

Hadoop & HDFS

INF 55x

Wensheng Wu

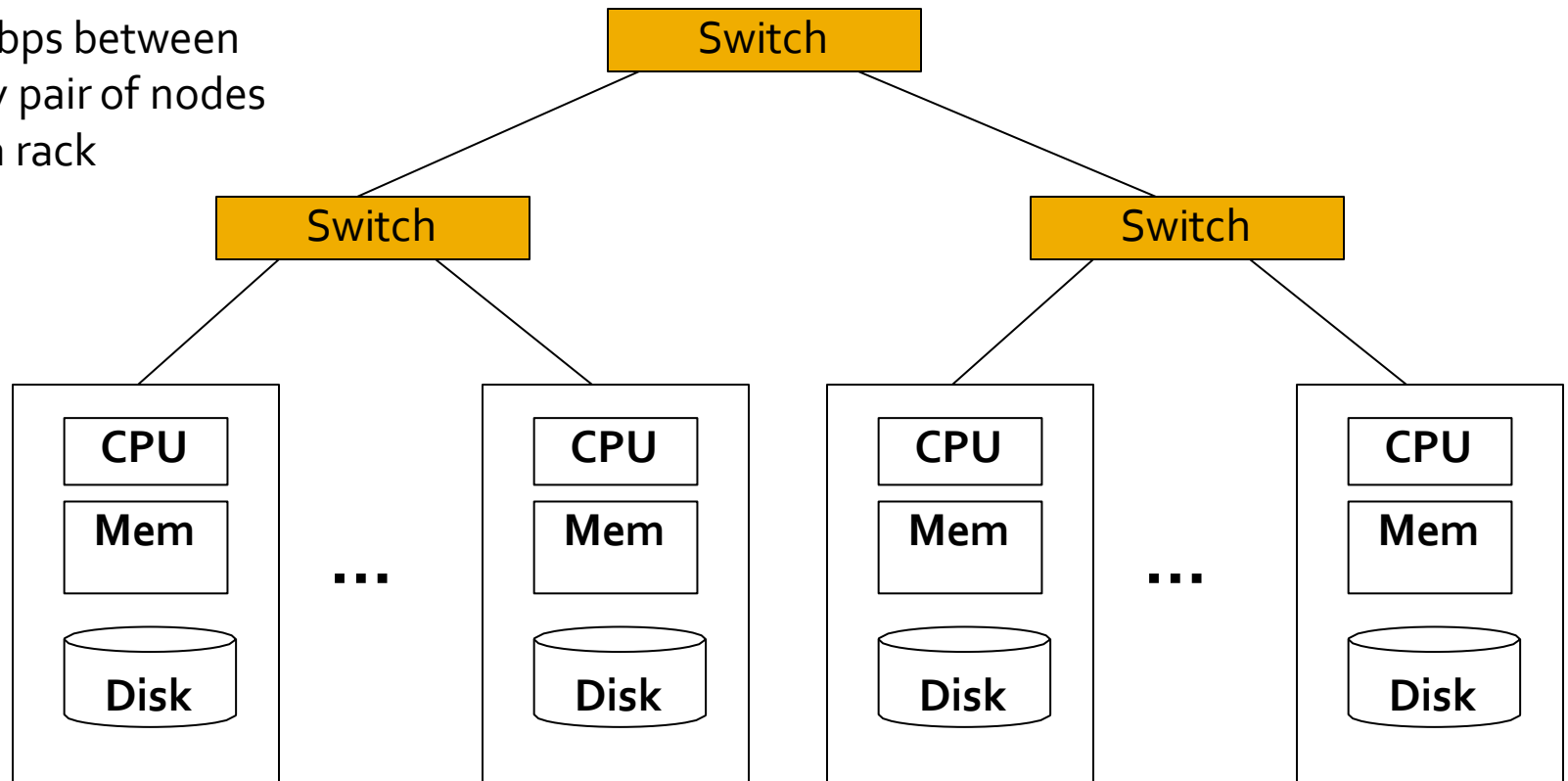
Hadoop

- A large-scale distributed batch-processing infrastructure
- Large-scale:
 - Handle a large amount of data and computation
- Distributed:
 - Distribute data & computation over multiple machines
- Batch processing
 - Process a series of jobs without human intervention

Cluster Architecture

2-10 Gbps backbone between racks

1 Gbps between
any pair of nodes
in a rack



Each rack contains 16-64 nodes

In 2011 it was guestimated that Google had 1M machines, <http://bit.ly/Shh0RO>³



History

- 1st version released by Yahoo! in 2006
 - named after an elephant toy
- Originated from Google's work
 - GFS: Google File System (2003)
 - MapReduce (2004)



Roadmap

- Hadoop architecture



- HDFS

- MapReduce

- Installing Hadoop & HDFS

Key components

- HDFS (Hadoop distributed file system)
 - Distributed data storage with **high reliability**
- MapReduce
 - A parallel, distributed computational paradigm
 - With a **simplified** programming model

HDFS

- Data are distributed among multiple data nodes
 - Data nodes may be added on demand for more storage space
- Data are replicated to cope with node failure
 - Typically replication factor: 2 or 3
- Requests can go to any replica
 - Removing the bottleneck (as in single file server)

