

CSx55: DISTRIBUTED SYSTEMS [HDFS]

HDFS: When to federate and replicate

A namenode often becomes

The pinch of the hourglass

To alleviate federate

To cope with failures

And other erratic behaviors

Have a hot standby replicate

Shrideep Pallickara

Computer Science

Colorado State University



Frequently asked questions from the previous class survey

- What is used more often strobos or pings?
- Does failure of namenode in HDFS Federation result in loss of data?



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA
COMPUTER SCIENCE DEPARTMENT

HDFS

L25.2

Topics covered in today's lecture

- HDFS
 - Federation (wrap-up)
 - High availability
 - Reading/writing data



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA
COMPUTER SCIENCE DEPARTMENT

HDFS

L25.3

HDFS FEDERATION (WRAP-UP) & HIGH AVAILABILITY



"I am pleased to see that we have differences. May we together become greater than the sum of both of us."

—Surak, Vulcan Philosopher

- Admin starts a new primary namenode
 - With one of the filesystem metadata replicas
 - Configure datanodes and clients to use this namenode
- The new namenode is unable to serve requests until:
 - ① Namespace image is **loaded** into memory
 - ② **Replay** of edit log is complete
 - ③ Received enough **block reports** from datanodes to leave safe mode



Recovering from a failed namenode

[2/2]

- Recovery can be really long
 - On large clusters with many files and blocks this can be about 30 minutes
- This also impacts routine maintenance



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA
COMPUTER SCIENCE DEPARTMENT

HDFS

L25.6

HDFS High Availability has features to cope with this

- Pair of namenodes in **active standby** configuration
- During failure of active namenode, standby takes over the servicing of client requests
 - In 10s of seconds



HDFS High-Availability: Additional items to get things to work

- Namenodes use a highly-available **shared storage** to store the **edit log**
- Datanodes must send block reports to **both** namenodes
 - Block mappings stored in memory not disk
- Clients must be configured to handle namenode failover



HDFS HA: Dealing with ungraceful failovers

- Slow network or a network partition can trigger failover transition
 - Previously active namenode thinks it is ***still*** the active namenode
- The HDFS HA tries to avoid this situation using **fencing**
 - Previously active namenode should be prevented from causing corruptions



Fencing mechanisms: To shutdown previously active namenode

- Kill the namenode's process
- Revoking access to the shared storage directory
- Disabling namenode's network port
 - Using the remote management command
- STONITH
 - Use specialized power distribution unit to forcibly power down the host machine



Basic Filesystem Operations

- Type `hadoop fs –help` to get detailed help on commands
 - We are invoking Hadoop's filesystem shell command `fs` which supports other subcommands
- Start copying a file from the local filesystem to HDFS

```
% hadoop fs –copyFromLocal input/docs/quangle.txt  
/user/tom/quangle.txt
```



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA
COMPUTER SCIENCE DEPARTMENT

HDFS

L25.11

Basic Filesystem Operations

- Copy file back to the local filesystem

```
%hadoop fs –copyToLocal /user/tom/quangle.txt  
input/docs/quangle.copy.txt
```

- Verify if the movement of the files have changed the files in any way

```
% openssl md5 quangle.txt quangle.copy.txt
```



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA
COMPUTER SCIENCE DEPARTMENT

HDFS

L25.12

Basic Filesystem Operations

```
% hadoop fs -mkdir books
```

```
% hadoop fs -ls .
```

Found 2 items

```
drwxr-xr-x - tom supergroup 0 2019-04-02 22:41 /user/tom/books
```

```
-rw-r--r-- 1 tom supergroup 118 2019-04-02 22:29 /user/tom/quangle.txt
```

- Directories are treated as metadata and **stored by the namenode** not the datanodes

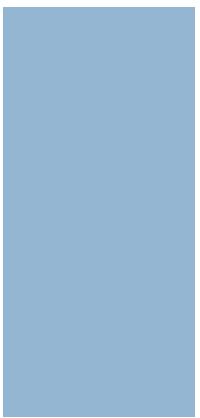


COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA
COMPUTER SCIENCE DEPARTMENT

