ISIT312 Big Data Management

HDFS Interfaces

Dr Fenghui Ren

School of Computing and Information Technology -
University of Wollongong

HDFS Interfaces

Outline

Hadoop Cluster vs. Pseudo-Distributed Hadoop

A Hadoop cluster is deployed in a cluster of computer nodes

- As Hadoop is developed in Java, all Hadoop services sit on Java Virtual Machines running on the cluster nodes

Hadoop provides a pseudo-distributed mode on a single machine

- All Java Virtual Machines for necessary Hadoop services are running on a single machine
- In our case this machine is a Virtual Machine running under Ubuntu 14.04

HDFS provides the following interfaces to read, write, interrogate, and manage the filesystem

- The file system shell (Command-Line Interface): `hadoop fs` or `hdfs dfs`
- Hadoop Filesystem Java API
- Hadoop simple Web User Interface
- Other interfaces, such as RESTful proxy interfaces (e.g.,HttpFS)

HDFS Interfaces

Outline

Shell Interface to HDFS

Commands are provided in the shell Bash

Bash shell

```
$ which bash
/bin/bash
```

Hadoop's home directory

Home of Hadoop

```
$ cd $HADOOP_HOME
$ ls
bin include libexec logs README.txt share
etc lib LICENSE.txt NOTICE.txt sbin
```

You will mostly use scripts in the `bin` and `sbin` folders, and use jar files in the `share` folder

Hadoop Daemons

Hadoop daemons

```
$ jps
28530 SecondaryNameNode
11188 NodeManager
28133 NameNode
28311 DataNode
10845 ResourceManager
3542 Jps
```

Hadoop is running properly only if the above services are running

Shell Interface to HDFS

Create a HDFS user account (already created in a virtual machine used by us)

```
$ bin/hadoop fs -mkdir -p /user/bigdata
```
Creating home of user account

Create an folder `input`

```
$ bin/hadoop fs -mkdir input
```
Creating a folder

View the folders in Hadoop home

```
$ bin/hadoop fs -ls
Found 1 item
drwxr-xr-x - bigdata supergroup 0 2017-07-17 16:33 input
```
Listing home of user account

Upload a file to HDFS

```
$ bin/hadoop fs -put README.txt input
$ bin/hadoop fs -ls input
-rw-r--r-- 1 bigdata supergroup 1494 2017-07-12 17:53 input/README.txt
```
Uploading a file

Read a file in HDFS

```
$ bin/hadoop fs -cat input/README.txt
<contents of README.txt goes here>
```
Listing a file

Shell Interface to HDFS

The path in HDFS is represented as a URI with the prefix `hdfs://`

For example

- `hdfs://<hostname>:<port>/user/bigdata/input` refers to the `input` directory in HDFS under the user of `bigdata`
- `hdfs ://<hostname>:<port>/user/bigdata/input/README.txt` refers to the file `README.txt` in the above `input` directory in HDFS

When interacting with HDFS interface in the default setting, one can omitt IP, port, and user, and simply mention the directory or file

Thus, the full-spelling of `hadoop fs -ls input` is

```
hadoop fs -ls hdfs://<hostname>:
<port>/user/bigdata/input
```

Shell Interface to HDFS

Some of frequently used commands

Commands of Hadoop shell interface

Command	Description
-put	Upload a file (or files) from the local filesystem to HDFS
-mkdir	Create a directory in HDFS
-ls	List the files in a directory in HDFS
-cat	Read the content of a file (or files) in HDFS
-copyFromLocal	Copy a file from the local filesystem to HDFS (similar to put)
-copyToLocal	Copy a file (or files) from HDFS to the local filesystem
-rm	Delete a file (or files) in HDFS
-rm -r	Delete a directory in HDFS

HDFS Interfaces

Outline

Web Interface of HDFS

Overview 'localhost:8020' (active)

Started:	Thu Jul 27 10:53:21 AEST 2017
Version:	2.7.3, rbaa91f7c6bc9cb92be5982de4719c1c8af91ccff
Compiled:	2016-08-18T01:41Z by root from branch-2.7.3
Cluster ID:	CID-50dd23ff-1ec0-4860-b1fd-f8b0067f5b57
Block Pool ID:	BP-680435313-10.0.2.15-1499005796923

Summary

Security is off.