HDFS architecture

NameNode:

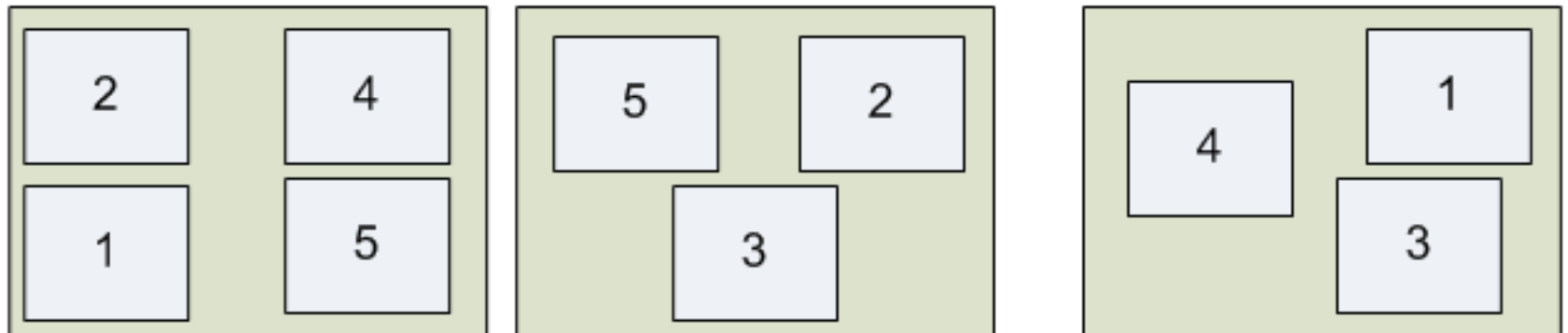
Stores metadata only

METADATA:

/user/aaron/foo → 1, 2, 4

/user/aaron/bar → 3, 5

DataNodes: Store blocks from files



HDFS has ...

- A single NameNode, storing meta data:
 - A hierarchy of directories and files
 - Attributes of directories and files (in inodes), e.g., permission, access/modification times, etc.
 - Mapping of files to blocks on data nodes
- A number of DataNodes:
 - Storing contents/blocks of files

Compute nodes

- Data nodes are compute nodes too
- Advantage:
 - Allow schedule computation close to data

HDFS also has ...

- A SecondaryNameNode
 - Maintaining checkpoints/images of NameNode
 - For recovery
- In a single-machine setup
 - all nodes correspond to the same machine

Metadata in NameNode

- NameNode has an inode for each file and dir
- Record attributes of file/dir such as
 - Permission
 - Access time
 - Modification time
- Also record mapping of files to blocks

Mapping information in NameNode

- E.g., file /user/aaron/foo consists of blocks 1, 2, and 4
- Block 1 is stored on data nodes 1 and 3
- Block 2 is stored on data nodes 1 and 2
- ...

Block size

- HDFS: 128 MB (version 2 & above)
 - Much larger than disk block size (4KB)
- Why larger size in HDFS?
 - Reduce metadata required per file
 - Fast streaming read of data (since larger amount of data are sequentially laid out on disk)
 - Thus good for workload with largely sequential read of large file

HDFS (vs. NFS)

- HDFS exposes the concept of blocks to client
- Reading and writing are done in two phases
 - Phase 1: client asks NameNode for block locations
 - By calling `getBlockLocations()`, if reading
 - Or calling `addBlock()` for allocating new blocks, if writing (need to call `create()/append()` first)
 - Phase 2: client talks to DataNode for data transfer
 - Reading blocks via `readBlock()` or writing blocks via `writeBlock()`

Client and Namenode communication

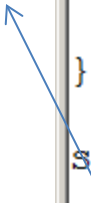
- Source code (version 2.8.1)
 - Definition of protocol
 - ClientNamenodeProtocol.proto
 - <hadoop-src-dir>\hadoop-hdfs-project\hadoop-hdfs-client\src\main\proto
 - Implementation
 - ClientProtocol.java
 - <hadoop-src-dir>\hadoop-hdfs-project\hadoop-hdfs-client\src\main\java\org\apache\hadoop\hdfs\protocol

Key operations

- Reading:
 - `getBlockLocations()`
- Writing
 - `create()`
 - `append()`
 - `addBlock()`

getBlockLocations


Before reading, client needs to first obtain locations of blocks



```
message GetEditsFromTxidResponseProto {
  required EventsListProto eventsList = 1;
}

service ClientNameNodeProtocol {
  rpc getBlockLocations(GetBlockLocationsRequestProto)
    returns(GetBlockLocationsResponseProto);
  rpc getServerDefaults(GetServerDefaultsRequestProto)
    returns(GetServerDefaultsResponseProto);
  rpc create(CreateRequestProto) returns(CreateResponseProto);
  rpc append(AppendRequestProto) returns(AppendResponseProto);
  rpc setReplication(SetReplicationRequestProto)
    returns(SetReplicationResponseProto);
  rpc setStoragePolicy(SetStoragePolicyRequestProto)
    returns(SetStoragePolicyResponseProto);
  rpc getStoragePolicies(GetStoragePoliciesRequestProto)
    returns(GetStoragePoliciesResponseProto);
  rpc setPermission(SetPermissionRequestProto)
    returns(SetPermissionResponseProto);
}
```

