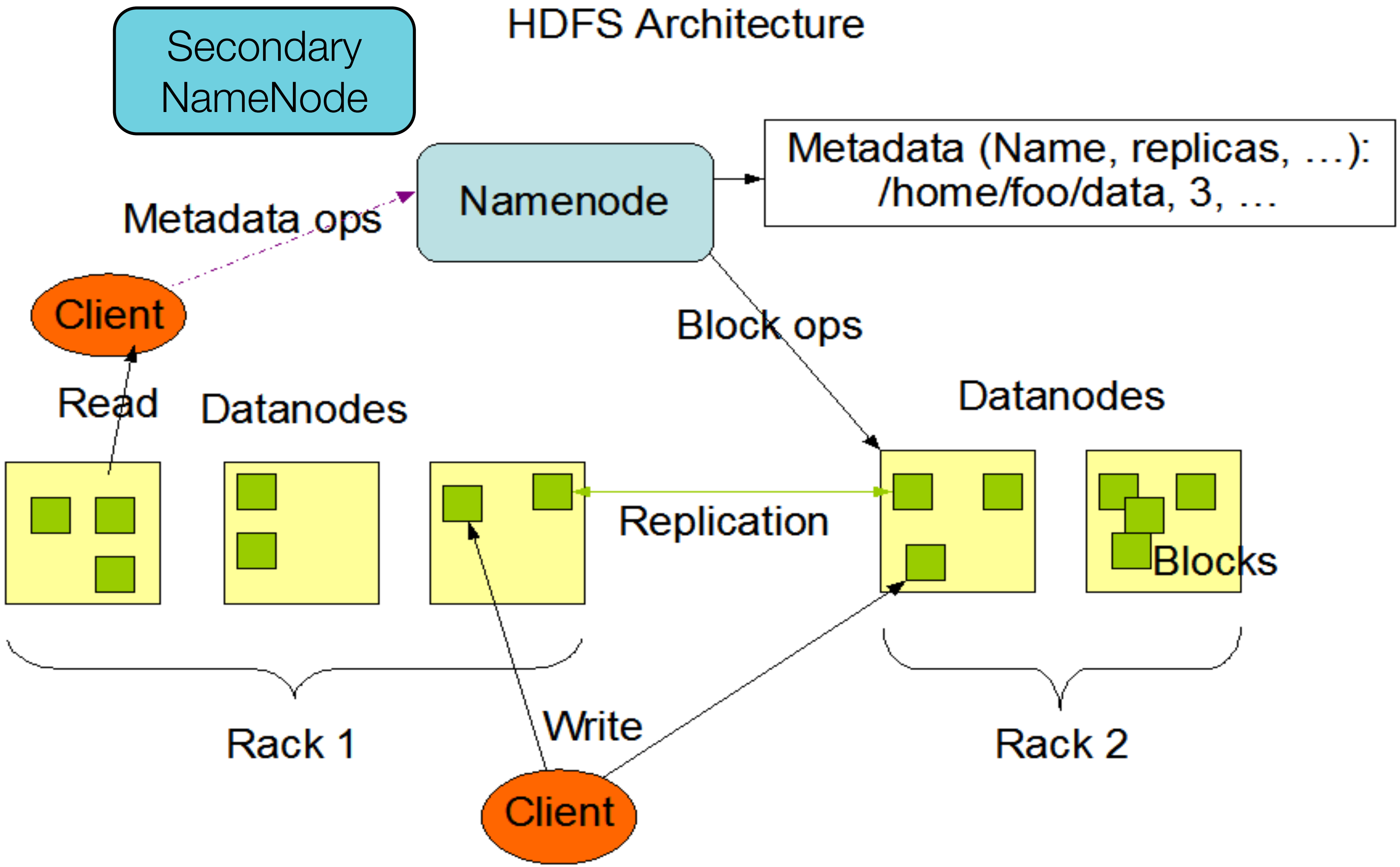
Earlier versions have only a single writer per file

