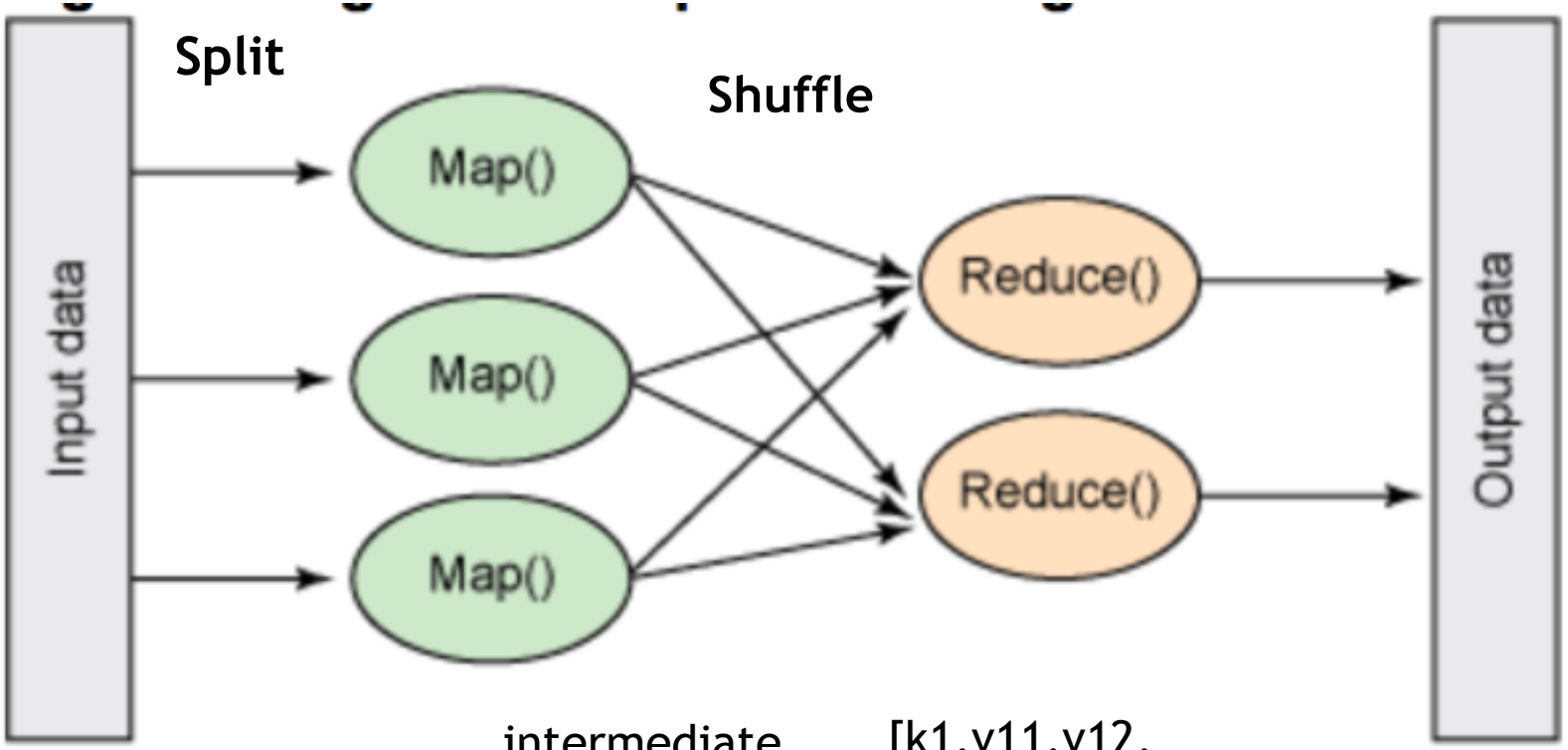
Hadoop Distributed Filesystem & I/O

Spring 2015,
X. Zhang
Fordham Univ.