getBlockLocations

- Input:  Compare to NFS:
 - NFSPROC_LOOKUP(parent FH, file/dir name)
 - NFSPROC_READ(file handle, offset, count)
 - File name
 - Offset (to start reading)
 - Length (how much data to be read)
- Output:
 - Located blocks (data nodes + offsets)

```

////////////////////////////////////
// File contents
////////////////////////////////////
/**
 * Get locations of the blocks of the specified file
 * within the specified range.
 * DataNode locations for each block are sorted by
 * the proximity to the client.
 * <p>
 * Return {@link LocatedBlocks} which contains
 * file length, blocks and their locations.
 * DataNode locations for each block are sorted by
 * the distance to the client's address.
 * <p>
 * The client will then have to contact
 * one of the indicated DataNodes to obtain the actual data.
 *
 * @param src file name
 * @param offset range start offset
 * @param length range length
 *
 * @return file length and array of blocks with their locations
 *
 * @throws org.apache.hadoop.security.AccessControlException If access is
 *         denied
 * @throws java.io.FileNotFoundException If file <code>src</code> does not
 *         exist
 * @throws org.apache.hadoop.fs.UnresolvedLinkException If <code>src</code>
 *         contains a symlink
 * @throws IOException If an I/O error occurred
 */
@Idempotent
LocatedBlocks getBlockLocations(String src, long offset, long length)
    throws IOException;

```

../java/...hdfs/protocol/LocatedBlocks.java

```
public class LocatedBlocks {
    private final long fileLength;
    // array of blocks with prioritized locations
    private final List<LocatedBlock> blocks;
    private final boolean underConstruction;
    private final LocatedBlock lastLocatedBlock;
    private final boolean isLastBlockComplete;
    private final FileEncryptionInfo fileEncryptionInfo;

    public class LocatedBlock {
        private final ExtendedBlock b;
        private long offset; // offset of the first byte of the block in the file
        private final DatanodeInfoWithStorage[] locs;
        /** Cached storage ID for each replica */
        private final String[] storageIDs;
        /** Cached storage type for each replica, if reported. */
        private final StorageType[] storageTypes;
        // corrupt flag is true if all of the replicas of a block are corrupt.
        // else false. If block has few corrupt replicas, they are filtered and
        // their locations are not part of this object
        private boolean corrupt;
        private Token<BlockTokenIdentifier> blockToken = new Token<BlockTokenIdentifier>();
        /**
         * List of cached datanode locations
         */
        private DatanodeInfo[] cachedLocs;


        // Used when there are no locations
        private static final DatanodeInfoWithStorage[] EMPTY_LOCS =
            new DatanodeInfoWithStorage[0];
    }
}
```

Block
File offset associated
with this block
Data nodes with
replicas of block

Create/append a file

```
message GetEditsFromTxidResponseProto {  
    required EventsListProto eventsList = 1;  
}  
  
service ClientNamenodeProtocol {  
    rpc getBlockLocations (GetBlockLocationsRequestProto)  
        returns (GetBlockLocationsResponseProto);  
    rpc getServerDefaults (GetServerDefaultsRequestProto)  
        returns (GetServerDefaultsResponseProto);  
    rpc create (CreateRequestProto) returns (CreateResponseProto);  
    rpc append (AppendRequestProto) returns (AppendResponseProto);  
    rpc setReplication (SetReplicationRequestProto)  
        returns (SetReplicationResponseProto);  
    rpc setStoragePolicy (SetStoragePolicyRequestProto)  
        returns (SetStoragePolicyResponseProto);  
    rpc getStoragePolicies (GetStoragePoliciesRequestProto)  
        returns (GetStoragePoliciesResponseProto);  
    rpc setPermission (SetPermissionRequestProto)  
        returns (SetPermissionResponseProto);  
}
```

This opens the file for
create/append



Creating a file

- Needs to specify:
 - Path to the file to be created, e.g., /foo/bar
 - Permission mask
 - Client name
 - Flag on whether to overwrite (entire file!) if already exists
 - How many replicas
 - Block size


```

/**
 * Create a new file entry in the namespace.
 * <p>
 * This will create an empty file specified by the source path.
 * The path should reflect a full path originated at the root.
 * The name-node does not have a notion of "current" directory for a client.
 * <p>
 * Once created, the file is visible and available for read to other clients.
 * Although, other clients cannot {@link #delete(String, boolean)}, re-create
 * or {@link #rename(String, String)} it until the file is completed
 * or explicitly as a result of lease expiration.
 * <p>
 * Blocks have a maximum size. Clients that intend to create
 * multi-block files must also use
 * {@link #addBlock}
 *
 * @param src path of the file being created.
 * @param masked masked permission.
 * @param clientName name of the current client.
 * @param flag indicates whether the file should be
 * overwritten if it already exists or create if it does not exist or append.
 * @param createParent create missing parent directory if true
 * @param replication block replication factor.
 * @param blockSize maximum block size.
 * @param supportedVersions CryptoProtocolVersions supported by the client
 *
 * ...
 */
@AtMostOnce
HdfsFileStatus create(String src, FsPermission masked,
    String clientName, EnumSetWritable<CreateFlag> flag,
    boolean createParent, short replication, long blockSize,
    CryptoProtocolVersion[] supportedVersions)
    throws IOException;

