The Hadoop Distribuited Filesystem

William Lucia

Introduction

- When a dataset outgrows the storage capacity of a single physical machine, it becames necessary to partition it across a number of separate machine.
- Filesystem that manage the storage across the network of machiens are called *distribuited filesystem.*
- Hadoop comes with a distribuited Filesystem called HDFS.

The Design of HDFS

- HDFS is a filesystem designed for storing very large files with streaming data access pattern, running on clusters of commodity hardware.

- HDFS is built around the idea that the most efficient data processing pattern is a *write-once, read-many-times pattern.*

The Design of HDFS

- Various analysis are performed on a dataset.
- Each analysis will involve a large portion, if not all, of the dataset , so the time to read the whole dataset is more important than the latency in reading the first record.

HDFS does not work for...

There are areas where HDFS in not a good fit today:

- *Low-latency data access*
- *Lots of small files*
- *Multiple writer, arbitrary file modifications*

Blocks

- A disk has a block size, which is the minimum amount of data that it can be read or write.
- Filesystems for a single disk build on this by dealing with data in blocks, which are an integral multiple of the disk block size.
- HDFS, too, has a concept of a block, but is a much larger unit (64 MB by default). The reason is to minimize the cost of a seeks.

General purpose filesystem abstraction

Having a block abstraction for distribuited filesystem brings several benefits:

- A file can be larger than any single disk in the network. There's nothing that requires the blocks from a file to be stored on the same disk.
- Making the unit of abstraction a block rather than a file simplifies the storage subsystem.
- Block fit well with replication for providing fault tolerance and availability.

Namenodes and Datanodes

An HDFS clustes has two types of node operating in a *master-worker* pattern:

- A *namenode* (the master).
- A number of *datanodes* (workers).

Namenode

- Manages the filesytem namespace.

- It maintains the filesystem tree and the metadata for all the files and directory in the tree. These information are stored persistently on the local disk in two files: the *namespace image* and the *edit log.*

- The namenode also knows the datanodes on which all the blocks for a given file are located. It doesn't store block location persistently.

Datanodes

- They store and retrieve blocks when they are told to (by clients or the namenode).
- They report back to the namenode periodically with lists of blocks that they are storing.

Namenode resilient to failure

- Without the *namenode*, the filesystem cannot be used.
- If namenode fails, all the files on the filesystem would be lost since there would be no way of knowing how to reconstruct the files from the blocks on the *datanodes*.