HDFS

L25.13



HADOOP FILE SYSTEMS



Hadoop filesystems

- Hadoop has an abstract notion of filesystem
- HDFS is just one implementation
 - Others include HAR, KFS (Cloud Store), S3 (native and block-based)
- Uses URI scheme to pick correct filesystem instance to communicate with
`% hadoop fs –ls file://` to communicate with local file system



Interacting with the filesystem

- Hadoop has a `FileSystem` class
- HDFS implementation is accessible through the `DistributedFileSystem`
 - Write your code against the `FileSystem` class for maximum portability



URI and URLs

- A URI (Uniform Resource Identifier) is a string that uniquely identifies a **resource**
 - Anything that can be named or addressed ... on the internet or within a system.
- A URL (Uniform Resource Locator) is a specific type of URI that **not only identifies** a resource
 - But also provides a way to locate it by describing how to access it (e.g., protocol, domain, path).
- In summary:
 - URI = identifier
 - URL = identifier + address

hdfs:// for HDFS

s3a:// for Amazon S3

abfs:// for Azure Blob Storage



Reading data from a Hadoop URL

```
InputStream in = null;
try {
    in = new URL("hdfs://host/path").openStream();
    // process in
} finally {
    IOUtils.closeStream(in);
}
```



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA
COMPUTER SCIENCE DEPARTMENT

HDFS

L25.18

Make Java recognize Hadoop's URL scheme

- Call `setURLStreamHandlerFactory()` on `URL` with an instance of `FsURLStreamHandlerFactory`
- Can only be called once per JVM, so it is typically executed in a static block



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA
COMPUTER SCIENCE DEPARTMENT

HDFS

L25.19

Displaying files from a Hadoop filesystem

```
public class URLCat {  
    static {  
        URL.setURLStreamHandlerFactory(  
            new FsUrlStreamHandlerFactory());  
    }  
  
    public static void main(String[] args) throws Exception {  
        InputStream in = null;  
        try {  
            in = new URL(args[0]).openStream();  
            IOUtils.copyBytes(in, System.out, 4096, false);  
        } finally {  
            IOUtils.closeStream(in);  
        }  
    }  
}
```

Buffer size used
for copying

Close streams after
copying is complete?



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA
COMPUTER SCIENCE DEPARTMENT

HDFS

L25.20

A sample run of the URLCat

```
% hadoop URLCat hdfs://localhost/user/tom/quangle.txt
```

On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA
COMPUTER SCIENCE DEPARTMENT

HDFS

L25.21

Using the FileSystem API

- A file on the Hadoop filesystem is represented by a Hadoop Path object
 - Not the `java.io.File` object
- Path has a Hadoop filesystem URI



Retrieving an instance of the FileSystem

- ```
public static FileSystem
 get(Configuration conf) throws IOException
```

  - ▣ **Configuration encapsulates client or server's configuration** `conf/core-site.xml`
- ```
public static FileSystem
    get(URI uri, Configuration conf)
    throws IOException
```

 - ▣ **URI scheme identifies the filesystem to use**
- ```
public static FileSystem
 get(URI uri, Configuration conf,
 String user) throws IOException
```



# With a FileSystem instance in hand: Retrieving the input stream for a file

- ```
public FSDataInputStream
      open(Path f) throws IOException
```
- ```
public FSDataInputStream
 open(Path f, int bufferSize)
 throws IOException
```
- FSDataInputStream **is a specialization of the** `java.io.DataInputStream`
  - **Also implements the** `Seekable` **interface**



# Displaying files using the FileSystem directly

```
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);
 }
 }
}
```



# The execution of the program

```
% hadoop FileSystemCat hdfs://localhost/user/tom/quangle.txt
```

On the top of the Crumpetty Tree  
The Quangle Wangle sat,  
But his face you could not see,  
On account of his Beaver Hat.