```

→ A hierarchy of files and directories

→ Creating a new file

Allocating new blocks for writing

Asking NameNode to allocate a new block
+ data nodes holding its replicas

```
rpc setPermission(SetPermissionRequestProto)
    returns (SetPermissionResponseProto);
rpc setOwner(SetOwnerRequestProto) returns (SetOwnerResponseProto);
rpc abandonBlock(AbandonBlockRequestProto) returns (AbandonBlockResponseProto);
rpc addBlock(AddBlockRequestProto) returns (AddBlockResponseProto);
rpc getAdditionalDatanode(GetAdditionalDatanodeRequestProto)
    returns (GetAdditionalDatanodeResponseProto);
rpc complete(CompleteRequestProto) returns (CompleteResponseProto);
rpc reportBadBlocks(ReportBadBlocksRequestProto)
    returns (ReportBadBlocksResponseProto);
rpc concat(ConcatRequestProto) returns (ConcatResponseProto);
rpc truncate(TruncateRequestProto) returns (TruncateResponseProto);
rpc rename(RenameRequestProto) returns (RenameResponseProto);
rpc rename2(Rename2RequestProto) returns (Rename2ResponseProto);
rpc delete(DeleteRequestProto) returns (DeleteResponseProto);
rpc mkdirs(MkdirsRequestProto) returns (MkdirsResponseProto);
rpc getListing(GetListingRequestProto) returns (GetListingResponseProto);
rpc renewLease(RenewLeaseRequestProto) returns (RenewLeaseResponseProto);
```

```

/**
 * A client that wants to write an additional block to the
 * indicated filename (which must currently be open for writing)
 * should call addBlock().
 *
 * addBlock() allocates a new block and datanodes the block data
 * should be replicated to.
 *
 * addBlock() also commits the previous block by reporting
 * to the name-node the actual generation stamp and the length
 * of the block that the client has transmitted to data-nodes.
 *
 * @param src the file being created
 * @param clientName the name of the client that adds the block
 * @param previous previous block
 * @param excludeNodes a list of nodes that should not be
 * allocated for the current block
 * @param fileId the id uniquely identifying a file
 * @param favoredNodes the list of nodes where the client wants the blocks.
 * Nodes are identified by either host name or address.
 * @param addBlockFlags flags to advise the behavior of allocating and placing
 * a new block.
 *
 * @return LocatedBlock allocated block information.
 *
 * ...
 */
@Idempotent
LocatedBlock addBlock(String src, String clientName,
    ExtendedBlock previous, DatanodeInfo[] excludeNodes, long fileId,
    String[] favoredNodes, EnumSet<AddBlockFlag> addBlockFlags)
    throws IOException;

```

Client and Datanode communication

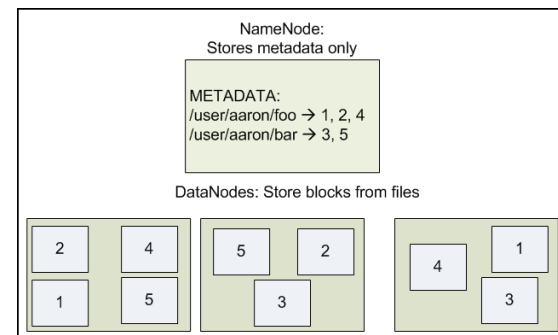
- Source code (version 2.8.1)
 - Definition of protocol
 - datatransfer.proto
 - Located at: <hadoop-src-dir>\hadoop-hdfs-project\hadoop-hdfs-client\src\main\proto
 - Implementation
 - DataTransferProtocol.java
 - <hadoop-src-dir>\hadoop-hdfs-project\hadoop-hdfs-client\src\main\java\org\apache\hadoop\hdfs\protocol\datatransfer

Operations

- readBlock()
- writeBlock()
- copyBlock() – for load balancing
- replaceBlock() – for load balancing
 - Move a block from one DataNode to another

Reading a file

1. Client first contacts NameNode which informs the client of the closest DataNodes storing blocks of the file
 - This is done by making which RPC call?
2. Client contacts the DataNodes directly for reading the blocks
 - Calling readBlock()

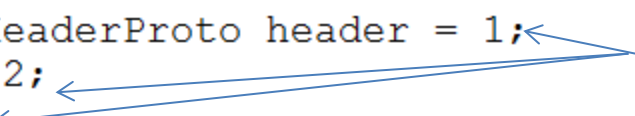


datatransfer.proto

```
message OpReadBlockProto {
  required ClientOperationHeaderProto header = 1;
  required uint64 offset = 2;
  required uint64 len = 3;
  optional bool sendChecksums = 4 [default = true];
  optional CachingStrategyProto cachingStrategy = 5;
}

message ChecksumProto {
  required ChecksumTypeProto type = 1;
  required uint32 bytesPerChecksum = 2;
}

message OpWriteBlockProto {
  required ClientOperationHeaderProto header = 1;
  repeated DatanodeInfoProto targets = 2;
  optional DatanodeInfoProto source = 3;
  enum BlockConstructionStage {
    PIPELINE_SETUP_APPEND = 0;
    // pipeline set up for failed PIPELINE_SETUP_APPEND recovery
    PIPELINE_SETUP_APPEND_RECOVERY = 1;
    // data streaming
```