MapReduce Programming Model



Input: a set of [key,value] pairs

intermediate [key,value] pairs

[k1,v11,v12, …]
[k2,v21,v22, …]
…

Output: a set of [key,value] pairs

Target Usage of HDFS

- storing very large files: MB, GB, TB, PB…
- streaming data access: data is write-once and read-many-times; data analysis involve a large potion or all data
  - optimize time to read whole data (from beginning to the end…)
- Commodity hardware (of various vendors), fault-tolerant

HDFS not suited for

- Low-latency data access
- Storing lots of small files
  - incur too much memory overhead in name node (which keeps meta data in memory)
- Multiple writers
  - can only be written by a single writer (program)
- Writing to arbitrary offsets in file
  - only support writing to the end of file

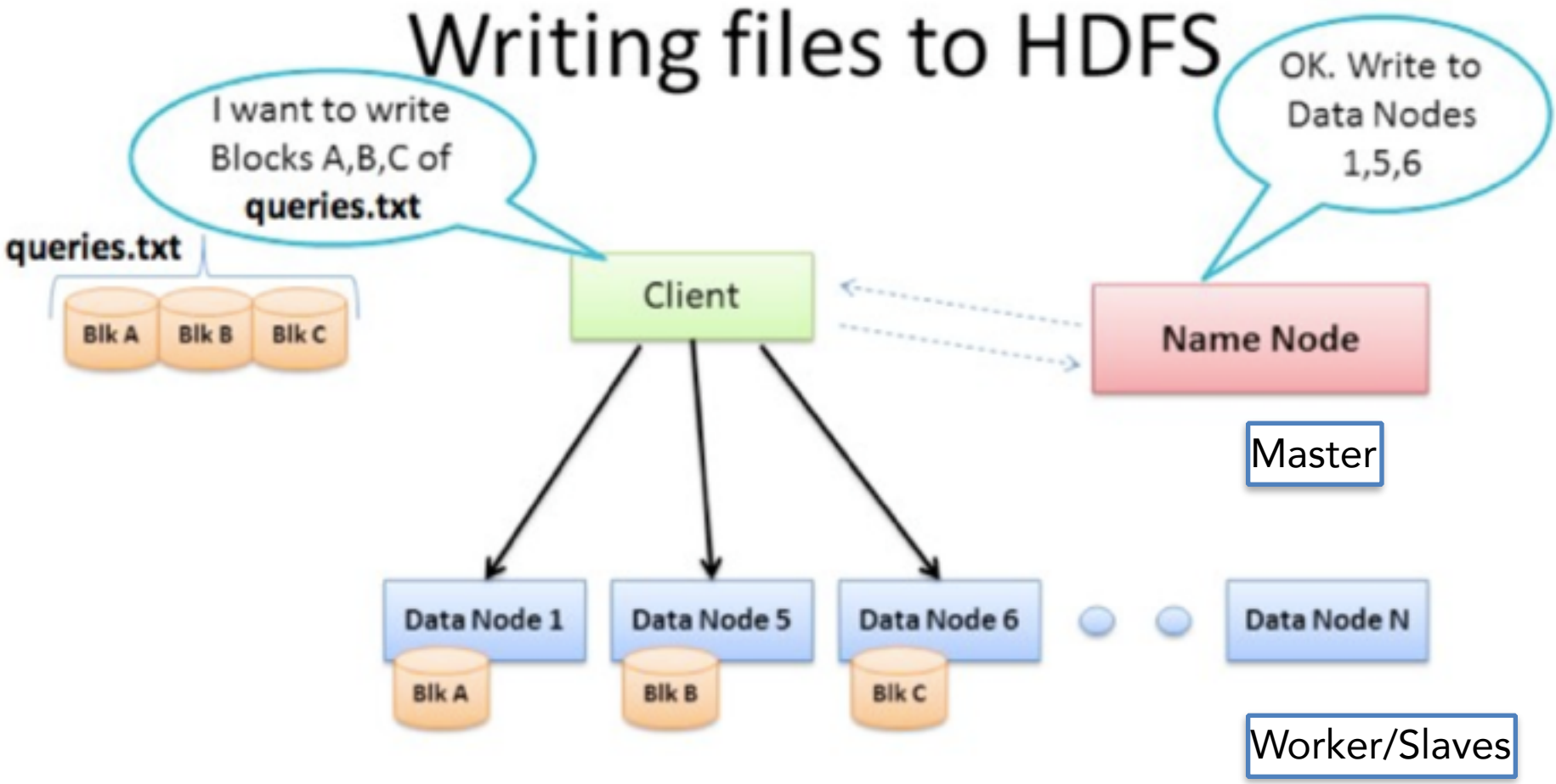
File systems blocks

- Disk drive has a block size (e.g., 512 B) => basic unit of data for disk read and write
  - When reading/writing disk, we don't go by bytes…
- Filesystems blocks, an integral multiple of disk block size, typically a few kilobytes
  - file system space is allocated in terms of block
  - A file is stored as multiple blocks, each stored as independent units (might be in different place in disk)
  - Transparent to filesystem user/programmer:
    - read(), write() calls take any length
  - Commands such as *df* and *fsck*, operates on filesystem block level

HDFS blocks

- 64MB per block by default
- Files in HDFS are broken into block-size chunks, stored as independent units
  - File smaller than 64 MB does not take a whole block of storage …
- Different blocks of a file can be stored on different nodes.
  - Support very large files
  - Each block replicated (for fault-tolerance and availability)
  - Map tasks normally operates on one block at a time => parallel processing, aggregate disk transfer rate…

HDFS: NameNode, DataNode



HDFS namenode

- Each HDFS cluster has one node operating as namenode (master)
- Manages file system namespace, i.e.,
  - filesystem tree structure
  - metadata for all files/directories: e.g., permission info, owner, last modification time, etc.
- Above info. stored in namenode's local disk (i.e., local file system) as two files:
  - namespace image
  - edit log
- Also cached in memory: where blocks of file are located … (reconstructed from datanodes…)
- HDFS federation: allow multiple namenodes each manage a subspace => support larger HDFS

HDFS datanode: worker/slave

- Stores file blocks and checksum for it.