- No *record append* operation supported in earlier versions

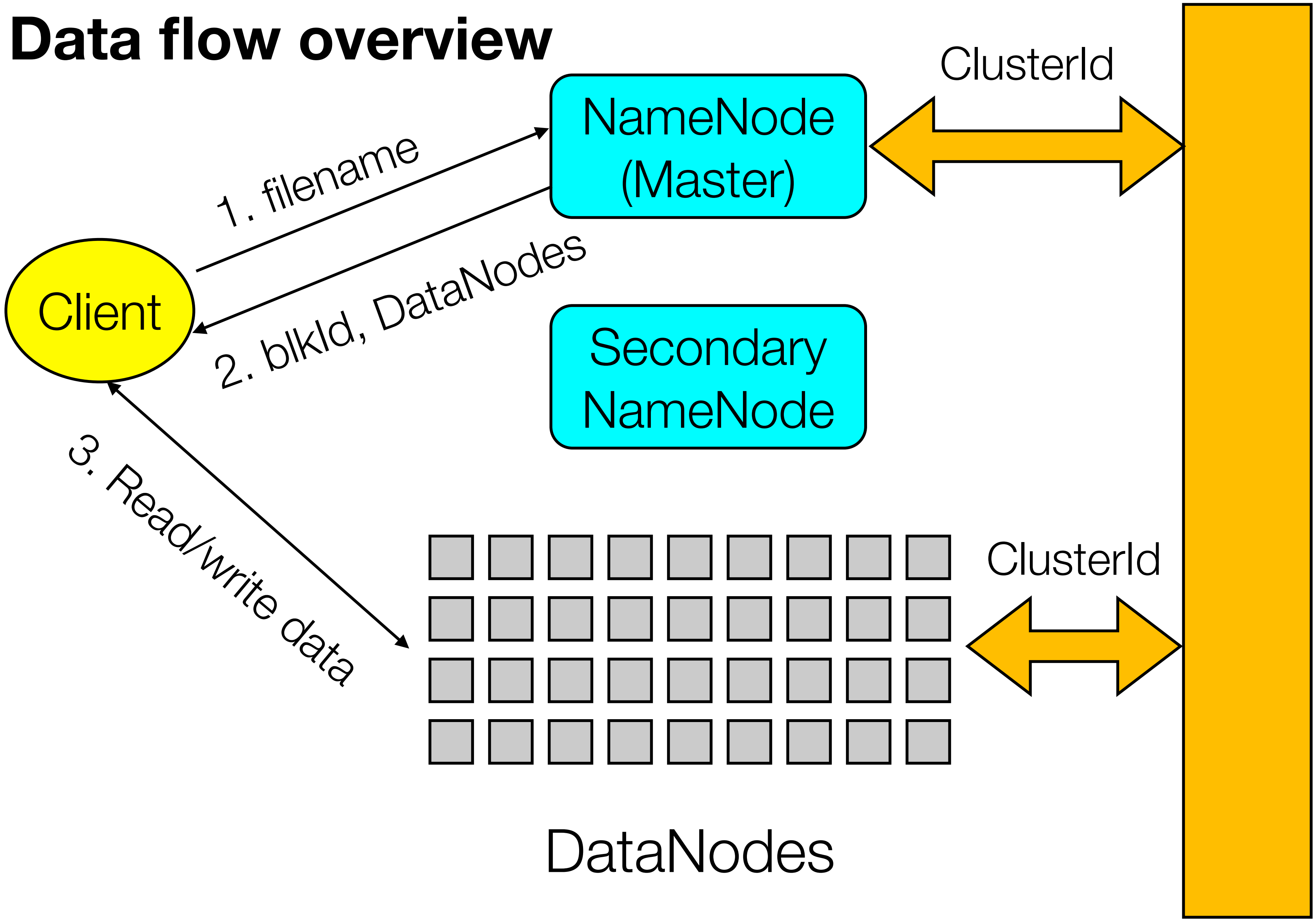
Open source

- Provides many interfaces and libraries for different filesystems
  - S3, KFS, etc.
  - Thrift (C++, Python, …), libhdfs (C)