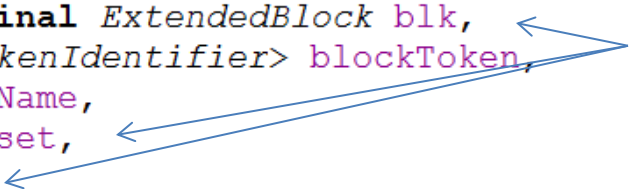


Block, offset, length

DataTransferProtocol.java

```
/**
 * Read a block.
 *
 * @param blk the block being read.
 * @param blockToken security token for accessing the block.
 * @param clientName client's name.
 * @param blockOffset offset of the block.
 * @param length maximum number of bytes for this read.
 * @param sendChecksum if false, the DN should skip reading and sending
 *        checksums
 * @param cachingStrategy The caching strategy to use.
 */
public void readBlock(final ExtendedBlock blk,
    final Token<BlockTokenIdentifier> blockToken,
    final String clientName,
    final long blockOffset,
    final long length,
    final boolean sendChecksum,
    final CachingStrategy cachingStrategy) throws IOException;

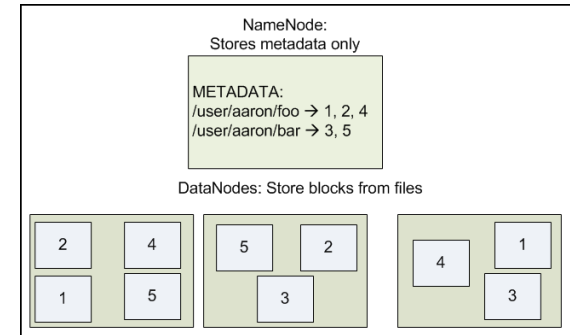
/**
 * Write a block to a datanode pipeline.
 * The receiver datanode of this call is the next datanode in the pipeline.
 * The other downstream datanodes are specified by the targets parameter.
 * Note that the receiver {@link DatanodeInfo} is not required in the
 * parameter list since the receiver datanode knows its info. However, the
 * {@link StorageType} for storing the replica in the receiver datanode is a
 * parameter since the receiver datanode may support multiple storage types.
 */
```



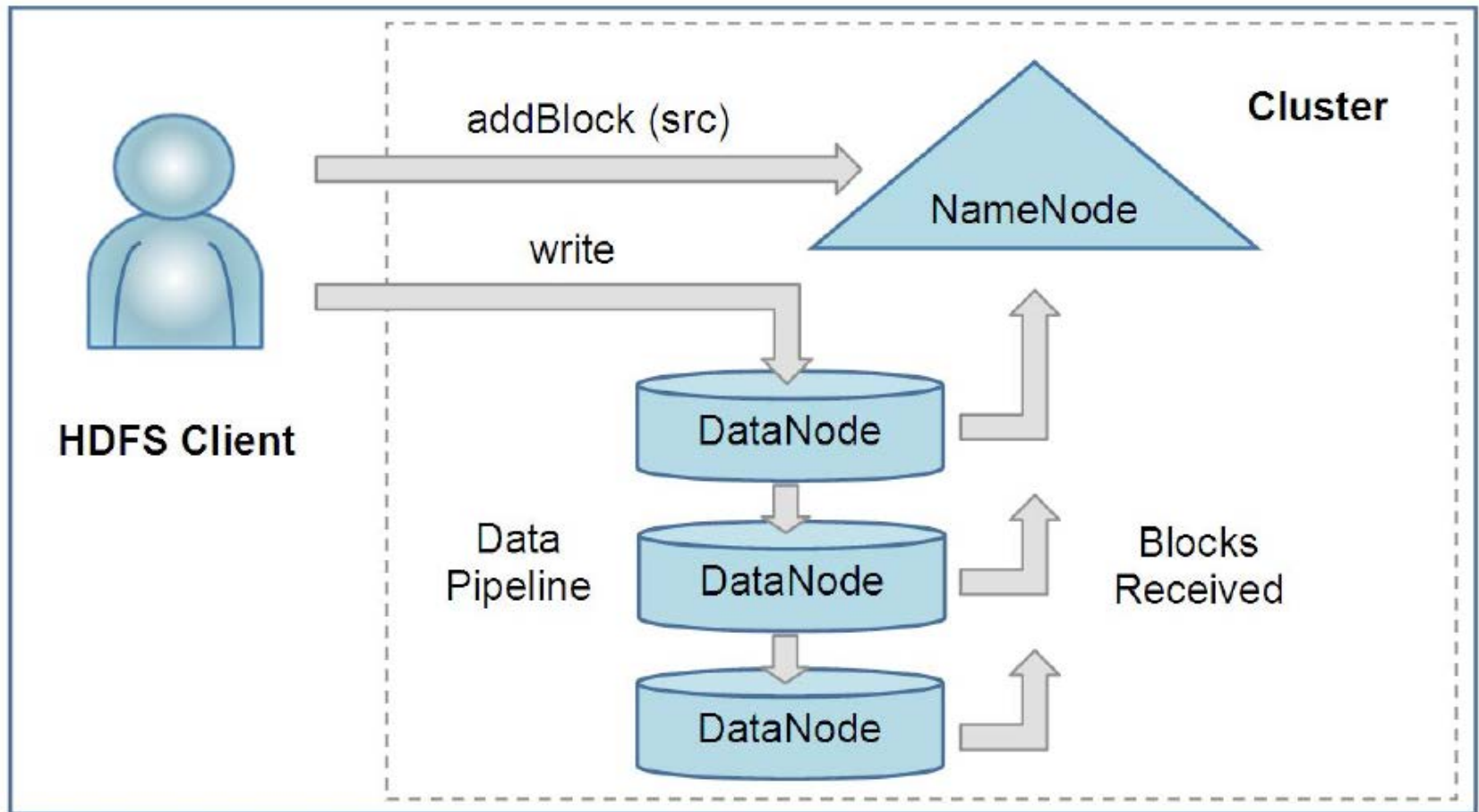
A diagram consisting of three blue arrows pointing from a red text label to specific parameters in the `readBlock` method signature. The label "Block, offset, length" is in red. The first arrow points from the label to the `blk` parameter. The second arrow points from the label to the `blockOffset` parameter. The third arrow points from the label to the `length` parameter.