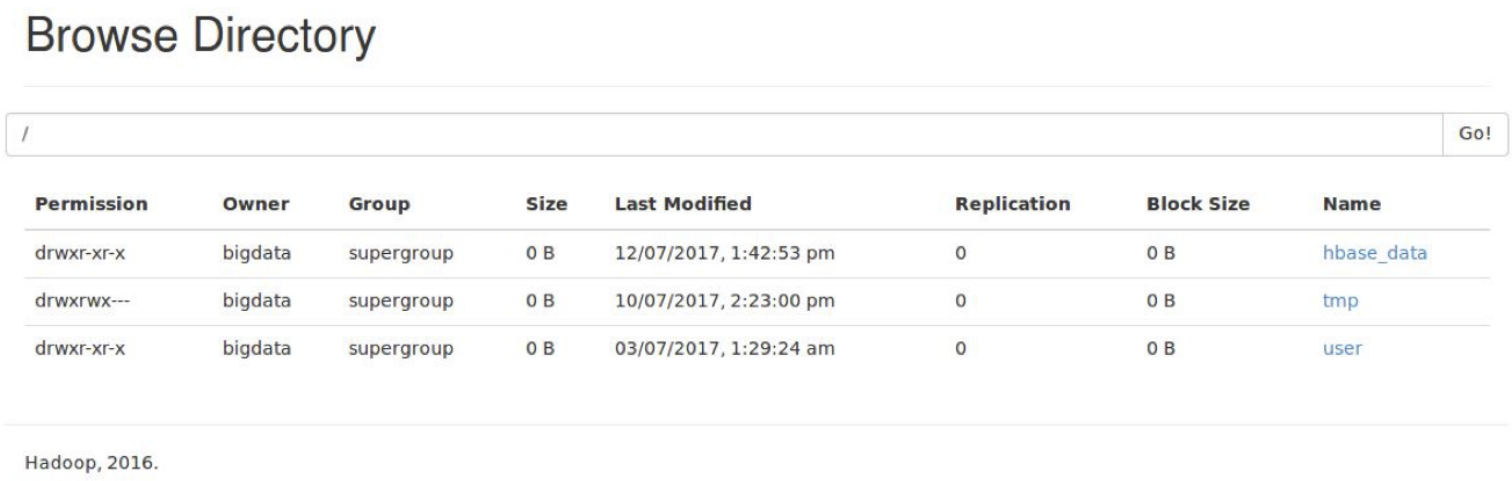
Safemode is off.

126 files and directories, 62 blocks = 188 total filesystem object(s).

Heap Memory used 39.84 MB of 263 MB Heap Memory. Max Heap Memory is 889 MB.

Non Heap Memory used 46.08 MB of 46.84 MB Commited Non Heap Memory. Max Non Heap Memory

Web Interface of HDFS



HDFS Interfaces

Outline

Java Interface to HDFS

A file in a Hadoop filesystem is represented by a Hadoop Path object

- Its syntax is URI
- For example, `hdfs://localhost:8020/user/bigdata/input/README.txt`

To get an instance of `FileSystem`, use the following factory methods

Factory methods

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

The following method gets a local filesystem instance

Get local file system method

```
public static  FileSystem getLocal(Configuration conf)  throws  IOException
```

Java interface to HDFS

A `Configuration` object is determined by the Hadoop configuration files or user-provided parameters

Using the default configuration, one can simply set

Configuration object

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

With a `FileSystem` instance in hand, we invoke an `open()` method to get the input stream for a file

Open method

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

A `Path` object can be created by using a designated URI

Path object

```
Path f =  new  Path(uri)
```

Java interface to HDFS

Putting together, we can create the following file reading application

Class FileSystemCat

```
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);
       FSDataInputStream in = null;
       Path path = new Path(uri);
       in = fs.open(path);
       IOUtils.copyBytes(in, System.out, 4096, true);
   }
}
```

Java interface to HDFS

The compilation simply uses the `javac` command, but it needs to point the dependencies in the class path.