**COLORADO STATE UNIVERSITY**

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

**HDFS**

**L25.26**

# Writing Data

- **Creation of a file**

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

- **Other versions of this method allow specification of**

- Overwriting existing files
  - Replication factor for the file
  - Buffer size to use
  - Block size



# Alternatively, you can append to an existing file

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

- Allows a **single writer** to modify an already written file
  - Open it and write data starting at the final offset



# FSDataOutputStream

- Unlike FSDataInputStream, this output stream **does not permit seeking**
- Only sequential writes or appends to a file are allowed



# Copying a local file to a Hadoop filesystem

```
public class FileCopyWithProgress {
 public static void main(String[] args) throws Exception {
 String localSrc = args[0];
 String dst = args[1];
 InputStream in =
 new BufferedInputStream(new FileInputStream(localSrc));

 Configuration conf = new Configuration();
 FileSystem fs = FileSystem.get(URI.create(dst), conf);
 OutputStream out = fs.create(new Path(dst),
 new Progressable() {
 public void progress() {
 System.out.print(".");
 }
 });
 IOUtils.copyBytes(in, out, 4096, true);
 }
}
```



# Directories

- ❑ FileSystem **supports creation of directories**

```
public boolean mkdirs(Path f)
throws IOException
```

- ❑ **Creates all necessary parent directories**
- ❑ **Writing a file by calling create() , automatically creates directories**



# FileStatus

- Encapsulates file system metadata for files and directories
- Includes:
  - File length
  - Block size
  - Replication
  - Modification time
  - Ownership and permission information



# But we often need to list status of multiple files ...

- ```
public FileStatus[] listStatus(Path f)
    throws IOException
```
- ```
public FileStatus[]
 listStatus(Path f, PathFilter filter)
 throws IOException
```
- ```
public FileStatus[] listStatus(Path[] files)
    throws IOException
```
- ```
public FileStatus[]
 listStatus(Path[] files, PathFilter filter)
 throws IOException
```



# File patterns

- Rather than enumerating each file and directory it is convenient to use **wildcards**
  - Match multiple files with a single expression
    - **Globbing**
- **FileSystem methods for processing globs**
  - `public FileStatus[] globStatus(Path pathPattern)  
throws IOException`
  - `public FileStatus[]  
globStatus(Path pathPattern,  
PathFilter filter)  
throws IOException`



# Hadoop provides the same glob support as UNIX

| Glob   | Name                    | Matches                                                                                                                   |
|--------|-------------------------|---------------------------------------------------------------------------------------------------------------------------|
| *      | asterisk                | Matches zero or more characters                                                                                           |
| ?      | question mark           | Matches a single character                                                                                                |
| [ab]   | character class         | Matches a single character in the set {a, b}                                                                              |
| [^ab]  | negated character class | Matches a single character that is not in the set {a, b}                                                                  |
| [a-b]  | character range         | Matches a single character in the (closed) range [a, b], where a is lexicographically less than or equal to b             |
| [^a-b] | negated character range | Matches a single character that is not in the (closed) range [a, b], where a is lexicographically less than or equal to b |
| {a,b}  | alternation             | Matches either expression a or b                                                                                          |
| \c     | Escaped character       | Matches character c when it is a metacharacter                                                                            |



# Looking at an example

[1 / 2]

- /2007/12/30
- /2007/12/31
- /2008/01/01
- /2008/01/02



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

L25.36

# Looking at an example

[2/2]

|                    |                         |
|--------------------|-------------------------|
| □ /*               | /2007 /2008             |
| □ /*/*             | /2007/12 /2008/01       |
| □ /*/12/*          | /2007/12/30 /2007/12/31 |
| □ /200?            | /2007 /2008             |
| □ /200[78]         | /2007 /2008             |
| □ /200[7-8]        | /2007 /2008             |
| □ /200[^01234569]  | /2007 /2008             |
| □ /*/*/{31,01}     | /2007/12/31 /2008/01/01 |
| □ /*/*/3{0,1}      | /2007/12/30 /2007/12/31 |
| □ /*/{12/31,01/01} | /2007/12/31 /2008/01/01 |



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA  
COMPUTER SCIENCE DEPARTMENT

HDFS

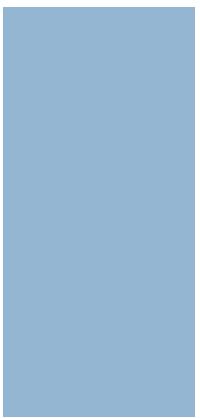
L25.37

# Deleting data

- Use the `delete()` method on `FileSystem`
- ```
public boolean
      delete(Path f, boolean recursive)
throws IOException
```

 - If `f` is a file or an empty directory then `recursive` is ignored.
 - Recursive deletion of directories happens only if `recursive` is true