- Retrieve blocks when client requests for it
- Update namenode with block information periodically
  - before updating: verify checksums
  - If checksum is incorrect for a particular block i.e. there is a ***disk level corruption for that block,*** it skips that block while reporting block information to namenode. => namenode replicates block somewhere else.
- Send heartbeat message to namenode => namenode detects datanode failure, and initiates replication of blocks
- Talk to each other to rebalance data, move and copy data around and keep replication high.

HDFS namenode error resilience

- Namenode failure renders whole filesystem useless
  - as it acts as a map to filesystem
- Ways to make namenode resilient to failure
  - backup to multiple filesystems (local disk, and a remote NFS)
  - secondary namenode
    - periodically merge namespace image and edit log of name node
    - lags behind namenode

HDFS High Availability

- a pair of namenodes in an active- standby configuration.
- When active namenode fails, standby takes over its duties to continue servicing client requests without a significant interruption.
- A few architectural changes are needed to allow this to happen:
  - The namenodes must use highly-available shared storage to share the edit log. (In the initial implementation of HA this will require an NFS filer, but in future releases more options will be provided, such as a BookKeeper-based system built on Zoo-Keeper.) When a standby namenode comes up it reads up to the end of the shared edit log to synchronize its state with the active namenode, and then continues to read new entries as they are written by the active namenode.
  - Datanodes must send block reports to both namenodes since the block mappings are stored in a namenode's memory, and not on disk.
  - Clients must be configured to handle namenode failover, which uses a mechanism that is transparent to users.

Outline

- Design of HDFS
- HDFS Concepts
- Command line interface (HDFS commands)
- URI and various file systems
- Accessing HDFS through HTTP, i.e., Web UI
- Java Programming API to HDFS

Access HDFS through commands

- **Pseudo-distributed mode configuration**
  - **Currently the default setting used …**

Two property:

fs.default.name, set to *hdfs://localhost/*, a default filesystem for Hadoop (i.e., unless otherwise specified, commands are referring to this file system)

e.g., hadoop fs -ls // list the default file system

Filesystems are specified by a URI: hdfs URI to configure Hadoop to use HDFS by default.