Compilation

```
export HADOOP_CLASSPATH=$($HADOOP_HOME/bin/hadoop classpath)
javac -cp $HADOOP_CLASSPATH FileSystemCat.java
```

Then, a `jar` file is created and run as follows

jar file and processing

```
jar cvf FileSystemCat.jar FileSystemCat*.class
hadoop jar FileSystemCat.jar FileSystemcat input/README.txt
```

The output is the same as processing a command `hadoop fs -cat`

Java interface to HDFS

Suppose an input stream is created to read a local file

To write a file on HDFS, the simplest way is to take a `Path` object for the file to be created and return an output stream to write to

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

And then just copy the input stream to the output stream

Another, more flexible, way is to read the input stream into a buffer and then write to the output stream

Java interface to HDFS

A file writing application

File writing

```
public class FileSystemPut {
   public static void main(String[] args) throws Exception {
      String localStr = args[0];
      String hdfsStr = args[1];
      Configuration conf = new Configuration();
      FileSystem local = FileSystem.getLocal(conf);
      FileSystem hdfs = FileSystem.get(URI.create(hdfsStr), conf);
      Path localFile = new Path(localStr);
      Path hdfsFile = new Path(hdfsStr);
      FSDataInputStream in = local.open(localFile);
      FSDataOutputStream out = hdfs.create(hdfsFile);
      IOUtils.copyBytes(in, out, 4096, true);
    }
  }
```

Java interface to HDFS

Another file writing application

File writing

```
public class FileSystemPutAlt {
    public static void main(String[] args) throws Exception {
      String localStr = args[0];
      String hdfsStr = args[1];
      Configuration conf = new Configuration();
      FileSystem local = FileSystem.getLocal(conf);
      FileSystem hdfs = FileSystem.get(URI.create(hdfsStr), conf);
      Path localFile = new Path(localStr);
      Path hdfsFile = new Path(hdfsStr);
      FSDataInputStream in = local.open(localFile);
      FSDataOutputStream out = hdfs.create(hdfsFile);
      byte[] buffer = new byte[256];
      int bytesRead = 0;
      while( (bytesRead = in.read(buffer)) > 0) {
        out.write(buffer, 0, bytesRead);
      }
      in.close();
      out.close();
    }
  }
```

Java interface to HDFS

Other file system API methods

The method `mkdirs()` creates a directory

The method `getFileStatus()` gets the meta information for a single file or directory