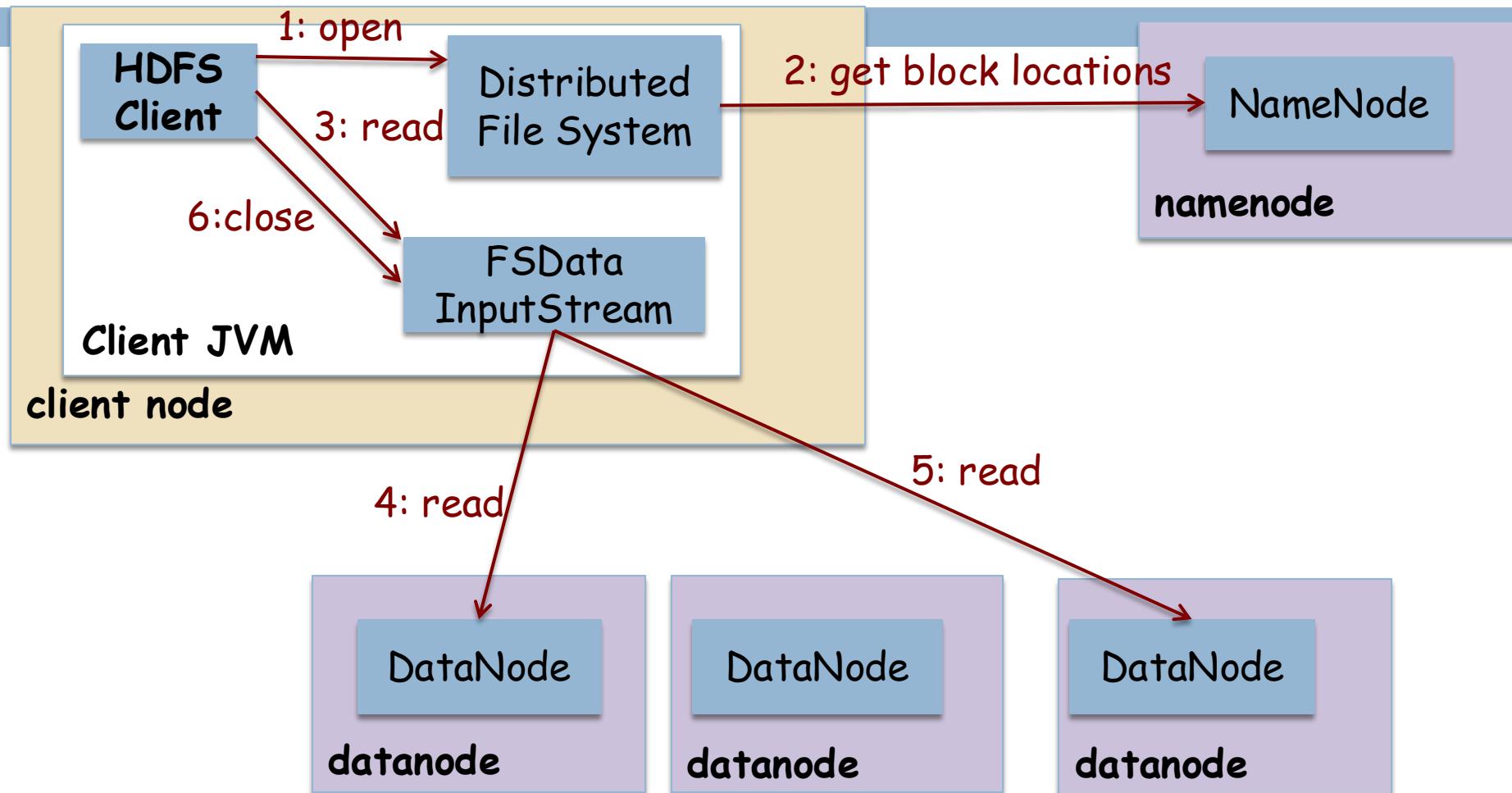




DATA FLOW IN HDFS



Data flow in HDFS [read]



Reading data

- `FSDataInputStream` **wraps a** `DFSInputStream`
 - `DFSInputStream` **manages I/O with the datanode and namenode**
- `DFSInputStream` **stores datanode addresses for the *first few blocks***
 - `Namenode` returns addresses of datanodes that have a copy of that block
 - Datanodes are **sorted** according to their *proximity to the client*