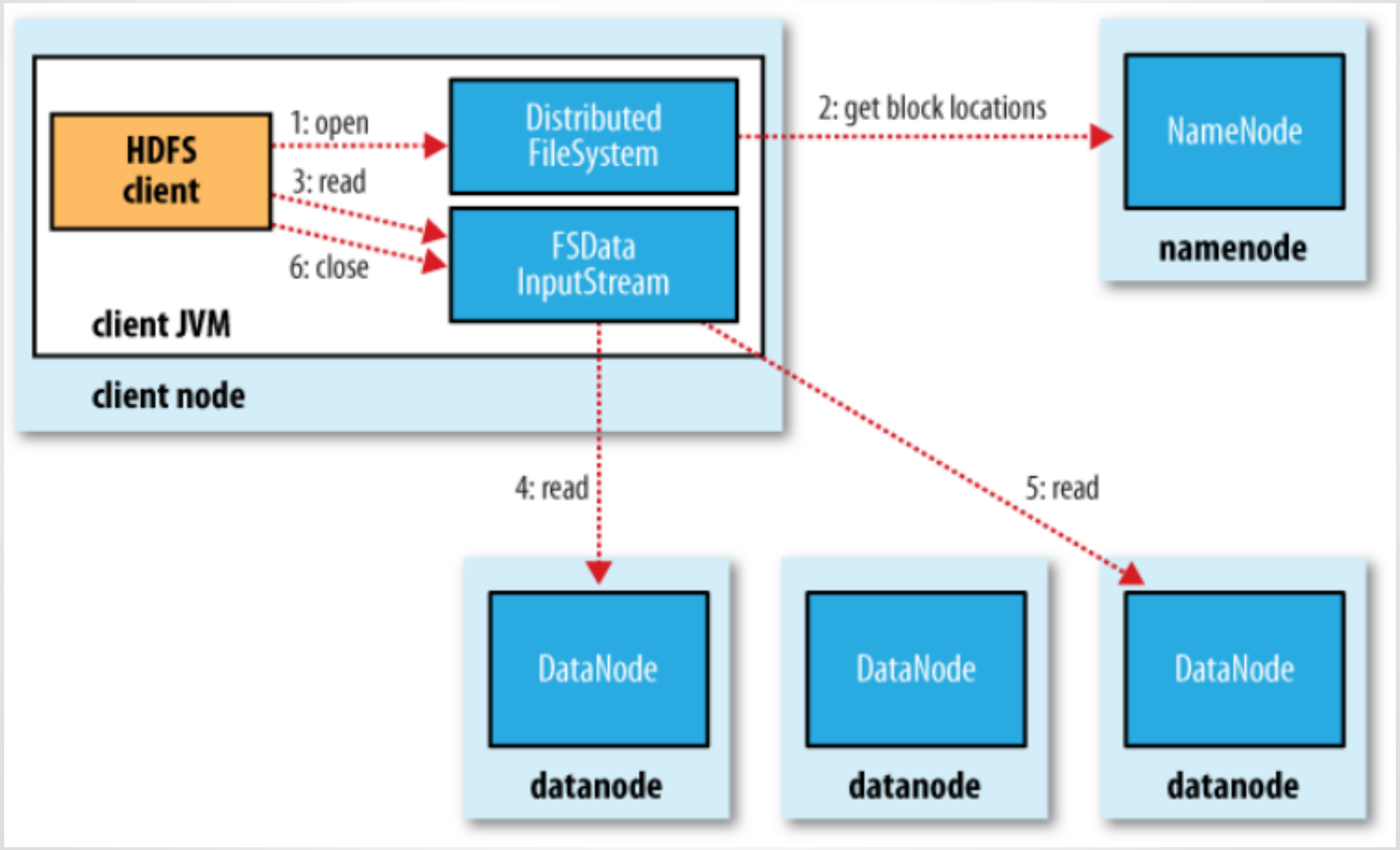
Namenode resilient to failure

It's important to make the namenodes resilient to failures, and Hadoop provides two mechanisms for this:

1. Back up the files that make up the persistent state of the filesystem metadata. Namenode can writes its persistent state to multiple filesystem.

2. Run a *secondary namenode***.** Its main role is to periodically merge the namespace image with the edit log to prevent the edit log from becoming too large. The merged namespace can be used in the event of namenode failing.

Data Flow
Anatomy of a File Read



Anatomy of a File Read

1. The client open the file it wishes to read by calling *open()* method.
2. *DistribuitedFileSystem* calls the namenode, using RPC, to determine the locations of the blocks.
3. The client calls *read()* on the stream.
4. Data is streamed from the *datanode* back to the client, which calls *read()* repeatedly on the stream.
5. When the end of one block is reached, *DFSInputStream* will close the connection with *datanode,* then find the best datanode for the next block.
6. When the client has finished reading, it calls *close().*