Writing a file

- Blocks are written one at a time
 - In a pipelined fashion through the data nodes
- For each block:
 - Client asks NameNode to select DataNodes for holding its replica (**using which rpc call?**)
 - e.g., DataNodes 1 and 3 for the first block of /user/aaron/foo
 - It then forms the pipeline to send the block



Writing a file



```

/**
 * Write a block to a datanode pipeline.
 * The receiver datanode of this call is the next datanode in the pipeline.
 * The other downstream datanodes are specified by the targets parameter.
 * Note that the receiver {@link DatanodeInfo} is not required in the
 * parameter list since the receiver datanode knows its info. However, the
 * {@link StorageType} for storing the replica in the receiver datanode is a
 * parameter since the receiver datanode may support multiple storage types.
 *
 * @param blk the block being written.
 * @param storageType for storing the replica in the receiver datanode.
 * @param blockToken security token for accessing the block.
 * @param clientName client's name.
 * @param targets other downstream datanodes in the pipeline.
 * @param targetStorageTypes target {@link StorageType}s corresponding
 * to the target datanodes.
 * @param source source datanode.
 * @param stage pipeline stage.
 * @param pipelineSize the size of the pipeline.
 * @param minBytesRcvd minimum number of bytes received.
 * @param maxBytesRcvd maximum number of bytes received.
 * @param latestGenerationStamp the latest generation stamp of the block.
 * @param pinning whether to pin the block, so Balancer won't move it.
 * @param targetPinnings whether to pin the block on target datanode
 */

```

```

void writeBlock(final ExtendedBlock blk, → Block to be written
final StorageType storageType,
final Token<BlockTokenIdentifier> blockToken, → Rest of data nodes
final String clientName,
final DatanodeInfo[] targets,
final StorageType[] targetStorageTypes,
final DatanodeInfo source, → Current data node in the pipeline
final BlockConstructionStage stage,
final int pipelineSize,
final long minBytesRcvd,
final long maxBytesRcvd,