Reading data

- Blocks are read in order
- DFSInputStream **opens new connections** to datanodes as the client *reads through* the stream



Network topology and Hadoop

- What does two nodes being **close** mean?
- For high-volume data processing:
 - Limiting factor is the *rate at which data transfers take place*
 - Use **bandwidth** between the nodes as a measure of distance
- Measuring bandwidth between nodes difficult
 - Number of pairs of nodes in a cluster grows as a square of the number of nodes



Measuring network distances in Hadoop

- Network is represented as a **tree**
- The distance between the nodes is the **sum of their distances to its closest common ancestor**



Bandwidth available for the following scenarios gets progressively less

- Processes on the same node
- Different nodes on the same rack
- Nodes on different racks in the same data center
- Nodes in different data centers



Distance notation

- A node $n1$ on rack $r1$ in data center $d1$ is represented as $/d1/r1/n1$
- Distances in the four possible scenarios
 - $distance(/d1/r1/n1, /d1/r1/n1) = 0$
 - Processes on the same node
 - $distance(/d1/r1/n1, /d1/r1/n2) = 2$
 - Different nodes on the same rack
 - $distance(/d1/r1/n1, /d1/r2/n3) = 4$
 - Nodes on different racks in the same data center
 - $distance(/d1/r1/n1, /d2/r3/n4) = 6$
 - Nodes in different data centers



Network topology and distances

- Hadoop **does not divine** network topology
- Needs assists for doing so



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA
COMPUTER SCIENCE DEPARTMENT