Anatomy of a File Read

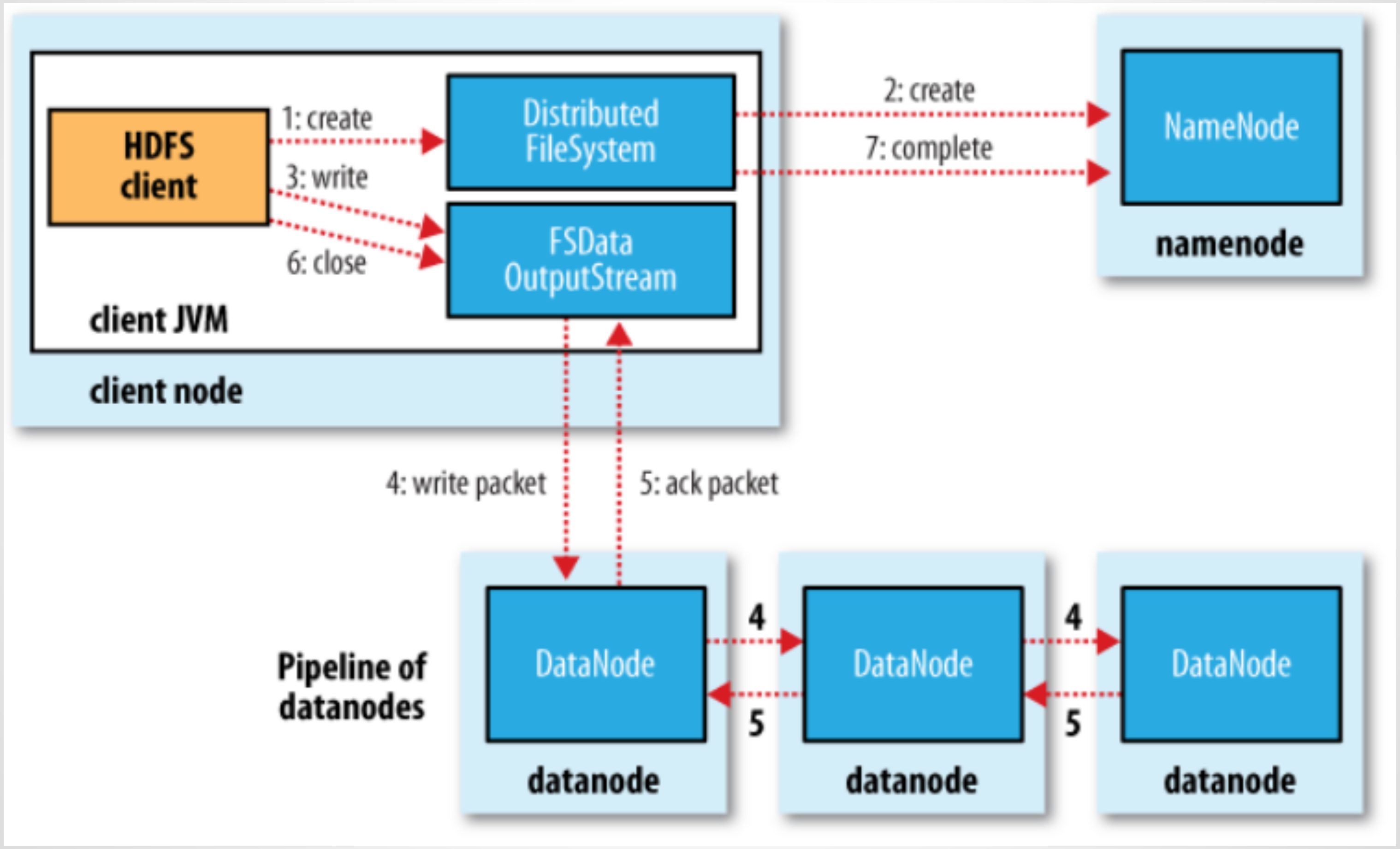
During reading, if the *DFSInputStream* encounters an error while communicating with a *datanode:*

1. It will try the next closest one for that block.

2. It will remember *datanodes* that have failed so that it doesn't needlessly retry them for later blocks.

3. *DFSInputStream* also verifies checksums for the data transferred to it from *datanode.* If a corrupted block is found, it is report to the *namenode* before the *DFSInputStream* attempts to read a replica of the block from another *datanode.*

Data Flow
Anatomy of a File Write



Anatomy of a File Write

1. The Client creates the file by calling *create()*.
2. *DistribuitedFileSystem* makes an RPC call to *namenode* to create new file in the filesystem's namespace, with no blocks associated with it.
3. As the client writes data, *DFSOutputStream* splits it into packets, which it writes to an internal queue, called *data queue*.
4. The *data queue* is consumed by the *DataStreamer*, whose responsability it is to ask the *namenode* to allocate new blocks...

Anatomy of a File Write

4. ...*namenode* picks a list of suitable datanodes to store the replicas.The list of datanodes forms a pipeline. The *DataStreamer* streams the packets to the first *datanode* in the pipeline, which stores the packet and forwards it to the second *datanode* in pipeline, and so on.

5. *DFSOutputStream* also mantain an internal queue of packets that are waiting to be aknowledged by datanodes, called the *ack queue.*

Anatomy of File Write

6. When the client has finished writing data, it calls *close().*

7. This action flushes all the remaining packets to the *datanode* pipeline and waits for acknowledgements before contacting the namenode to signal that the file is complete.

Anatomy of a File Write

If a *datanode* fails while data is being written to it, the the following action are taken:

1. Any packets in the *ack queue* are added to the front of *data queue*.
2. The failed *datanode* is removed from pipeline and the remainder of the block's data is written to the two good *datanodes* in the pipeline.
3. The namenode notice that the block is under-replicated, and it arranges for a further replica to be created on another node.
4. Subsequent blocks are then treated as normal.