```

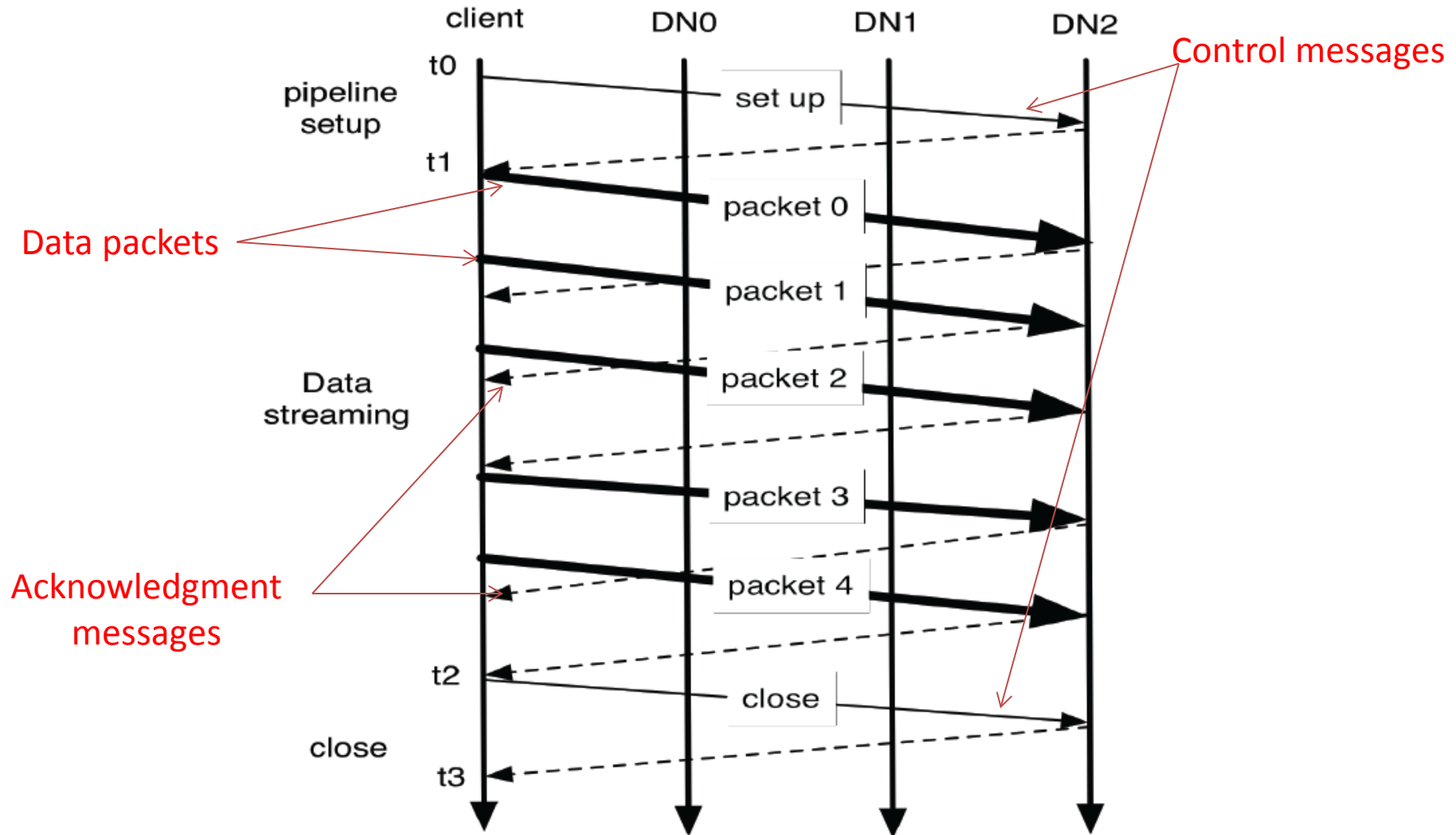
Data pipelining

- Consider a block X to be written to DataNode A, B, and C (replication factor = 3)
 1. X is broken down into packets (typically 64KB/packet)
 2. Client sends the packet to DataNode A
 3. A sends it further to B & B further to C

Acknowledgement

- C acknowledges to B
 - B to A
 - And finally A to client
- All send acknowledgements to Namenode
 - NameNode will update the metadata for the file
 - Reflecting that a new block has been added to the file

Data pipelining for writing blocks



Acknowledgement

- Client does not wait for the acknowledgement of previous packet before sending next one
- Is this synchronous or asynchronous?
- Advantage?

Roadmap

- Hadoop architecture
 - HDFS
 - MapReduce
- Installing Hadoop & HDFS



Hadoop installation

- Install the Hadoop package
 - Log into your EC2 instance and then execute:
 - `wget`
<http://archive.apache.org/dist/hadoop/common/hadoop-3.1.2/hadoop-3.1.2.tar.gz>
 - `tar xvf hadoop-3.1.2.tar.gz`
- Might want to remove installation package (~200MB) to save space

Install java sdk

- `sudo yum install java-1.8.0-devel`
 - Java 1.8 is needed for spark


Setup environment variables

- Edit ~/.bashrc by adding the following:
 - export JAVA_HOME=/usr/lib/jvm/java
 - export HADOOP_CLASSPATH=\${JAVA_HOME}/lib/tools.jar
 - export HADOOP_HOME=/home/ec2-user/hadoop-3.1.2
 - export
PATH=\${JAVA_HOME}/bin:\${HADOOP_HOME}/bin:\${PATH}
- source ~/.bashrc
 - This is to get the new variables in effect
 - Or you may also log out and log in again

Set up pseudo-distributed mode

- Edit <your hadoop installation directory>/etc/hadoop/core-site.xml by adding the following property:
 - <configuration>
 - <property>
 - <name>fs.defaultFS</name>
 - <value>hdfs://localhost:9000</value>
 - </property>
 - </configuration>
- hdfs://localhost:9000 will be the URI for root of hdfs

Pseudo-distributed mode

- Edit etc/hadoop/hdfs-site.xml, add this:
 - <configuration>
 - <property> 
 - <name>dfs.replication</name>
 - <value>1</value>
 - </property>
 - </configuration>
- dfs.replication = 1 (replication factor)

Setup passphraseless ssh

- Reason:
 - So that Hadoop can automatically start the DataNode daemons on machines running the daemons
- Note that DataNode is running on localhost in our setup
 - So all daemons run on the same host

Setup passphraseless ssh

-P specifies passphrase: here is an empty string

- `ssh-keygen -t rsa -P "" -f ~/.ssh/id_rsa`
 - This generates public/private key pairs
 - `id_rsa` is the private key; `id_rsa.pub` public key
- `cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys`
 - Add public key into the list of authorized keys
- `chmod 0600 ~/.ssh/authorized_keys`
 - Change the file permission properly

