Distributed File Systems: Hadoop Distributed File System and Google File System

Ciprian Lucaci
ciprian.lucaci@tum.de
Technische Universität München, Germany

Daniel Straub
daniel.straub@tum.de
Technische Universität München, Germany

ABSTRACT

Distributed file systems have been a technology enabler to store and process large files which exceed the size of any drive while working in a distributed and fail tolerant manner. The first and most known implementations are Google File System and Hadoop Distributed File System.

Keywords

Distributed File Systems, Distributed Systems

1. INTRODUCTION

During the late 90s the Y2K issue was a popular topic in media, but together with the 2000s a bigger and more silent issue took its place. The rise of internet traffic, saving and processing every online footprint (e.g. user clicks), having countless sensors pouring out data into databases and then the dawn of the smartphone era meant that thousands of terabytes and petabytes of data were being generated daily. This became a problem due to multiple reasons. Firstly, there was no large enough storage medium to store all the data in a single physical location, and secondly processing such amount of data in order to extract valuable information became an even bigger challenge. Even if there were enough space to store huge amounts of data on a single drive, only seeking through such amount of data would be prohibitive in terms of access times. In this context a new IT domain was born - Big Data.

The requirements of an innovative file system which could cope with such large amounts of data can be traced along the following lines. Firstly, large amounts of data should be stored reliably. Secondly, high bandwidth streaming is a priority (e.g. play a youtube video before it is all loaded locally). Another key aspect is distributing storage across multiple locations and distributed computation. Finally, data besides being stored must be analyzed and transformed in a cost-effective manner.

The proposed solution came from Google where Google File System [6] was introduced alongside the MapReduce algorithm. The central pillars of the proposed solution are the following. First of all, moving computation where data resides means sending the algorithm in a distributed manner where the data is stored, compute partial results and then combine them. The old paradigm was to bring all the data to a central location, e.g. a supercomputer, and there do all the processing. Moving a few lines of code of an algorithm across network and retrieving partial results is much faster than moving the entire data across networks. Secondly, low costs is achieved by using commodity machines. This means that instead of using very expensive and customized hardware, the new system could run on any computer available at any IT store. Thus the new system must have a high fault tolerance because in a cluster of thousands of servers, the failure rate of an average computer can become quite high compared to the old paradigm. Finally, vertical and horizontal scalability provide maintainability and extensibility to the system. Vertical scalability means that a node in the server cluster can be upgraded at any time in terms of hardware components, e.g. RAM, CPU, storage. Horizontal scalability means that a new node can always be automatically added to the system without the need of restarting or changing the running server cluster.

2. HADOOP DISTRIBUTED FILE SYSTEM (HDFS)

Hadoop Distributed File System was first introduced in [1]. It was inspired by the Google File System and it published by Apache Foundation as an open-source system.

Hadoop Distributed File System is part of a larger ecosystem. Together with the MapReduce paradigm it constitutes the Hadoop framework to store and process big data sets. On top of these other frameworks such as PIG, Hive, or Hbase can be added as additional layers for data display, processing, tracing and other features.

2.1 Architecture

Hadoop distributed file system runs on a server cluster at in the user space level of the host operating system. A server cluster can span from only a few nodes to thousands of nodes and a server cluster can be also distributed geographically. As a good practice, it is recommended [1] to have multiple independent Hadoop clusters than only one huge cluster. This section contains a bottom up introduction to HDFS architecture.

2.1.1 Blocks

HDF has many similarities with the file system of the common operating systems. Such a similarity is the use of the concepts of `blocks`. A large file is split into blocks and a HDFS block by default contains at most 64 MB and can be configured also to 128 MB, 256 MB and so on. In this context, large files usually mean a couple hundreds of GB, files and data which is larger than any single drive. Data accepted by HDFS to be stored can be in any format.

Alongside the data, the other important part is the `Metadata` which includes a checksum to ensure data integrity and a block generation timestamp.

2.1.2 Data Nodes

The next important component of the HDFS is the location of the blocks. These are stored on a `Data Node`. Each Data Node usually has around 70% of the available storage capacity allocated for storing blocks. There is no ”perfect” Hadoop configuration for any use case, therefore any configuration number should be taken only as a concrete example for understanding purposes.

