Writing and Reading

This lesson explains the write and read pipelines in HDFS.

We'll cover the following

Writing and Reading

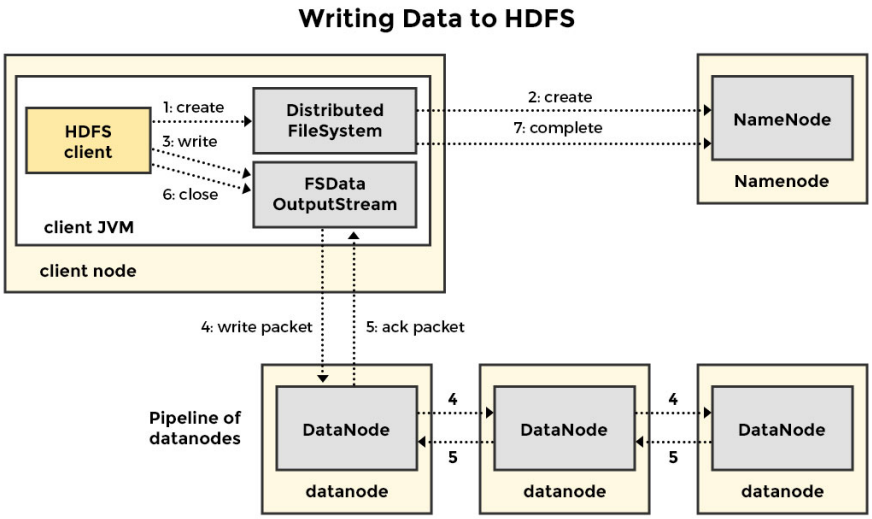
We'll now study the interactions between a client application and HDFS when reading or writing files.

Write path

A client initiates the write process. A client could be an application using the Java API or a person working with the hdfs command line utility. The flow of this interaction between client and HDFS goes like this:

- A client buffers data on the local disk initially. It waits for one HDFS-block worth of data to accumulate before contacting the Namenode.
- The Namenode, once contacted by the client, verifies if the file exists and that the client has the required permissions to create that file. If these checks pass, the Namenode makes a corresponding change in its namespace. It then returns the client a list of DataNodes to write to. These DataNodes host the blocks (and their replicas) that make up the file.

- Upon receiving the list from the Namenode, the client starts writing to the first DataNode.
- That first DataNode receives data from the client in portions. It receives the first portion, writes it to its local repository, and then starts transferring that portion to the second DataNode in the list.
- The second DataNode receives data from the first, writes to its local repository and starts transferring that portion to the third DataNode in the list.
- A pipeline of data transfer is formed from the client to all the involved DataNodes. A DataNode can, at the same time, receive and transfer data.



Read path

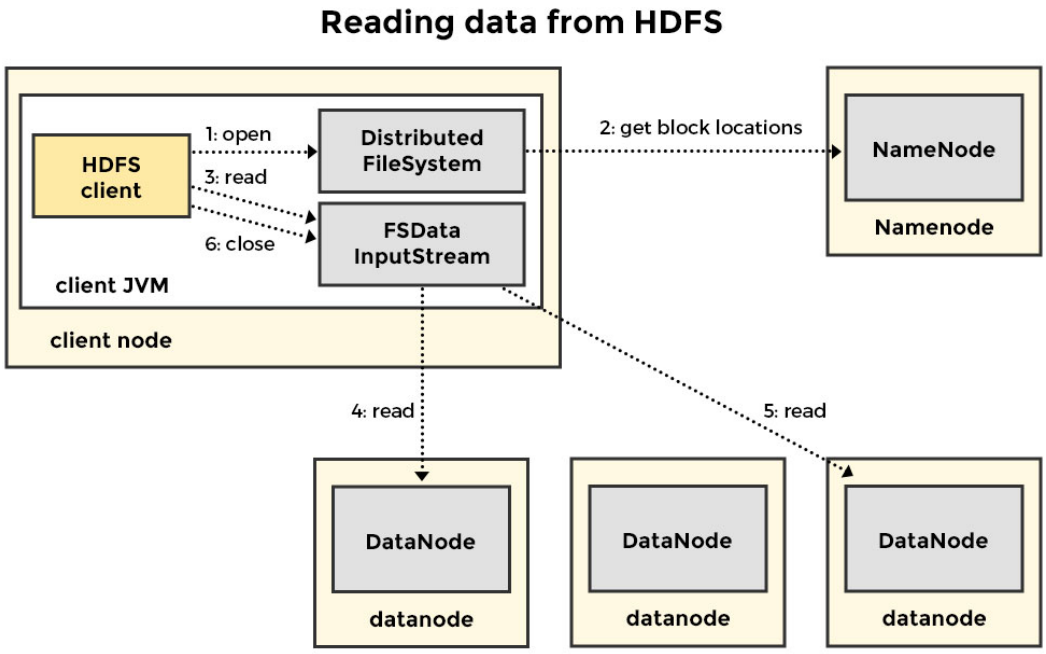
- The read interaction starts when a client makes an RPC call to the Namenode in order to retrieve the location of the first few blocks of a file the client wants to read.
- The Namenode responds with a list of Datanode addresses for each requested block. The list returned for each block consists of Datanodes with a copy of the block. The list of Datanodes is sorted by proximity to the client. If the client, like a Map task, runs on a

Datanode also hosting a copy of the data block, then t[illegible] the block is read.

- If a local copy of the data block is not available or the client is running outside of the cluster, then the client connects to the nearest Datanode to retrieve the first block in the file. Data is streamed from the Datanode to the client until the end of the block is reached.
- The process is repeated. The client connects to the closest Datanode for retrieving the next block of data. The file blocks are read in order.
- After the first batch of data blocks is read, the client requests the Namenode for the next batch of data blocks and reads them in the same way.
- The entire experience for the client application is seamless. From its perspective, the client application is reading a continuous stream.
- Note that by *client* we mean an application like a program that internally invokes Hadoop provided developer Java APIs for communicating with HDFS. The user is not responsible for making network calls to the Namenode or sorting the list of Datanodes by least distance to the client. Hadoop libraries hide this complexity from the user. Under the hood, specific Java classes are responsible for executing the various steps described. We skipped naming them in order to keep the explanation simple.
- If an error occurs when communicating with a Datanode, the next closest Datanode hosting the copy of the data block is tried. The failed Datanode is remembered so that it does not retrieve any blocks in future.
- The received blocks are tested for corruption by computing their checksums. If the checksum does not match, a replica of the same block is read from a different Datanode. The Namenode is also informed of the corrupted block to take remedial action.
- Note that the Namenode is only responsible for servicing requests for Datanode locations. The actual data blocks are never routed via the Namenode to the client. Once the client receives the addresses of th

Datanodes, it talks directly to one of Datanodes to ret[illegible] without involving the Namenode. This prevents the Namenode from becoming a bottleneck and allows HDFS to scale to a large number of clients.

Below is a picture how data is read in HDFS.



Back
Datanode
Next
High Availability
Mark as Completed
Report an Issue