The method `listStatus()` lists contents of files in a directory

The method `exists()` checks whether a file exists

The method `delete()` removes a file

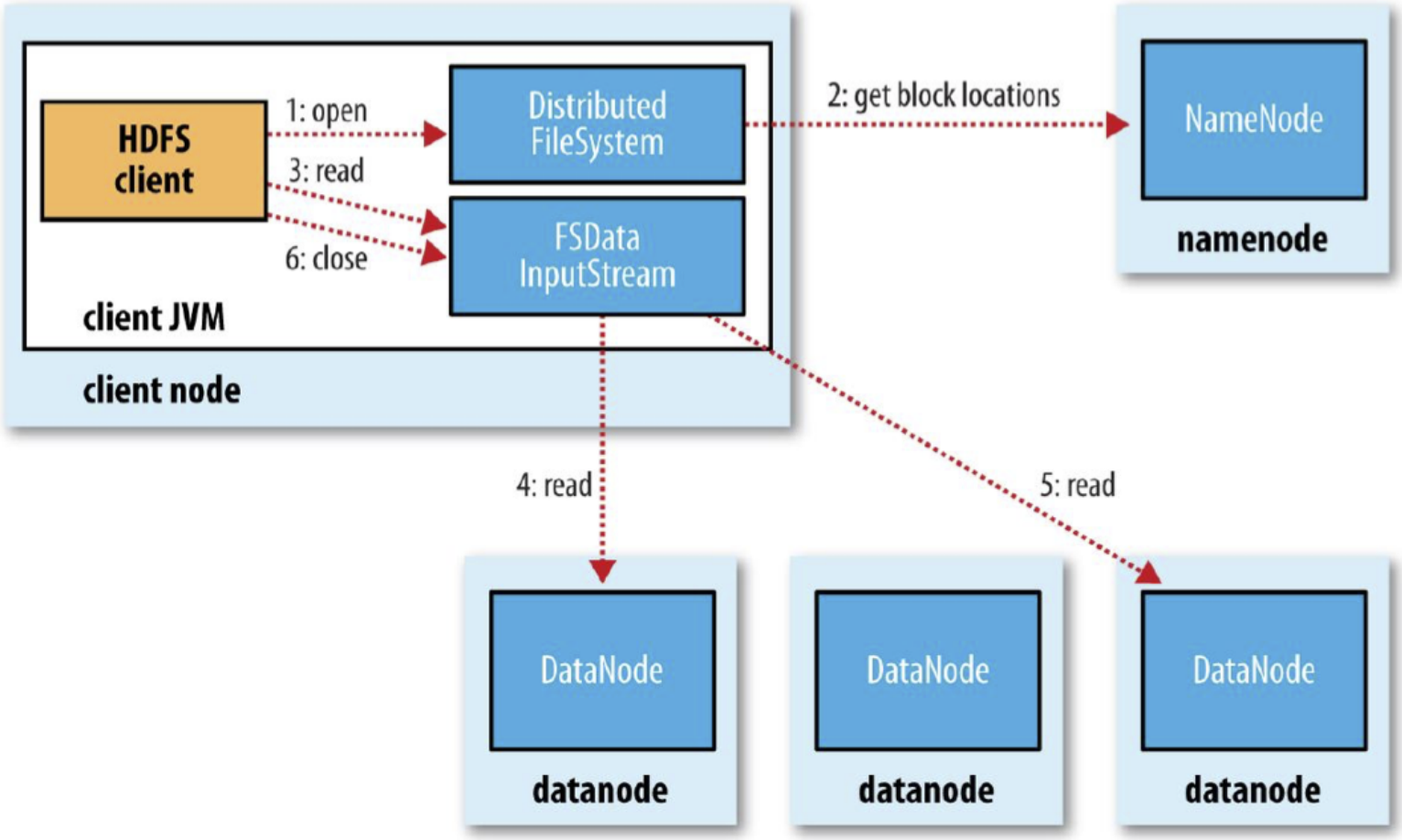
The Java API enables the implementation of customised applications to interact with HDFS

HDFS Interfaces

Outline

Internals of HDFS

What happens "inside" when we read data into HDFS ?



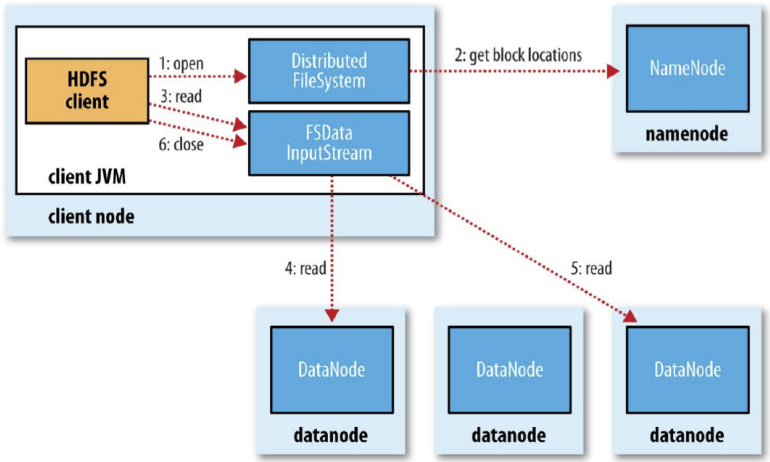
Internals of HDFS

Read data from HDFS

Step 1: The client opens the file it wishes to read by calling `open()` on the `FileSystem` object, which for HDFS is an instance of `DistributedFileSystem`

Step 2: `DistributedFileSystem` calls the `namenode`, using remote procedure calls (RPCs), to determine the locations of the first few blocks in the file

Step 3: The `DistributedFileSystem` returns an `FSDataInputStream` to the client and the client calls `read()` on the stream