HDFS runs on commodity and the failure rate of such a server is quite high, so is down the entire content of that server is then down. Therefore, in order to provide high fault-tolerance, each block is replicated n-times, i.e. `replication factor`, on other data nodes.

Each Data-node issues a `block report` periodically (by default one/hour) which represents all blocks available on that server. Additionally, it issues a `heartbeat` periodically (by default every 3 seconds) providing information about current transfers, storage usage and other useful information for balancing the load of the cluster.

Data nodes are organized into racks and a cluster can contain thousands of data nodes. Data nodes are identified by a unique id which is independent of IP address and port. Also, the location of a data node is rack and network switch aware.

2.1.3 Name Node

On of the most important components is the `Name node`. It interacts with the Data-nodes by the `master-worker pattern`. This means that actions performed by the data-nodes are dispatched by the name-node. Data-nodes are concerned only with storing and retrieving data, while the Name-node is concerned with the organization and location of the data on the data-nodes.

The files can be organized in directories thus forming a tree. A stored element name is composed by the location of block and the block id. The `namespace` represents the file in the HDFS and and the blocks associated to each file. For example *file1* is located in */user/dir1/* and is composed of blocks *node1bk1, node1bk2, node3bk5, node4bk2*. In order to improve the look-up speed of a file and associated blocks, the entire namespace is located in RAM of the NameNode. Therefore it is recommended that the Name node is a machine with increased memory and computing power compared to the average data node. The entries and the namespace are represented by an `inode` which also contains information about permissions, modification and access times.

The most important elements contained in the `metadata` are the edit log, system image and block location. A log of all the changes made to the namespace is called the `edit log`. The `system image` is represented by the edit log and the namespace. `Block location` contains the mapping the mapping of any block id to all the replicas stored on data nodes.

2.1.4 Client

The final core component of a useful distributed file system is the `client`. The HDFS client is a code library written in Java which exposes HDFS interface to read, write, create, delete files or folders.

2.1.5 Secondary Node

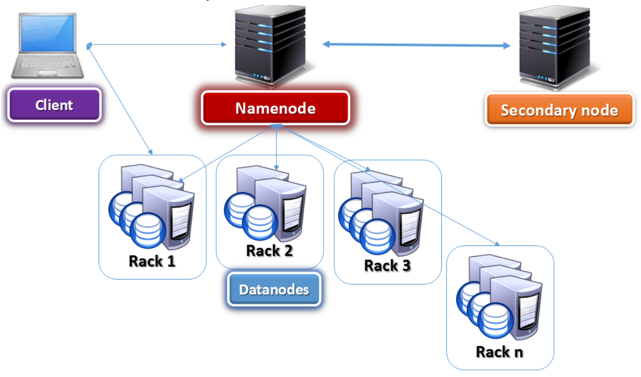

Figure 1: HDFS Architecture

Is there any weakness in the architecture .. add text Checkpoint node Backup node

2.2 Workflow

Single writer, multiple reader model

2.2.1 Startup

1. Handshake - name space id - software version 2. Register - storage id 3. Block report 4. Heartbeat

2.2.2 Write

1. Request block in namespace for file 2. Send block to a Data node 3. Create first replica 4. Create second replica

2.2.3 Read

5. Get list of datanodes hosting file blocks 6. Get blocks in order

2.3 Features

.. add text

2.3.1 Block placement policy

1. No Data node contains more than one replica of any block
2. No rack contains more than two replicas of the same block (if sufficient racks on the cluster)

2.3.2 *Replica management*
@Name node - optimize disk usage - replication factor - rack awareness

2.3.3 *Balancer*
@Name node - disk space usage - move replicas

2.3.4 *Block scanner*
@Datanode

2.4 Purpose
.. add text Optimized for - large files - commodity hardware - streaming - batch processing - multiple reads Not optimized for - big amount of small files - concurrent modification - arbitrary modification - general purpose applications Cross platform - java, thrift, rest - web access, console - opensource Companies using hdfs - linkedin, amazon, new york times, twitter, ebay, facebook, spotify, ibm, yahoo

3. GOOGLE FILE SYSTEM (GFS)
The Google File System, a distributed file system, was implemented by Google to meet Google's needs. It is widely used within their company.