Architecture


HDFS Architecture
Secondary NameNode
Namenode
Metadata (Name, replicas, …):
/home/foo/data, 3, …
Metadata ops
Client
Block ops
Read
Datanodes
Datanodes
Replication
Blocks
Rack 1
Write
Rack 2
Client

Data flow overview



NameNode (master)

Functions of a NameNode

Manages filesystem namespace

- Maps a filename to a set of blocks
- Maps a block to the DataNodes where it resides

Cluster configuration management

Replication engine for blocks

NameNode metadata

Metadata kept in memory

Types of metadata

Transaction log

- List of files
- List of blocks for each file
- File attributes, e.g., creation time, replication factor
- List of DataNode for each block

A transaction log: records file creations, file deletions, etc.

DataNode

DataNode

Stores data in the local file system

Stores metadata of a block (e.g., CRC checksum)

Serves data and metadata to clients

Communicates with NameNode through periodic "heartbeat" (once per 3 secs)

DataNode

Block report

- Periodically (1-hour by default) sends a report of all existing blocks to the NameNode

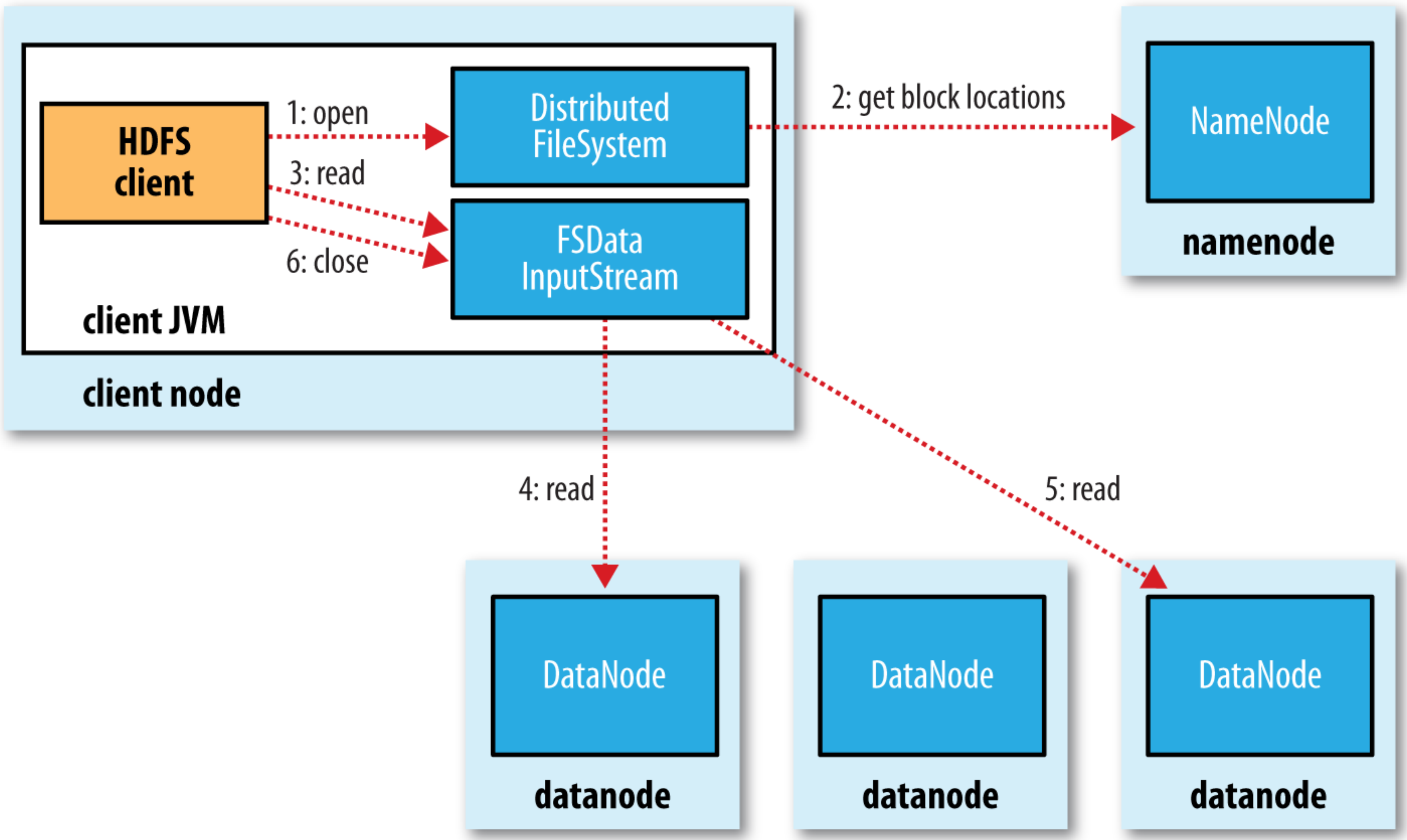
Facilitates pipelining of data

- Forwards data to other specified DataNodes

HDFS Workflow

File read and write

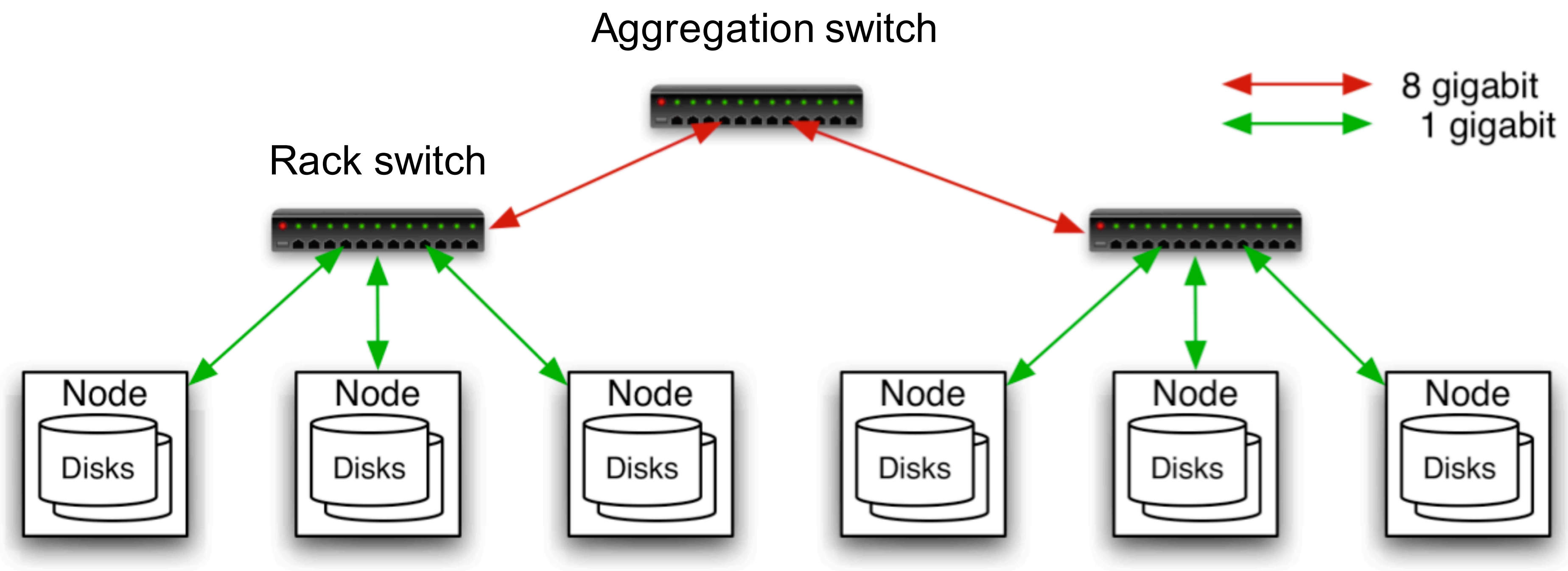
Anatomy of a file read



Source: T. White, “Hadoop: The Definitive Guide,” O’REILLY, 4th Eds., 2015.