HDFS daemons will use this property to determine the host and port for HDFS namenode. (Here it's on localhost, on the default HDFS port, 8020.)

And HDFS clients will use this property to work out where the namenode is running so they can connect to it.

dfs.replication set to 1

File System Operations

- **Read files, creating directories, moving files, copying files, deleting files, and listing directories**
- **hadoop fs -help ## to see a summary of all filesystem commands**

% **hadoop fs -copyFromLocal input/docs/quangle.txt hdfs://localhost/user/tom/ quangle.txt**

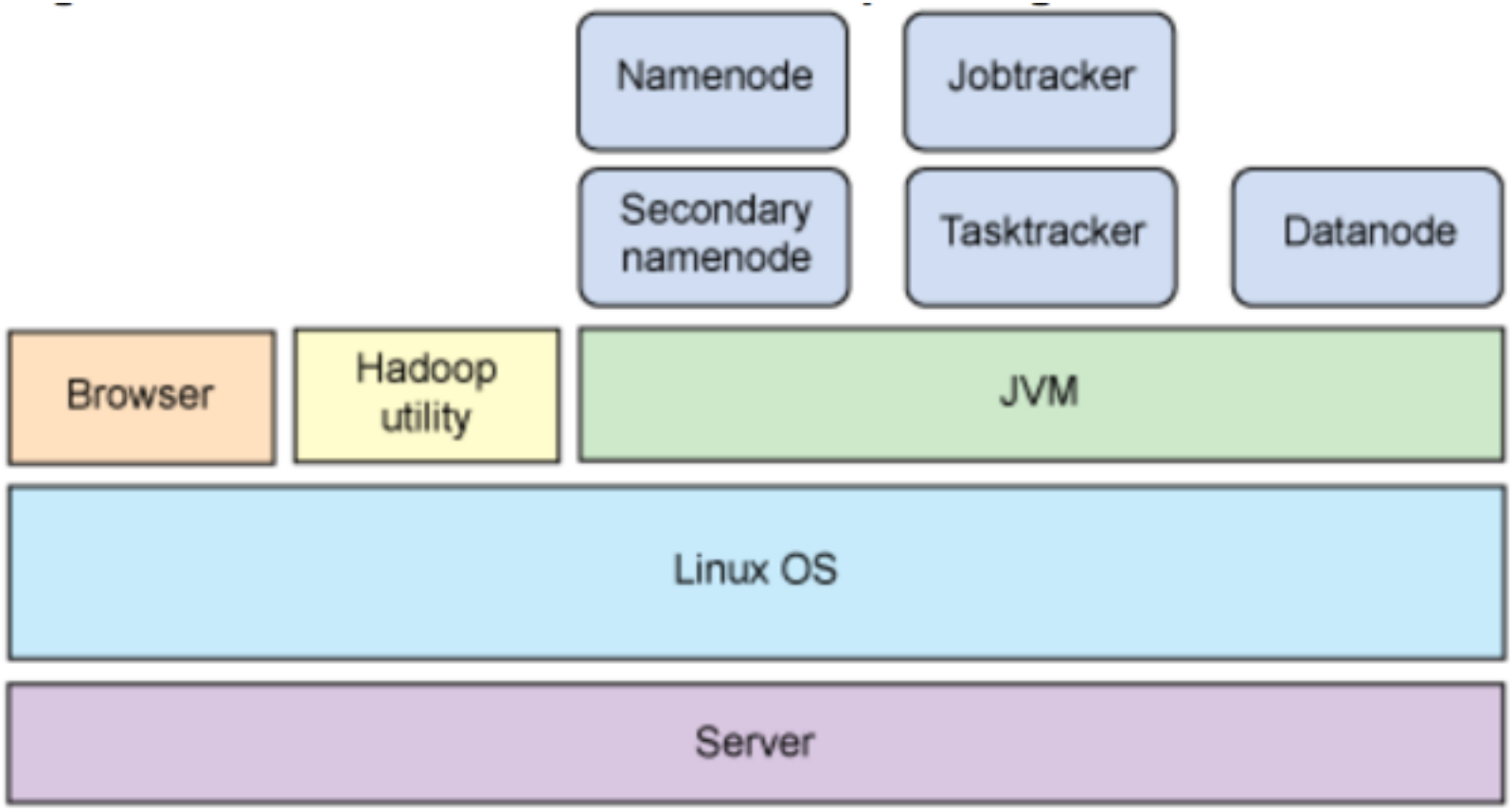
% **hadoop fs -copyFromLocal input/docs/quangle.txt /user/tom/quangle.txt**