3.1 Purpose
Since the concept of the GFS is the basis of HDFS the purpose of GFS is not much different to the purpose of HDFS. It was designed to store large files, meaning 100 MB and more per file. It should run on commodity hardware, therefore the system has to be fault tolerant. Optimized operations are appending to files and sequential reading of files, which are typical operations when streaming files. The goal of GFS is high data throughput. While HDFS is open source and widely used by different companies, GFS is not and is only used by Google itself.

3.2 Architecture
The GFS consists of three different types of components: the chunkservers, the master and the clients (see figure 2). While chunkservers store the data which is available in the system, the master controlls the data. The clients can access available data through the master.

The GFS divides files into chunks of a fixed size. The default chunk size is 64 MB. Chunks are replicated to increase fault tolerance and load balancing (see 3.4.1). To identify chunks GFS uses a 64 bit long chunk id which is globally unique. [6]

A GFS instance usually consists of many chunkservers which store chunks and one master (see figure 2). A chunkserver knows about its stored chunks and the corresponding chunk ids. In contrast the master knows about all existing files and which chunks are part of a specific file. Furthermore the master knows where the chunks are stored. It also handles access to chunks as explained in 3.3. While the master coordinates access to files, the transmission of data is done directly between a chunkserver and a client. The master is never directly involved in a transmission of data.

3.3 Workflows
The GFS allows different operations. The most important ones are *Read*, *Write* and *Record Append*.

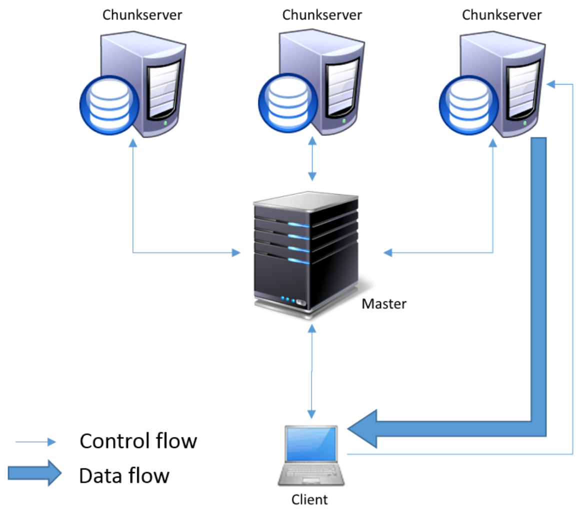

Figure 2: Google File System Architecture

3.3.1 *Read*
Reading is the simplest of the mentioned operations and will be explained with figure 3.

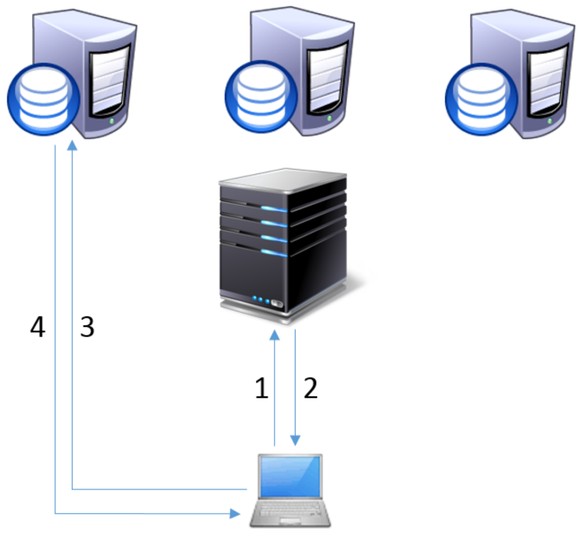

Figure 3: GFS read workflow

To read a file, the client first needs to calculate which part respectively chunk it wants to read. This can be done by the client, because the fixed size of the chunks is known system wide. The client calculates for a specific file the chunk index which corresponds to the byte offset it wants to read (not shown in the figure). Then the client sends a request to the master, containing the name and previously calculated chunk index of the requested file (step 1 in the figure). The master responds with the unique chunk id (see 2) of the requested chunk and its positions within the system, i.e. the chunkservers containing the requested chunks (step 2). In the 3rd step the client chooses one of the chunkservers and sends a request containing the chunk id and the byte range within that chunk. Finally the chunkserver sends the requested data to the client.

3.3.2 *Write*

Writing to a file is a bit more complex than the *Read* operation. The *Write* operation can be defined as modifying a specific byte range within a file.