How to choose the "closest" block?

Choosing the “closest” block



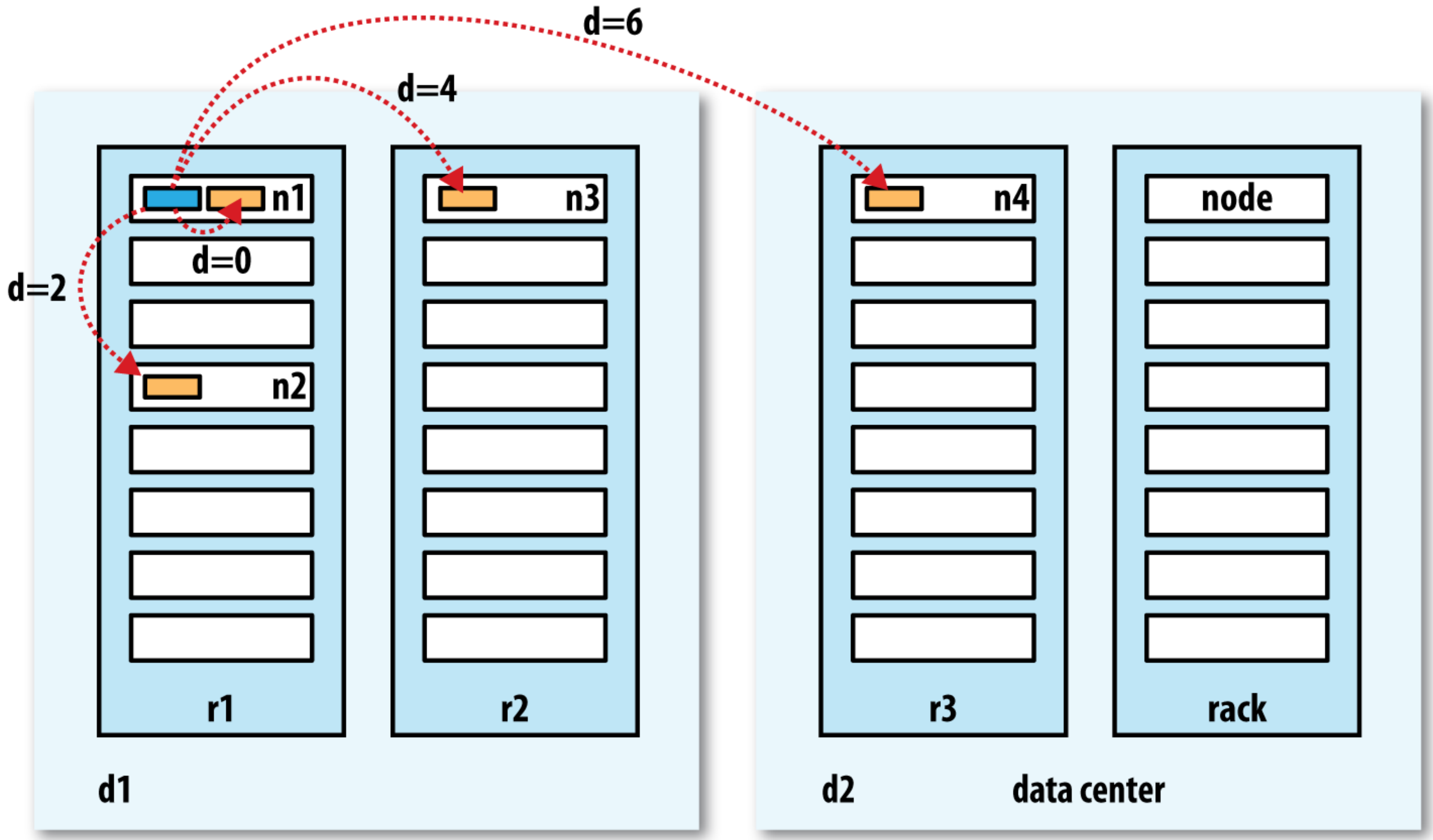
Choosing the “closest” block

Computing the ***distance*** between two nodes

Denote a node *n1* on rack *r1* in DC *d1* by */d1/r1/n1*

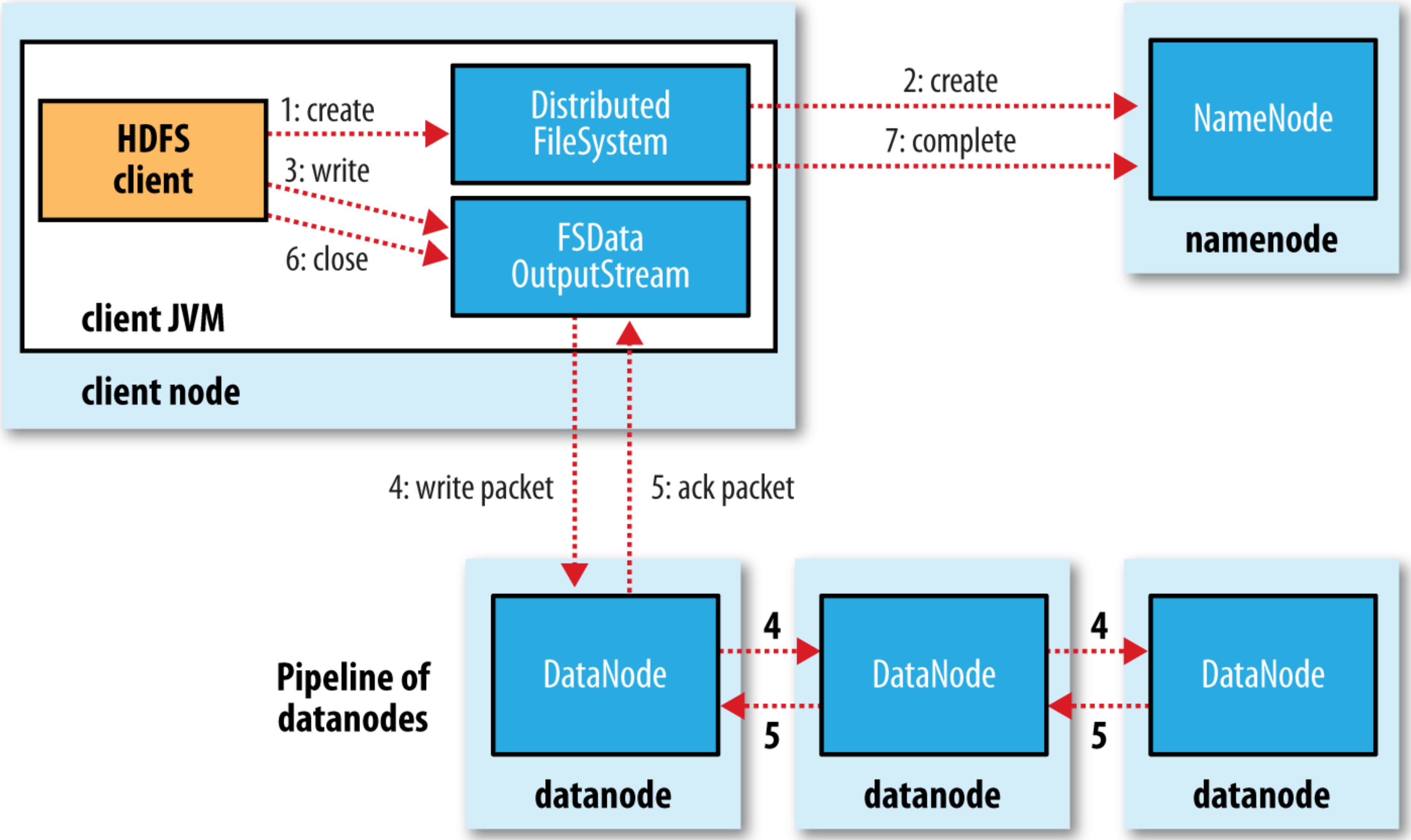
- dist(*/d1/r1/n1, /d1/r1/n1*) = 0 (process on the same node)
- dist(/d1/r1/n1, /d1/r1/n2) = 2 (different nodes on the same rack)
- dist(*/d1/r1/n1, /d1/r2/n3*) = 4 (nodes on different racks in the same datacenter)
- dist(*/d1/r1/n1, /d2/r3/n4*) = 6 (nodes in different datacenters)

