# Google File System

(Operating System)

IIIT Hyderabad

Department of Computer Science

Under the guidance of

Prof. Shatrunjay Rawat

TA: Danish Zargar

Contributors

Amogh Bhole(2019201071)

Souptik Mondal (2019201090)

Shubhankar Saha(2019201097)

Divyansh Shrivastava (2019201048)

## Abstract

Google File System (GFS) is a distributed file system containing three major components that is a Client, a Master Server and a bunch of chunk servers. Our goal in this project is to implement a fully functional Google File System based upon its original Research Paper. The functionalities that are supported in our implementation include uploading and downloading a file between the client and the chunk servers, Replication of chunks among the chunk servers after an upload is finished, Updating a file to append some extra contents to the end of it, Leasing a file for a fixed amount of time such that no other client can download the file at that time. Along with these basic functionalities, our implementation can also do chunk redistribution upon failure of some chunk servers.

# 1. Introduction

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of huge data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power supplies. Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system.

We have added functionality for:

1. Upload file
2. Download File
3. Updating file
4. Lease functionality
5. Heartbeat
6. Replication of chunks
7. Distribution in case of node down
8. Hash matching while downloading the file
9. Backup Master Server

## 1.1 Upload

While uploading, the client have to enter "upload filename".

Then a request to the master is sent with the file name , file size and the hash of the file.

Master stores the details received by the client in it's internal table and latter stores /updates in the json files.

Master calculates the number of chunks the file has to be divided and the assigns them unique chunk_ids.

Master also does a round robin allocation of the chunks into the then "AVAILABLE" chunk servers.

At last Master replies the client back with the list of chunk_servers ip_port and the list of chunk_id's to be uploaded into them.

Client then makes chunks of the file and sends them to the appropriate Chunk Servers as instructed by the Master.

Chunk Server stores them.

Sends acknowledgement when done

Client sends acknowledgement to master when upload to CS is done.

Master then initiates replication.

It sends instructions to CS about which chunk_id it has to copy from which Chunk server.

The info about primary and secondary chunk details are stored in a table corresponding to the file name.

## 1.2 Download

client have to enter "download file_name"

Then a request to the master is sent with the file name .

Master checks if the file is avaiable(not in lease).

Master picks the Primary entry of the file table containing chunk_ids and ip_port list.

And send these details to client.

Master also sends the hash of this file to client.

Client after receiving the list of chunk_ids and ip_port connects to each chunk Server independently

And initiates download of chunks.

After obtaining all the chunks.

Clients merges the chunks into a single file (file_name).

and calculates the hash.

Finally sends acknowledgment to the master when done.

**1.3 Update**

Client enters "update old_file new_file"

client sends small 'u' old_file|new_file to master

Master maintains a DS where it links the old file with the new file.

And performs upload on the new_file.

**1.4 Lease**

Any client enters and sends lease file_name to the master.

Master on receiving this requests changes the status of that file to unavailable in the file_status table or 60 seconds.

Now when a request is received for this file, and the file is unavailbe.

A waiting signal is sent to that client.

and the request is made to wait for 20 seconds.

After 20 seconds, again the file status is checked.

If available then normal processing of the request is done.

And if not, then an file currently buzy, try again later signal is sent to client.

**1.5 Heart Beat**

The main functionality of the Heartbeat is to check the status of the chunk-servers after a regular interval (10 seconds). Here we have used three status bits for this purpose (A: for "alive", C: for "coma" and D: for "Dead")

After 10 seconds the master server ping each of the chunk servers to check if its live or not. If it doesn't respond then that chunks server's status bit becomes "C". After 10 seconds again if that chunk server doesn't respond then it's status becomes "D" and if it responds then its status becomes "A" i.e. alive.

According to these status replications is done.

### 1.6 Replication

Replication is an essential feature of the Google File System in a way that it is the basis of the redundancy of nodes and gives GFS it’s strong reliability.

In our Implementation, Replication of chunks is done among the chunk servers coordinated by the Master Server. Replication is initiated after the upload operation of any file is finished. Client signals the master that the file is uploaded successfully and replication process can begin. The secondary Replica locations for the file are calculated upon the upload phase only while the primary replica locations are getting calculated. Master then sends all the relevant information to the respective chunk servers about where to get the chunks from for the replication processes. The data transmission is limited only to the chunks servers only.

### 1.7 Node down

When a chunk_server is down,then the heratbeat module detects it.

Call's node_down function with the ip_port when a CS is in coma state.

A submodule is called which finds out 3 necessary CS details for this operation.

1. Prev Chunk_server

2. Next active CS

3. Next to next CS

Prev Chunk_server-- used for getting the secondary chunks of the Down CS. from Prev Chunk_server's primary.

Next active CS -- It contains the Lost primary CS's chunks as it's secondary.

So , the secondary part of this CS is pulled to it's primary.

The primary secondary chunk_ids of Dead CS is copied to it's secondary using Prev Chunk_server.

Next to next CS -- The secondary chunk_ids (now primary chunk_ids) of the "Next active CS" is replicated to it's secondary

Both logically and physically.

The above operation is done file wise logically and as awhole physically.

i.e. modification of internal tables are done file wise.

and the instructions of individual CS for replication is given as a whole. (All chunk_id info together to each each CS)

### 1.8 Hash Function

Here we are using SHA1 Hash function for checking if the downloaded file is corrupted or not during download or upload.

At the time of file upload, hash value is being calculated and sent it along with the file name to master server.

During download, master server sends the hash value to the client. The client download files from chunk servers and merge them. After that it again calculate the hash value of the merged file and check with the original hash value sent by the master server.

### 1.9 Backup Master

We are having a secondary backup Master server running all the time along with the primary master Server.

The primary Master writes all it’s internal dictionaries and chunk_id counter into json files with a frequency of 15 seconds.

Our secondary backup Master reads these json files with a frequency of 15 seconds only when the Primary master is up.

When primary master is down , the client sends the request to the secondary backup Master and things are processed accordingly.

## 2. Data Structures Used

**dict_chunk_details**

Dictionary used to map the file name and the dict of primary and secondary with the dict of ipport mapped to chunk ids.

**dict_all_chunk_info**

file name mapped to all the chunk ids

**dict_size_info**

file name mapped to size

**dict_status_bit**

ip port containing the current status of chunk server

**dict_file_status**

file name containing status of the file

**dict_file_hash**

file name mapped to the hash

**dict_filename_update**

map file name as list of multiple files for update feature

## 3. Code

https://github.com/divxvid/GFS_19_StackSmashers

## 4. References

https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

https://en.wikipedia.org/wiki/Google_File_System

https://computer.howstuffworks.com/internet/basics/google-file-system.htm