Hadoop File System Design

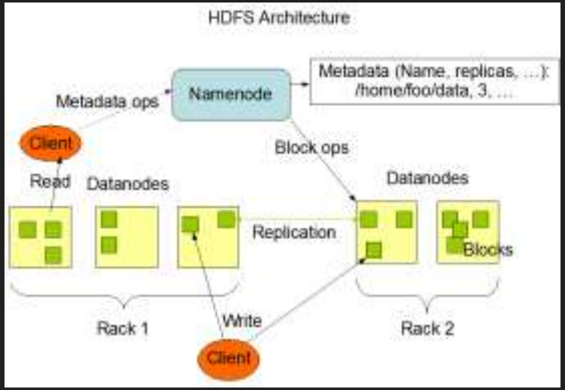
Background

Distributed file systems are a key component of many large scale distributed systems in modern day technology companies. HDFS is the most popular open source file system, and is modeled after the Google File System (GFS). Because HDFS can be run on commodity hardware, it is immensely popular.

Not only is HDFS important for enabling large scale batch processing via MapReduce or dataflow engines like Spark or Tez, but additionally HDFS acts as a building block upon which many open source databases have been created. HDFS is designed with computation in mind, opting to try and optimize for doing any compute close to the data.

Overview

- HDFS is designed for the use case of being able to write a file once, and then read it many times over
- Files are stored in chunks across nodes in the cluster (typically around 128 mb per chunk) to improve parallelism of writing and reading
- Chunks are replicated to ensure data availability



NameNode

- NameNode stores all metadata regarding files, as well as keeping track of all of the chunks (and their version numbers) that comprise the file, and the data nodes on which they are located
- Keeps all metadata in memory
    - All changes to file system metadata go to the EditLog (like a write ahead log)
        - Sequential writes are faster
    - State occasionally checkpointed to disk on a persistent FSImage file
        - This is when edit log writes are applied to FSImage file
    - On startup use the combination of FSImage and EditLog to load state into memory

NameNode continued

NameNode does not keep location of all chunks on disk, only in memory.

When it first boots, it goes into safe mode:

- Receives a block report from each data node saying which chunks it holds
- NameNode then uses this to construct local state in memory
- If there are chunks that are not stored on enough replicas, the NameNode will automatically replicate them until the replication threshold is reached
    - This will also happen if NameNode assumes certain data nodes to be down due to a lack of heartbeats

Hadoop Replication

“Rack Aware” Replication:

- Replicate chunks in a way that reduces latency for clients as well as reducing the possibility of all replicas going down in the event of a rack/data center failure
  - For replication factor of 3: one replica in the same rack as the writer, two replicas on the same remote rack (does this to reduce network bandwidth)

Hadoop Replication Continued

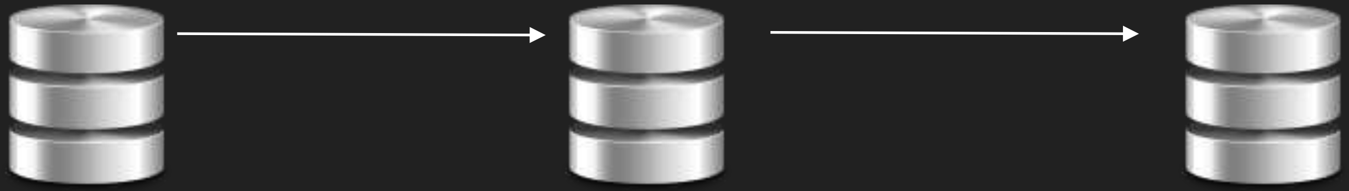
Replication Pipelining:

- Data pipelined from one replica node to the next
- Writes are only considered successful if all replicas in pipeline acknowledge it, otherwise client is expected to retry the write until successful
    - While in theory this leads to strong consistency, the reality is that it can lead to inconsistencies in the datanodes

Hadoop Replication Continued

Replication Pipelining:

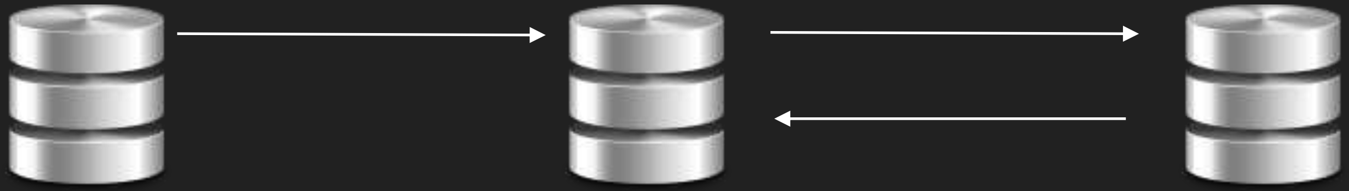
- Data pipelined from one replica node to the next
- Writes are only considered successful if all replicas in pipeline acknowledge it, otherwise client is expected to retry the write until successful
    - While in theory this leads to strong consistency, the reality is that it can lead to inconsistencies in the datanodes



Hadoop Replication Continued

Replication Pipelining:

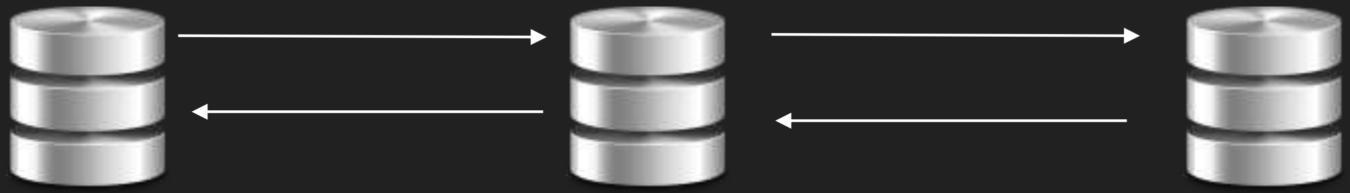
- Data pipelined from one replica node to the next
- Writes are only considered successful if all replicas in pipeline acknowledge it, otherwise client is expected to retry the write until successful
    - While in theory this leads to strong consistency, the reality is that it can lead to inconsistencies in the datanodes



Hadoop Replication Continued

Replication Pipelining:

- Data pipelined from one replica node to the next
- Writes are only considered successful if all replicas in pipeline acknowledge it, otherwise client is expected to retry the write until successful
    - While in theory this leads to strong consistency, the reality is that it can lead to inconsistencies in the datanodes