Distance between two nodes



Source: T. White, “Hadoop: The Definitive Guide,” O’REILLY, 4th Eds., 2015.

Anatomy of a file write



Source: T. White, “Hadoop: The Definitive Guide,” O’REILLY, 4th Eds., 2015.

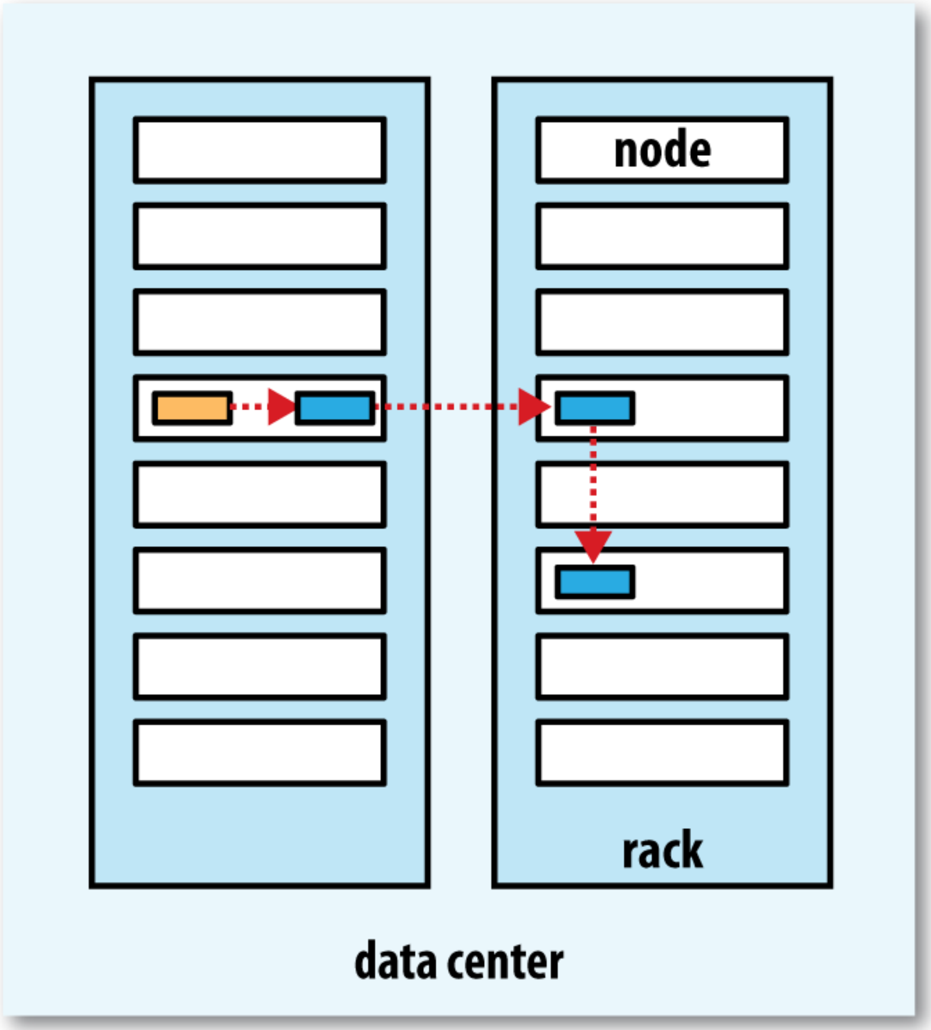
Block placement

Current strategy (replaceable with customized policy)

- One replica on local node
- 2nd and 3rd replica on two nodes of same *remote* rack
- Additional replicas are randomly placed

Once the replica locations have been chosen, a **pipeline** is built

Replica pipeline



HDFS Client

Block Replica

Source: T. White, “Hadoop: The Definitive Guide,” O’REILLY, 4th Eds., 2015.