% **hadoop fs -copyFromLocal input/docs/quangle.txt quangle.txt**

Other commands: ls, mkdir …

```
[zhang@puppet ~]$ hadoop fs -ls
Found 12 items
-rw-r--r--   1 zhang supergroup       1494 2015-01-27 16:20 Filter.class
-rw-r--r--   1 zhang supergroup       1353 2015-01-27 16:20 Max.class
drwxr-xr-x   - zhang supergroup          0 2015-01-18 16:37 in0
drwxr-xr-x   - zhang supergroup          0 2015-01-13 17:36 input
drwxr-xr-x   - zhang supergroup          0 2015-01-18 16:35 ncdc
drwxr-xr-x   - zhang supergroup          0 2015-02-03 20:37 out2
drwxr-xr-x   - zhang supergroup          0 2015-02-03 19:21 out3
drwxr-xr-x   - zhang supergroup          0 2015-01-18 16:46 out_streaming
drwxr-xr-x   - zhang supergroup          0 2015-01-18 16:58 out_streaming_new
drwxr-xr-x   - zhang supergroup          0 2015-01-28 16:35 out_streaming_new3
drwxr-xr-x   - zhang supergroup          0 2015-02-08 14:04 output
drwxr-xr-x   - zhang supergroup          0 2015-01-27 16:00 record_count
```

Pseudo-distributed mode



To check whether they are running:

ps -aef | grep namenode

Apache Hadoop Main 2.6.0 API

- For now, focus on Package org.apache.hadoop.mapreduce (replace org.apache.hadoop.mapred).
- Use Index to look up class/interface by name
  - Mapper, Reducer: a generic type (C++ template class) with **type parameters**
  - TextInputFormat, default InputFormat used by mapper, decides how input data is parsed into <key,value> pairs …