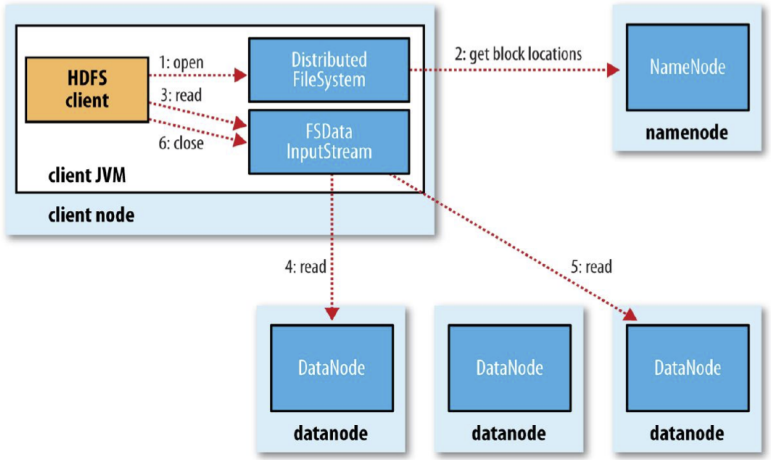


Internals of HDFS

Step 4: `FSDataInputStream` connects to the first datanode for the first block in the file, and then data is streamed from the datanode back to the client, by calling `read()` repeatedly on the stream

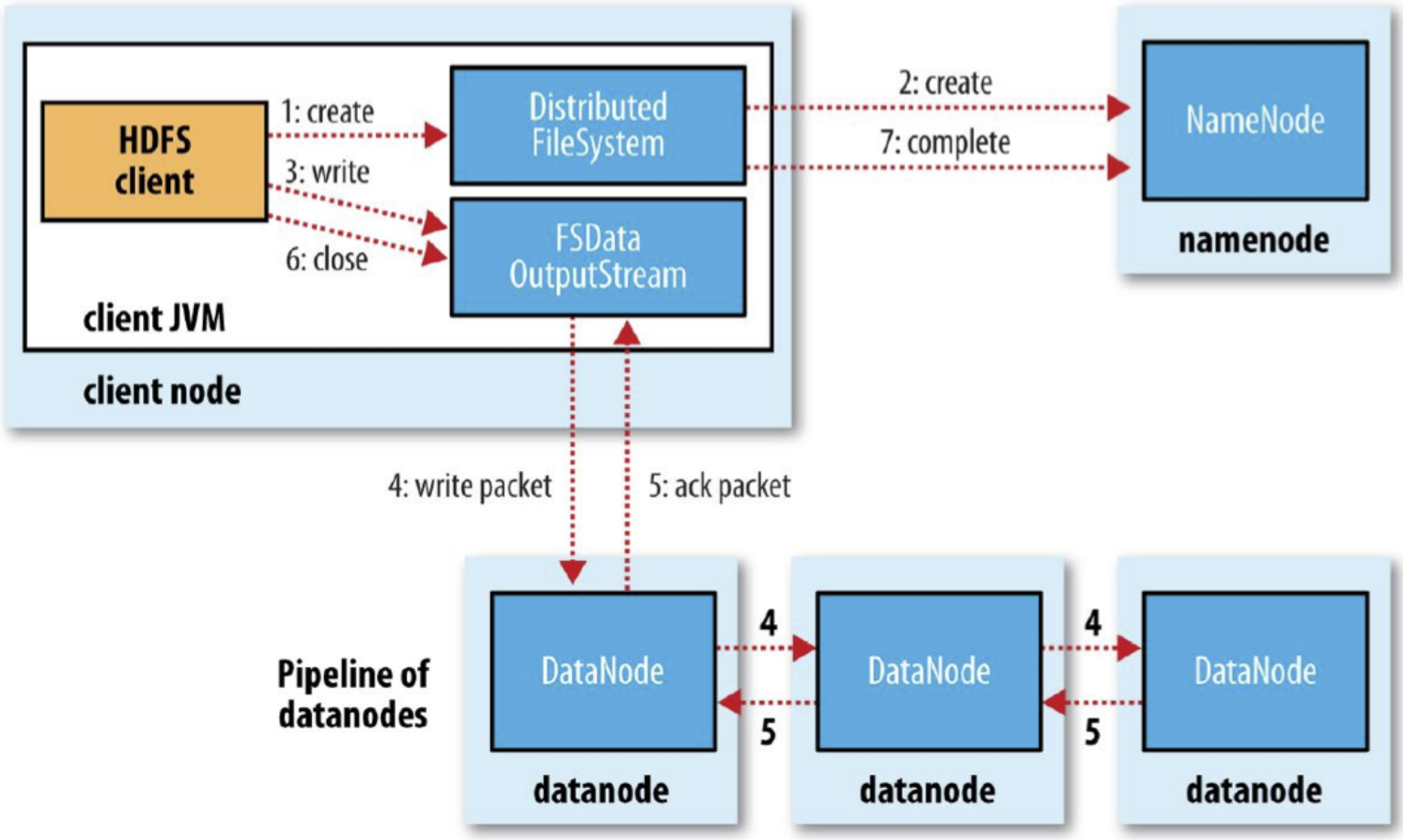
Step 5: When the end of the block is reached, `FSDataInputStream` will close the connection to the datanode, then find the best (possibly the same) datanode for the next block