Check if it works

- ssh localhost
 - It should login to localhost without asking for password (may need to confirm yes first time)
- exit
 - Make sure you exit from "ssh localhost"

```
[ec2-user@ip-172-31-24-7 hadoop]$ ssh localhost
Last login: Sat Jan 28 19:14:57 2017 from 75-140-79-227.dhcp.mtpk.ca.charter.com

  _|_  ( _|_ /)
 _|_  \ _|_ | _|_  Amazon Linux AMI

https://aws.amazon.com/amazon-linux-ami/2016.09-release-notes/
[ec2-user@ip-172-31-24-7 ~]$ |
```


Formatting hdfs & starting hdfs

- bin/hdfs namenode -format
- sbin/start-dfs.sh
 - sbin/stop-dfs.sh to stop it

```
[ec2-user@ip-172-31-52-194 hadoop-2.7.3]$ sbin/start-dfs.sh
Starting namenodes on [localhost]
localhost: starting namenode, logging to /home/ec2-user/hadoop-2.7.3/logs/hadoop-ec2-user-namenode-ip-172-31-52-194.out
localhost: starting datanode, logging to /home/ec2-user/hadoop-2.7.3/logs/hadoop-ec2-user-datanode-ip-172-31-52-194.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /home/ec2-user/hadoop-2.7.3/logs/hadoop-ec2-user-secondarynamenode-ip-172-31-52-194.out
[ec2-user@ip-172-31-52-194 hadoop-2.7.3]$ jps
30298 DataNode
30164 NameNode
30468 SecondaryNameNode
30577 Jps
[ec2-user@ip-172-31-52-194 hadoop-2.7.3]$
```

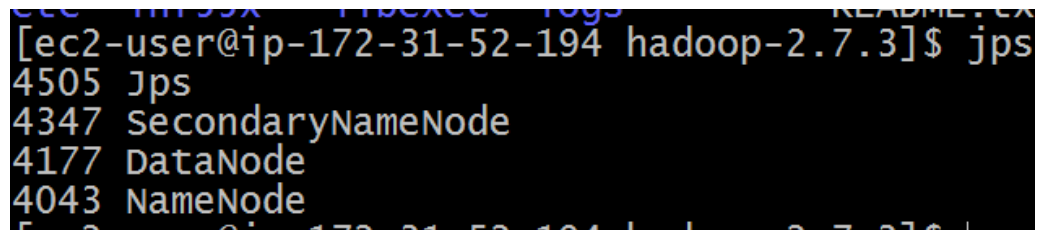
Verifying HDFS is started properly

- Execute jps, you should see 3 java processes:

- SecondaryNameNode

- DataNode

- NameNode

A terminal window screenshot showing the output of the 'jps' command. The prompt is '[ec2-user@ip-172-31-52-194 ~]\$ jps'. The output lists four processes: '4505 Jps', '4347 SecondaryNameNode', '4177 DataNode', and '4043 NameNode'.

```
[ec2-user@ip-172-31-52-194 ~]$ jps
4505 Jps
4347 SecondaryNameNode
4177 DataNode
4043 NameNode
```

- If NameNode is not started
 - Try to stop hdfs & reformat namenode (see previous slide)

Working with hdfs

- Setting up home directory in hdfs
 - `bin/hdfs dfs -mkdir /user`
 - `bin/hdfs dfs -mkdir /user/ec2-user`
(ec2-user is user name of your EC2 account)
- Create a directory "input" under home
 - `bin/hdfs dfs -mkdir /user/ec2-user/input`
 - Or simply:
 - `bin/hdfs dfs -mkdir input`

 This will automatically create the "input" directory under /user/ec2-user

Working with hdfs

- Copy data from local file system
 - `bin/hdfs dfs -put etc/hadoop/*.xml /user/ec2-user/input`
 - Ignore error if you see one like this: "WARN hdfs. DataStreamer: Caught exception..."
- List the content of directory
 - `bin/hdfs dfs -ls /user/ec2-user/input`

Working with hdfs

- Copy data from hdfs
 - `bin/hdfs dfs -get /user/ec2-user/input input1`
 - If `input1` does not exist, it will create one
 - If it does, it will create another one under it
- Examine the content of file in hdfs
 - `bin/hdfs dfs -cat /user/ec2-user/input/core-site.xml`

Working with hdfs

- Remove files
 - `bin/hdfs dfs -rm /user/ec2-user/input/core-site.xml`
 - `bin/hdfs dfs -rm /user/ec2-user/input/*`
- Remove directory
 - `bin/hdfs dfs -rmdir /user/ec2-user/input`
 - Directory "input" needs to be empty first

Where is hdfs located?

- /tmp/hadoop-ec2-user/dfs/

```
[ec2-user@ip-172-31-52-194 data]$ pwd
/tmp/hadoop-ec2-user/dfs/data
[ec2-user@ip-172-31-52-194 data]$ cd ..
[ec2-user@ip-172-31-52-194 dfs]$ ls
data  name  namesecondary
[ec2-user@ip-172-31-52-194 dfs]$ ls data
current  in_use.lock
[ec2-user@ip-172-31-52-194 dfs]$ ls name
current  in_use.lock
[ec2-user@ip-172-31-52-194 dfs]$ ls namesecondary/
current  in_use.lock
[ec2-user@ip-172-31-52-194 dfs]$ |
```

References

- K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "[The hadoop distributed file system](#)," in Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on, 2010, pp. 1-10.