Leases

GFS uses the mechanism of leases for writing to files. This is done to ensure a consitent mutation order for all replicas of a chunk. A chunkserver may have leases for chunks, but only one chunkserver may have the lease for a specific chunk at any time. The chunkserver which has the lease for a chunk is called *primary*, the others are called *secondaries*. Leases can be requested by chunkservers and are granted by the master.

The following steps explain the execution of a *Write* operation.

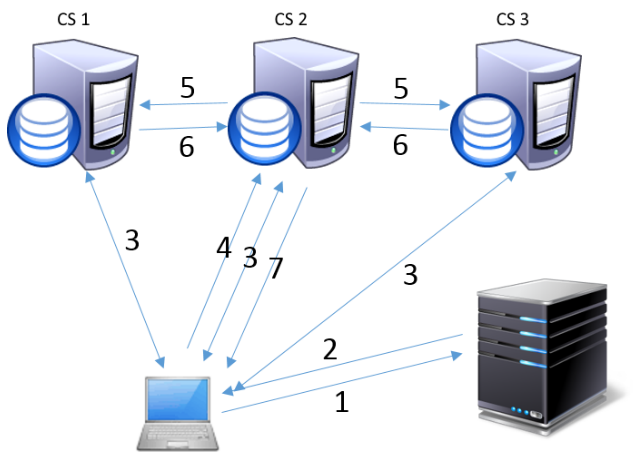

Figure 4: GFS write workflow

Assume the following scenario with reference to figure 4:

There are three chunkservers (CS), all of them containing the chunk x where x is the unique identifier of the chunk. CS2 is the primary. The master knows that there is a specific file (called file1) which is divided into different chunks with the first chunk being chunk x. It also knows that chunk x is available on the mentioned CSs CS1, CS2 and CS3. The client wants to write to the first byte of file1.

1. Like in the read operation the client's first step is to calculate the chunk index it wants to write to. With the intention to write the first byte of file x the client calculates the chunk index which is 0 (the first chunk of the file). Now the client sends a request to the master containing file name and chunk index (file1, 0).

2. The master replies with the chunk id, the primary and the secondaries (x, **CS2**, CS1, CS3). If at this point no CS has the lease for requested chunk, the master grants the lease to one of the chunks before answering the clients request.

3. With that information the client pushes the data which it wants to write to all replicas (the primary and the secondaries). All receiving CSs write the data to its buffer cache and respond with an acknowledgement for receiving the data.

4. After all CSs have acknowledged receiving the data, the client sends a write request to the primary. At this point the primary applies the requested write to itself.

5. Then the primary forwards the write request to all secondaries leading them to apply the changes in the same order.

6. On successfull writing the data the secondaries will indicate the completion of the write with a response to the primary.

7. At last the primary returns the result of the write to the client. If there was any error, the client will retry the write from step 3 (pushing the data to CSs). If the error persists, the client will restart from step 1.

3.3.3 *Record Append*

With *Record Append* the GFS provides an atomic write operation. When a client uses this operation to change a file, the client transmits only the data it wants to write, not the position. The CS then decides where the data is written and returns the offset to the client.

The workflow for record append is similar to the write operation explained in 3.3.2. The differences are in the following steps:

3. The client generally pushes the data to the last chunk of the file.

4. Instead of requesting a write, the client sends a record appen request.

5. Before the primary applies the changes it checks if the data fits in the chunk (see below). If it fits the following steps are the same as for the write operation.

Something different happens, if the primary's check fails:

If a client wants to append data to a file and the last chunk does not provide enough free space the primary does not apply the changes.

0100 0100 0110 1101

Instead the chunk is padded (e.g. filled up with zeros) and a new chunk is created for the file. The primary then responds to the client with an error asking it to retry the request with the new created chunk.

0100 0100 0110 1101 0000 0000

The client will retry, now pushing the data to the new chunk and will succeed.

0010 0001 1000

Since the chunk size is limited the record append is technically limited to data with a size of 64 MB or less. GFS further restricts the record append operation to data with no more than one forth of the chunk size. This decision was made to reduce the fragmentation which is created by unsuccessful record append operations.

3.4 Features

GFS has a few features implemented to enhance performance and fault tolerance. Chunk replication and garbage collection are two of them which will be explained in the following sections.