Step 6: When the client has finished reading, it calls `close()` on the `FSDataInputStream`



Internals of HDFS

Write data into HDFS



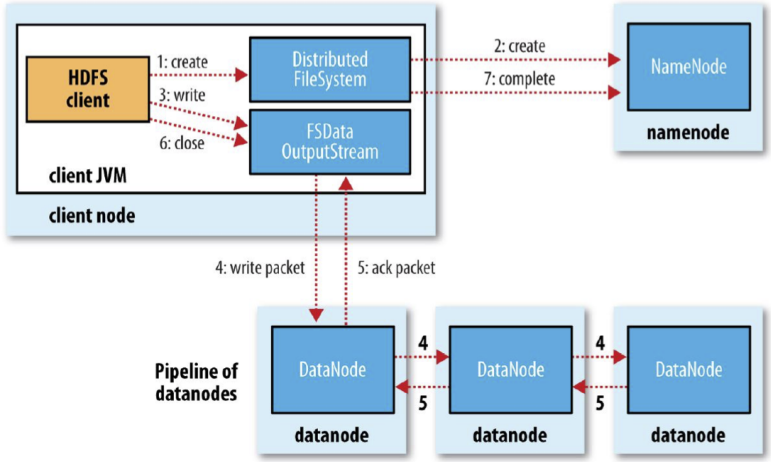
Internals of HDFS

Step 1: The client creates the file by calling `create()` on `DistributedFileSystem`

Step 2: `DistributedFileSystem` makes an RPC call to the `namenode` to create a new file in the file system namespace and returns an `FSDataOutputStream` for the client to start writing data to

Step 3: The client writes data into the `FSDataOutputStream`

Step 4: Data wrapped by the `FSDataOutputStream` is split into packages, which are flushed into a queue; data packages are sent to the blocks in a datanode and forwarded to other (usually two) datanodes

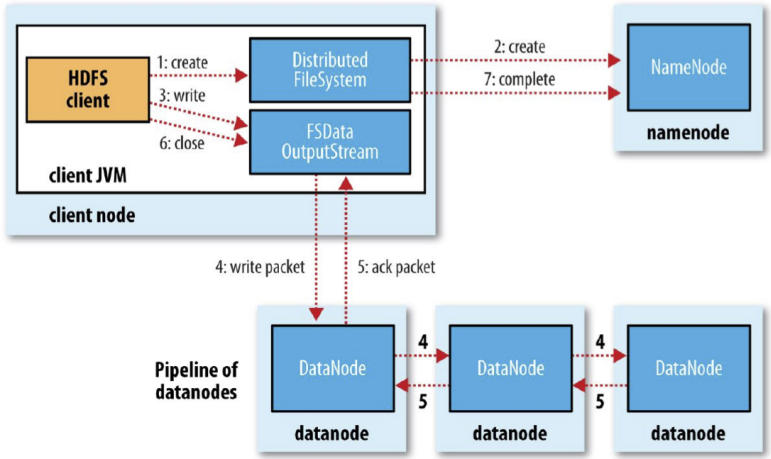


Internals of HDFS

Step 5: If `FSDataStream` receives an ack signal from the datanode the data packages are removed from the queue

Step 6: When the client has finished writing data, it calls `close()` on the stream

Step 7: The client signals the namenode that the writing is completed



References

Vohra D., Practical Hadoop ecosystem: a definitive guide to Hadoop-related frameworks and tools, Apress, 2016 (Available through UOW library)

Aven J., Hadoop in 24 Hours, SAMS Teach Yourself, SAMS 2017