Hadoop Reads

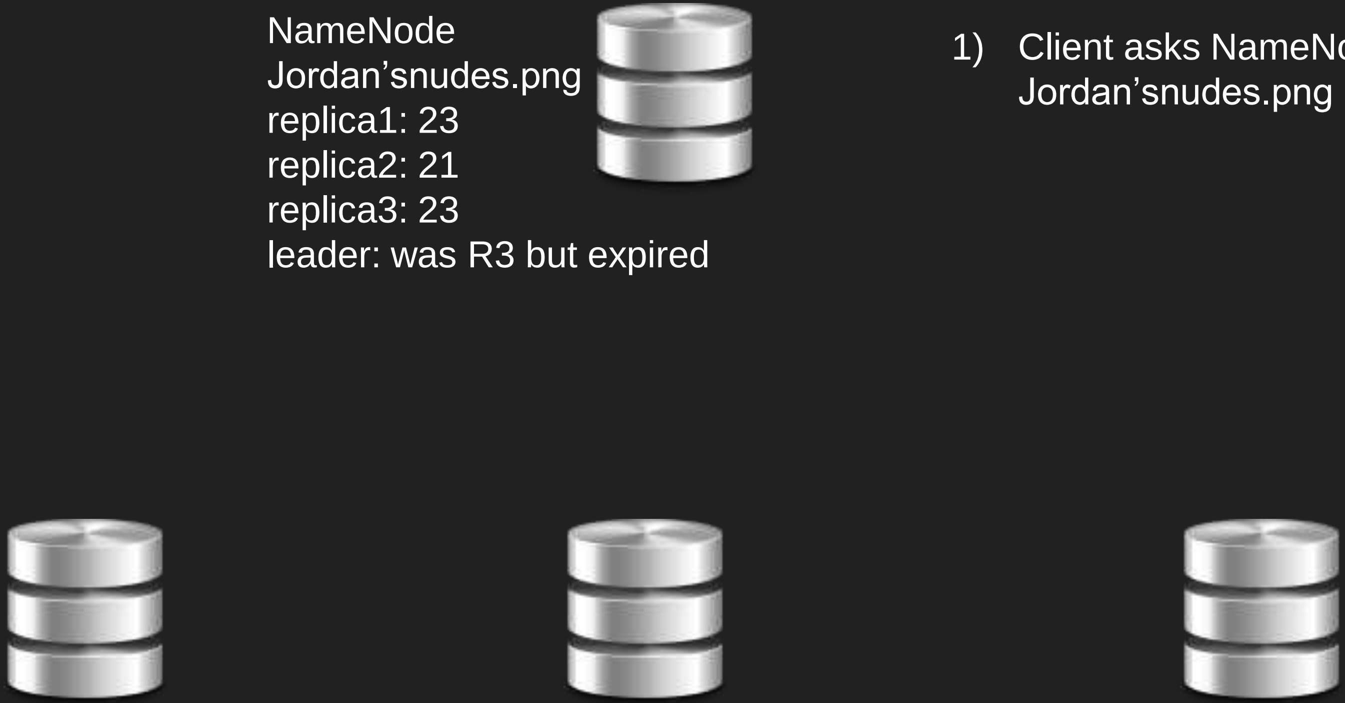
- Client queries master to get list of data nodes carrying a specific chunk
- Client figures out which data node is closest to it, and caches this result for subsequent reads of the chunk
- Client performs the read from the proper data node

Hadoop Writes

- In order to append to a file, first contact the NameNode to see the data nodes on which the chunks are located and pick a primary replica
  - If there is already a primary replica designated for the chunk (its lease is still valid), perform the write to the primary replica
  - Otherwise, we need to pick a primary replica, which must be one of the datanodes with an up to date (same version number as the NameNode) version of the file
  - If this does not exist, we have lost data
- Once the primary replica is determined, all other replicas are considered secondary, and the write is performed on the primary replica, moving through the replication pipeline
- Once all replicas acknowledge the write, the client receives a success message

Hadoop Writes Visualized

NameNode
Jordan’snudes.png
replica1: 23
replica2: 21
replica3: 23
leader: was R3 but expired