Outline

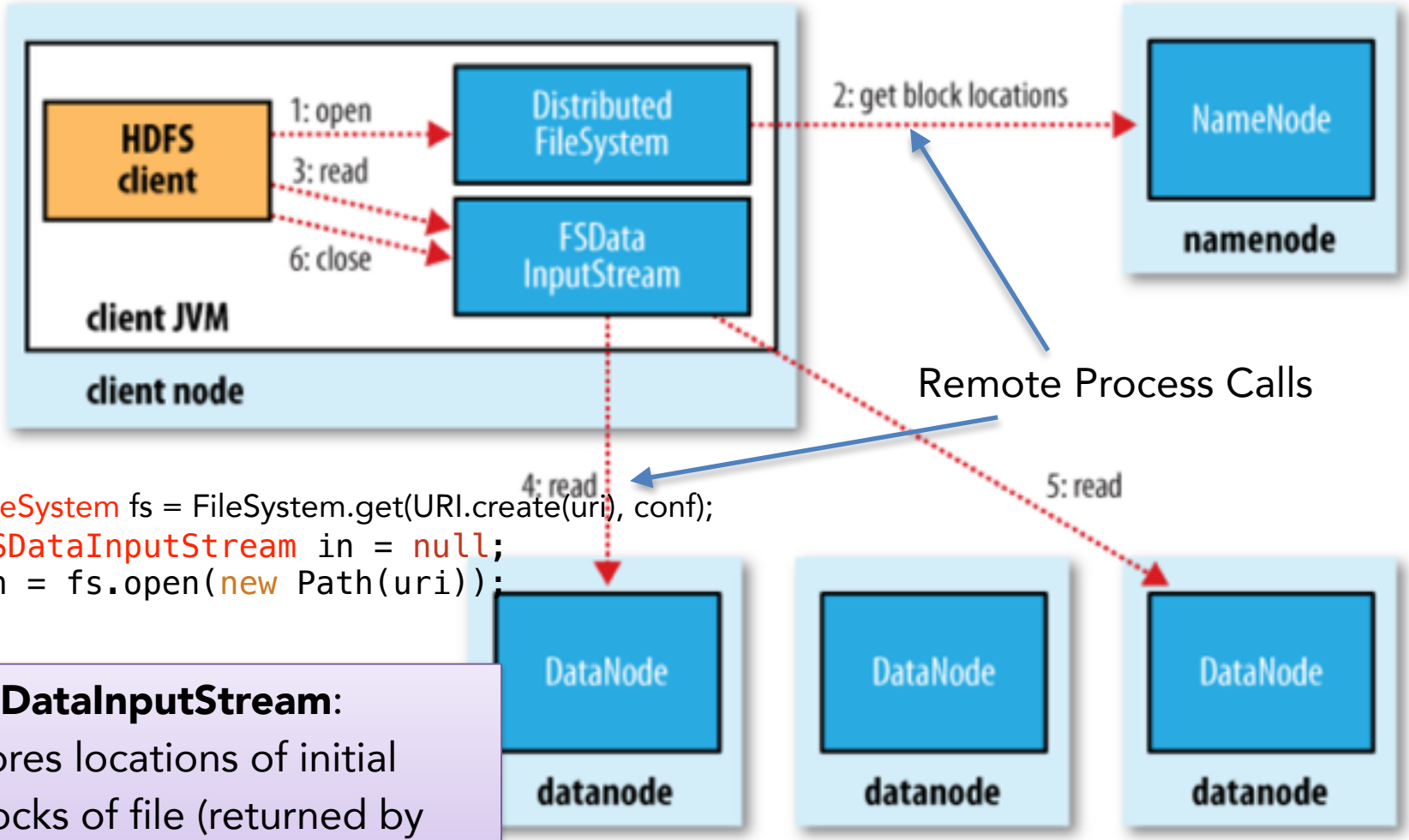
- Design of HDFS
- HDFS Concepts
- Command line interface (HDFS commands)
- URI and various file systems
- Accessing HDFS through HTTP, i.e., Web UI
- Java Programming API to HDFS
  - Code walk-through
  - Learn from example Codes
  - Use the Apache online manual for MapReduce API
- Data Flows
- Advanced topics: data ingest, distcp, archives

Read HDFS File

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;
public class FileSystemDoubleCat {

  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;
    try {
      in = fs.open(new Path(uri));
      IOUtils.copyBytes(in, System.out, 4096, false);
      in.seek(0); // go back to the start of the file
      IOUtils.copyBytes(in, System.out, 4096, false);
    } finally {
      IOUtils.closeStream(in);
    }
  }
}
```

HDFS File Read Operation



1. FileSystem fs = FileSystem.get(URI.create(uri), conf);