Handle failures

Heartbeats

DataNodes send heartbeats to the NameNode

- Once every 3 secs

NameNode uses heartbeats to detect DataNode failure

- No response in 10 mins is considered a failure

Replication engine

Upon detecting a DataNode failure

- Choose new DataNodes for replicas
- Balance disk usage
- Balance communication traffic to DataNodes

Data corrections

Checksums to validate data (CRC32)

File creation

- Client computes checksum per 512 byte
- DataNode stores the checksum

File access

- Client retrieves data and checksum from DataNodes
- If validation fails, try other replicas

NameNode failure

A single point of failure

Transaction log stored in multiple directories

- Directory on local file system
- A directory on a remote file system (NFS)

Add a **secondary NameNode**

Secondary NameNode

Not Standby/Backup NameNode

- only for checkpointing
- has a NON-Realtime copy of FSImage

Copies NameNode's FSImage & Transaction Log

Merges them to a new FSImage

Uploads new FSImage to the NameNode and purges Transaction Log

Summary

As an open-source implementation of GFS, HDFS shares the same design assumptions

- Very large files
- Streaming data access pattern
- Commodity hardware

Limitations

Low-latency data access

- tens of millisecond range
- HDFS emphasizes throughput over latency

Lots of small files

- billions of files
- All meta data are kept in memory, resulting in overflow

Multiple writers, arbitrary file modifications

HDFS FileSystem API (Java): `org.apache.hadoop.fs`

Reading data

A general filesystem API is provided by **FileSystem**

Retrieve an instance using static factory methods:

```
public static FileSystem get(URI uri, Configuration conf) throws IOException
```

used to infer the filesystem scheme (e.g., **hdfs://** for HDFS, **file:///** for local filesystem)

encapsulates a client's config, usually set in **etc/hadoop/core-site.xml**

Reading data

A file in HDFS is represented by a Hadoop `Path` object

- HDFS URI: hdfs://localhost/user/weiwa/hkust.txt

Get the input stream of a file using `open()` method

```
public FSDataInputStream open(Path f) throws IOException
public abstract FSDataInputStream open(Path f, int bufferSize)
  throws IOException
```

Reading data

Putting them together: `cat` a file

```
public class FileSystemCat {

  public static void main(String[] args) throws Exception {
    String uri = args[0];
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(uri), conf);
    InputStream in = null;
    try {
      in = fs.open(new Path(uri));
      IOUtils.copyBytes(in, System.out, 4096, false);
    } finally {
      IOUtils.closeStream(in);
    }
  }
}
```

A handy I/O tool

Writing data

Use the **create()** method

```
public FSDataOutputStream create(Path f) throws IOException
```

```
String localSrc = args[0];
String dst = args[1];

Configuration conf = new Configuration();
LocalFileSystem localFS = LocalFileSystem.get(conf);
FSDataInputStream in = localFS.open(new Path(localSrc));

FileSystem outFS = FileSystem.get(URI.create(dst), conf);

FSDataOutputStream out = outFS.create(new Path(dst));

IOUtils.copyBytes(in, out, 4096, true);
```

Deleting data

Use the **delete()** method on FileSystem to permanently remove files or directories:

```
public boolean delete(Path f, boolean recursive) throws IOException
```

- **recursive** is ignored if **f** is a file or an empty directory
- returns true if delete is successful

```
String uri = args[0];
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(URI.create(uri), conf);
fs.delete(new Path(uri), true);
```

Credits

Some slides are adapted from Dhruba Borthakur's slides