3.4.1 Chunk Replication

As mentioned earlier, chunks in the GFS are replicated. By default there are three replicas of each chunk.

Chunk Creation: When a new chunk is created the master decides where the replicas are placed. This decision is affected by three factors. First new replicas are preferably placed on chunkservers with the most free space. This is done to fill up newly added chunkservers and to have a balanced usage of disk space. Secondly the number creations per chunkserver at any time should not be too high. This avoids flooding new chunkservers with requests. And third the replicas for each chunk should not be on the same rack of chunkservers. This enhances the fault tolerance of the system when whole racks go down e.g. due to network problems.

Chunk Re-replication: As soon as the number of replicas for a chunk is lower than specified the master will re-replicate it. When re-replicating a chunk the same factors are considered as during the chunk creation. Additionally the re-replication of chunks is prioritized: chunks which have lost more replicas are prioritized to others. Deleted files (see 3.4.2) get a lower priority. And at last files currently in use get a higher priority.

Chunk Rebalancing: The master periodically checks the distribution of replicas. Depending on chunkserver load and disk space usage the master moves replicas between chunkservers to improve the load and disk space usage.

3.4.2 Garbage Collection

When a file is deleted in the GFS the master logs the deletion and renames the file, creating a "hidden" file. The deletion time stamp is included in the file name. The garbage collection runs periodically and searches fo files which have been deleted at least three days (default value) ago. Finding such a file results in the final deletion of the file. (figure 5)

Until the deletion by the garbage collection the file can still be accessed (using the new file name) or undeleted.

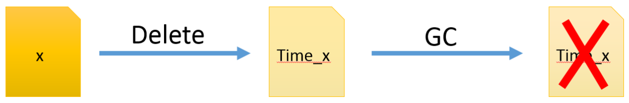

Figure 5: GFS garbage collection

4. COMPARISON

Comparing the two distributed file systems, only a closer look uncovers a few differences between HDFS and GFS. But before pointing out these differences a short summary of similarities will be given.[5]

Similarities

Both systems are optimized for operations of the same kind. Processing large files with high performance is one important key feature in the systems. Optimization was done with the assumption that files are written once and read many times. Appending to files has a high performance and guarantees consistency. In contrast modifying files is supported but not seen as important operation and therefore not optimized. Having these things in mind one can see that GFS and HDFS allow performant streaming of files.

Differences There are only a few minor differences. GFS allows concurrent appending of files since appending is an atomic operation. This is not the case in HDFS. The HDFS's namenode is a single point of failure and provides no automatic recovery. In GFS there is the master, which is in some way also a single point of failure. But on a failure the master tries to recover. If this is not working there are also so called shadow masters which can take over operation. HDFS provides the feature of permission allowing to restrict access to files for different users. The GFS seems not to implement these at the current state.

It can be seen that there are not many differences. The main difference between both systems is the availability: GFS is a proprietary system in contrast to HDFS which is open source.

5. CONCLUSIONS AND FUTURE WORK

... not finished ...

This paragraph will end the body of this sample document. Remember that you might still have Acknowledgments or Appendices; brief samples of these follow.

6. REFERENCES

[1] Shvachko, K. and Hairong Kuang and Radia, S. and Chansler, R.: The Hadoop Distributed File System, in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium*, pp.1-10

[2] Hadoop: What it is and why it matters, online at `http://www.sas.com/en_us/insights/big-data/hadoop.html`, [accessed: May 2015]

[3] Hadoop tutorial, online at `http://www.bigdataplanet.info/2013/10/hadoop-tutorials-part-1-what-is-hadoop.html`, [accessed: May 2015]

[4] Thomas Kiencke: Hadoop Distributed File System, *Institute of Telematics, University of Lubeck, Germany*, online at `https://media.itm.uni-luebeck.de/teaching/ws2012/sem-sse/thomas-kiencke-hdfs-ausarbeitung.pdf`, [accessed: May 2015]

[5] R.Vijayakumari, R.Kirankumar, K.Gangadhara Rao: Comparative analysis of Google File System and Hadoop Distributed File System, in *International Journal of Advanced Trends in Computer Science and Engineering*, Vol.3 , No.1, pp.: 553-558, 2014

[6] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung: The Google File System, in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, 2003*, pp.29-43