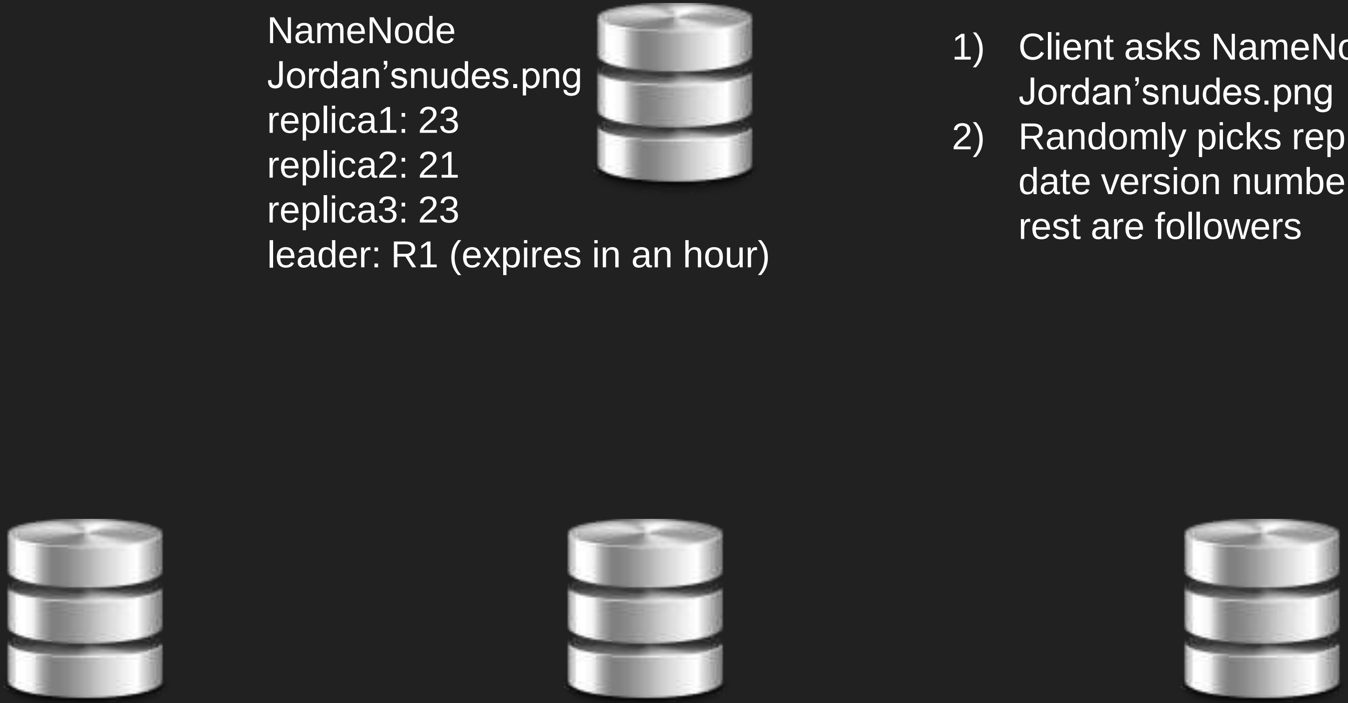


1) Client asks NameNode for data on Jordan’snudes.png

Hadoop Writes Visualized

NameNode
Jordan’snudes.png
replica1: 23
replica2: 21
replica3: 23
leader: R1 (expires in an hour)

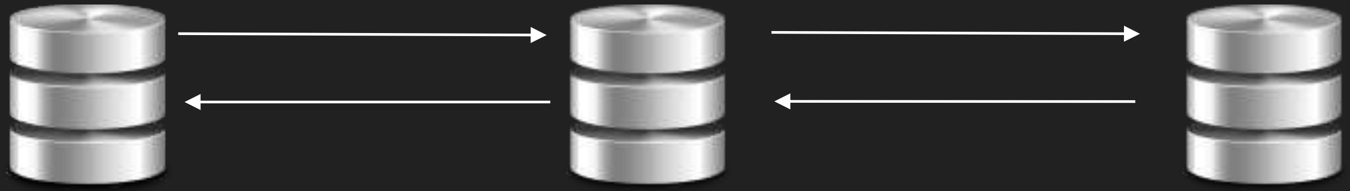
1) Client asks NameNode for data on Jordan’snudes.png
2) Randomly picks replica with up to date version number as leader, rest are followers



Hadoop Writes Visualized

NameNode
Jordan’snudes.png
replica1: 23
replica2: 21
replica3: 23
leader: R1 (expires in an hour)

1) Client asks NameNode for data on Jordan’snudes.png
2) Randomly picks replica with up to date version number as leader, rest are followers
3) R1 is leader, client sends write there and let it propagate through replication pipeline



Issues with Hadoop

In the original GFS paper, and the architecture I have presented so far, there is only one NameNode. If it crashes, we are screwed!

How can we fix this? High Availability HDFS.

High Availability HDFS

Recall that the NameNode keeps an EditLog of all metadata changes, which is really all of its persistent state. Instead of keeping this on the NameNode, we can use a coordination service (in this case ZooKeeper)! Known as quorum journal manager.

By creating a replicated log, we can have a backup NameNode which reads from the replicated log and keeps itself up to date.

HDFS Conclusion

By using a rack aware replication schema as well as rack aware reads, HDFS is able to provide extremely high availability and throughput for both file reads and writes. While the original implementation of HDFS was not at all fault tolerant, the design implementing ZooKeeper completely changes this!

Keep in mind that HDFS is not perfect, and we saw that there can be some data consistency issues as a result of the replication pipeline.

On the whole though, using HDFS for storing data in conjunction with large scale compute is a great option, and we will see how certain databases leverage it.