```
FSDataInputStream in = null;
in = fs.open(new Path(uri));
```

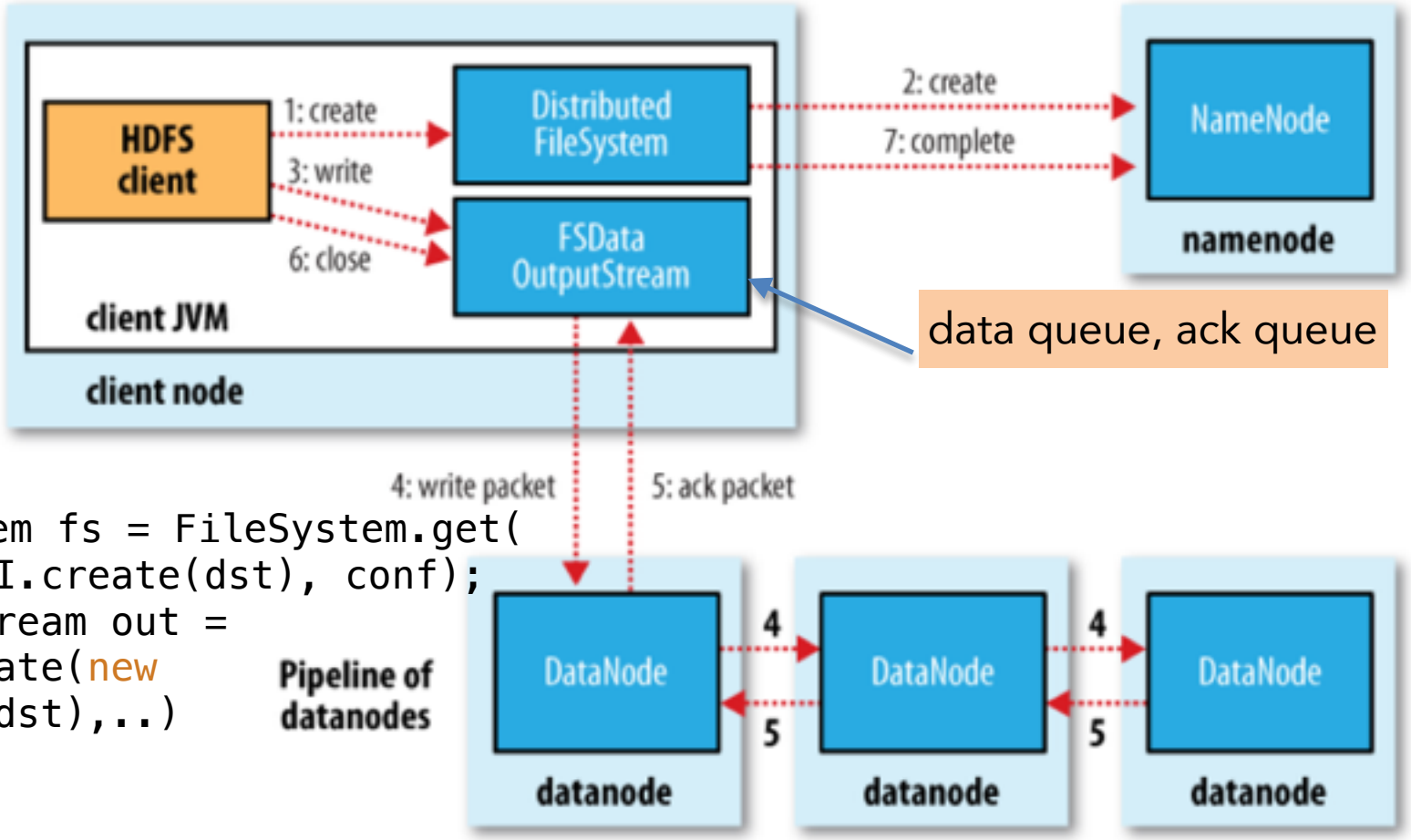
FSDataInputStream:
stores locations of initial blocks of file (returned by NameNode)
connect to closest datenode.

NameNode: how does it work?

- a Listener object listens to TCP port serving RPC requests from client, accepts new connections, and adds to *Server* object's *connectionList*
- a number of RPC Reader threads read requests from connections in connectionList, decode the RPC requests, and add them to rpc call queue – *Server.callQueue*.
- Actual worker threads kick in – these are the *Handler* threads.

  The threads pick up RPC calls and process them. The processing involves the following:
  - First grab the write lock for the namespace
  - Change the in-memory namespace
  - Write to the in-memory FSEdits log (journal)
- Now, release the write lock on the namespace. Note that the journal has not been sync'd yet – this means we cannot return success to the RPC client yet
- Next, each handler thread calls logSync. Upon returning from this call, it is guaranteed that the logfile modification have been sync'd to disk.

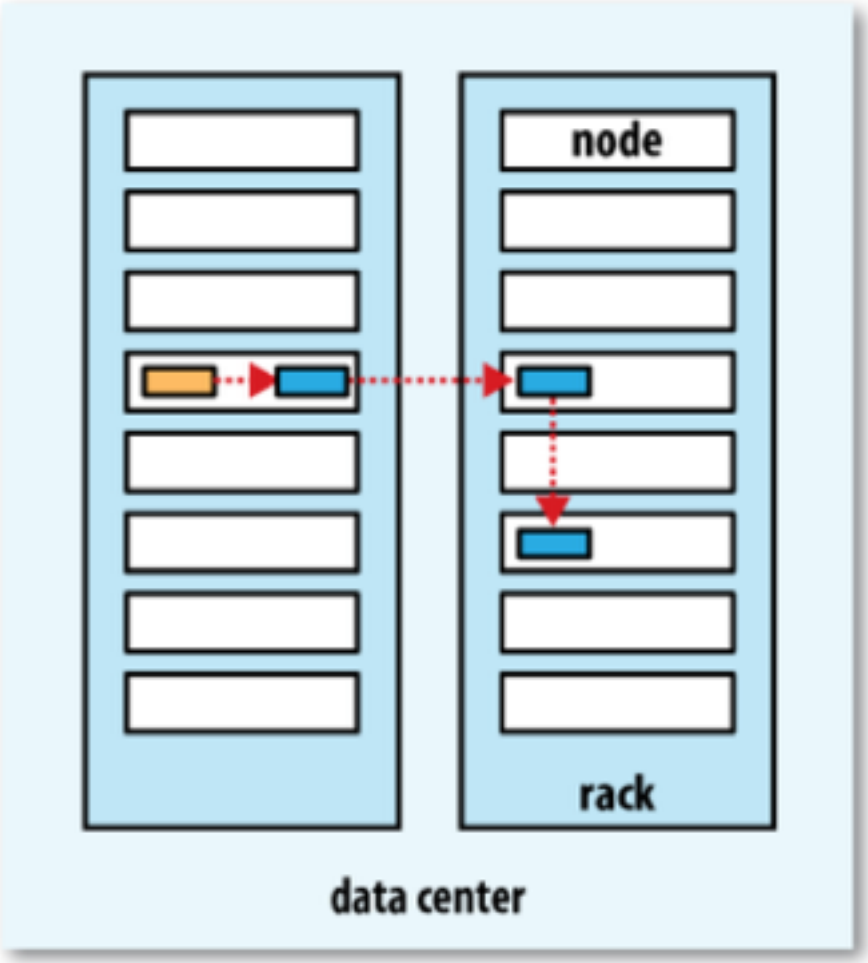
HDFS Write Operation