HDFS

L25.47



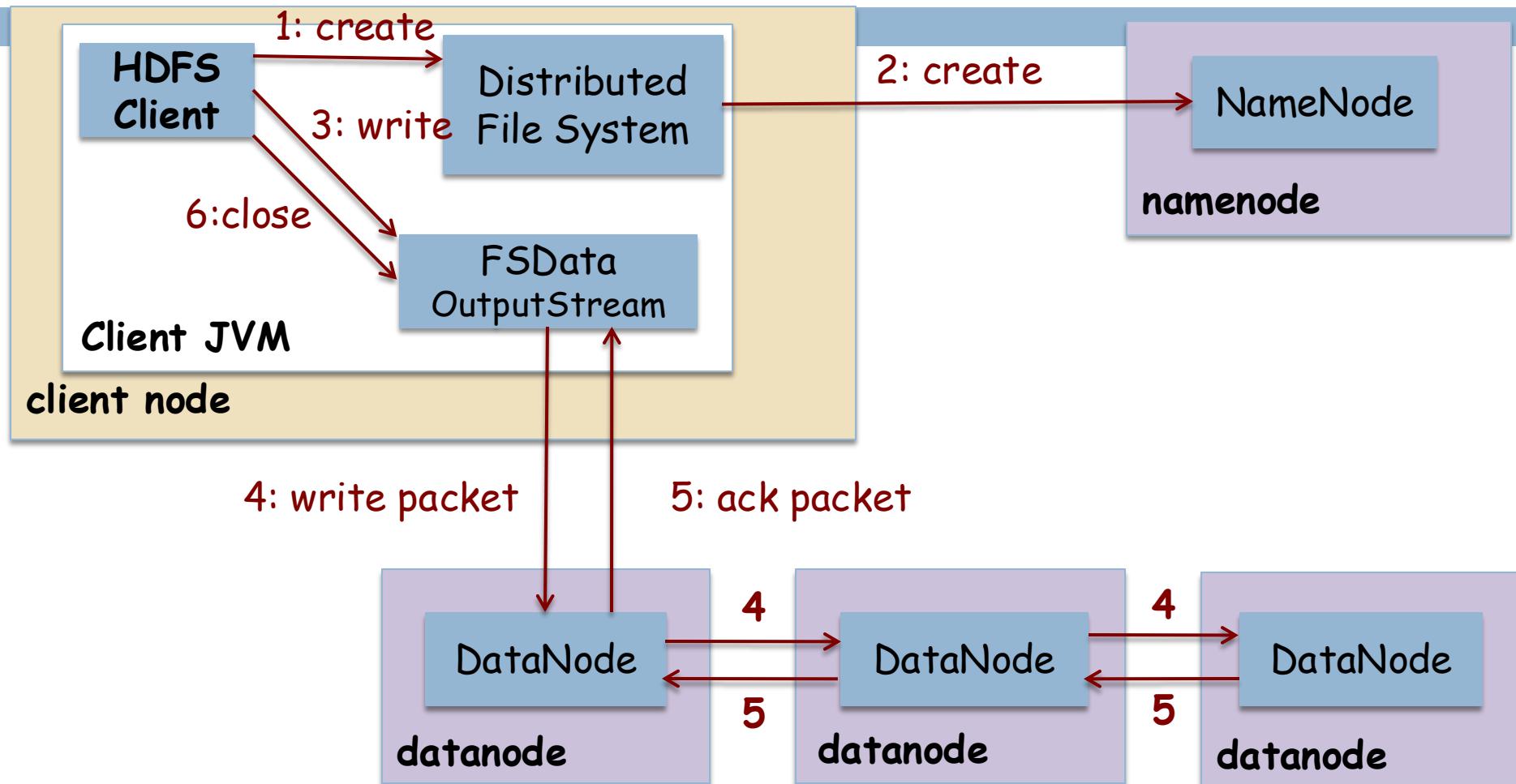
HDFS: WRITING DATA

File writes

- We will look at creating a new file and writing data to it
- File creation is done using `create()` on `DistributedFileSystem`
- `DistributedFileSystem` does an **RPC** to the namenode
 - Namenode checks existence of file and permissions
 - Creates file in the filesystem's namespace with no blocks in it



Data flow in HDFS [writes]



Anatomy of a file write

- `DistributedFileSystem` **returns an** `FSDataOutputStream` **for client to write data to**
- `FSDataOutputStream` **wraps a** `DFSOutputStream`
 - `DFSOutputStream` **handles communications with the datanodes and the namenode**



As the client writes data ...

- DFSOutputStream **splits it into packets**
 - Written to an internal queue, the **data queue**
- Data queue is consumed by the DataStreamer
- DataStreamer asks namenode to allocate new blocks
 - Pick list of suitable datanodes to store replicas
 - List of datanodes forms a **pipeline**



Assuming a replication level of 3

- DataStreamer streams packets to the first datanode in the pipeline
 - 1st datanode stores the packet and forwards it to the 2nd datanode in pipeline
- The second datanode stores the packet and forwards it to the 3rd (and last) datanode in pipeline



Managing acknowledgements

- DFSOutputStream maintains an internal queue of packets waiting to be ACKed by datanodes
 - This is the **ack queue**
- When is a packet removed from the ACK queue?
 - Only when it has been acknowledged by all the datanodes in the pipeline



Handling datanode failures during writes

[1 / 2]

- The pipeline is closed
- Current block on good datanodes is given a new identity
 - Allows *partial block on failed node* to be **deleted** if that datanode recovers later on



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA
COMPUTER SCIENCE DEPARTMENT

HDFS

L25.55

Handling datanode failures during writes

[2/2]

- Failed datanode is removed from the pipeline
- *Remainder* of the block's data is written to the two good datanodes in the pipeline
- Namenode **notices** block is *under-replicated*
 - Arranges for replicas to be created on another node
- Subsequent blocks are treated as normal



COLORADO STATE UNIVERSITY

Professor: SHRIDEEP PALICKARA
COMPUTER SCIENCE DEPARTMENT

HDFS

L25.56

It is possible that multiple datanodes fail while a block is being written

- As long as `dfs.replication.min` (default 1) replicas are written, the write will succeed
- Block is **asynchronously replicated** across cluster until its target replication factor is reached
 - `dfs.replication` (default 3)



When a client has finished writing data

- It calls `close()` on the stream
- **Flushes** all remaining packets to the datanode pipeline
 - Wait for acknowledgements before contacting the namenode to signal that file is complete
- Namenode knows about blocks that comprise the file
 - DataStreamer requests block allocations
 - Client only waits for blocks to be minimally replicated



The contents of this slide set are based on the following references

- Tom White. *Hadoop: The Definitive Guide*. 3rd Edition. Early Access Release. O'Reilly Press. ISBN: 978-1-449-31152-0. Chapters [2 and 3].