```
FileSystem fs = FileSystem.get(
      URI.create(dst), conf);
outputStream out =
  fs.create(new
   Path(dst),..)
```

Default Replication Strategy

- two properties
  - dfs.replication.min
  - dfs.replication
- First replica: same node as client
- Second replica: off-rack at random
- third replica: same rack as second one, at random
- other replica: randomly placed on the cluster



Coherency Model

- A coherency model for a filesystem describes the data visibility of reads and writes for a file.
- After creating a file, it is visible in filesystem namespace, as expected:

```
Path p = new Path("p"); fs.create(p); assertThat(fs.exists(p), is(true));
```

However, any content written to the file is not guaranteed to be visible, even if the stream is flushed. So the file appears to have a length of zero:

```
Path p = new Path("p");
OutputStream out = fs.create(p); out.write("content".getBytes("UTF-8")); out.flush(); assertThat(fs.getFileStatus(p).getLen(), is(0L));
```

- it is always the current block being written that is not visible to other readers.HDFS provides a method for forcing all buffers to be synchronized to the datanodes via the sync() method on FSDataOutputStream. After a successful return from sync(), HDFS guarantees that the data written up to that point in the file is persisted and visible to all new readers:[8]

```
Path p = new Path("p");
FSDataOutputStream out = fs.create(p); out.write("content".getBytes("UTF-8"));
out.flush();
out.sync();
assertThat(fs.getFileStatus(p).getLen(), is(((long) "content".length())));
```

Outline

- Design of HDFS
- HDFS Concepts
- Command line interface (HDFS commands)
- URI and various file systems
- Accessing HDFS through HTTP, i.e., Web UI
- Java Programming API to HDFS
  - Code walk-through
  - Learn from example Codes
  - Use the Apache online manual for MapReduce API
- Data Flows
- Advanced topics: data ingest, distcp, archives
- Hadoop I/O: compression/decompression, splittable compression scheme, serialization (writables)