GOOGLE FILE SYSTEM

`

PROJECT REPORT

Faculty: Lini Thomas
Mentor: Sanchit Saini

TEAM MEMBERS:

Tushar Bhatt - 2020202011
Arpit Agarwal- 2020202009
Shivam Rai - 2020202001

<u>ABSTRACT</u>

Google File System (GFS) is a distributed file system containing three major components that is a Client, a Master Server and chunk servers. Our aim in this project was to implement a fully functional Google File System based upon its original Research Paper. The functionalities that we have included 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 some amount of time such that no other client can access (download) the file at that time. Along with these before mentioned functionalities, our implementation also includes chunk redistribution upon failure of some chunk servers by implementing *HEARTBEAT* functionality by continuously checking for whether chunk servers are active or not.

<u>INTRODUCTION</u>

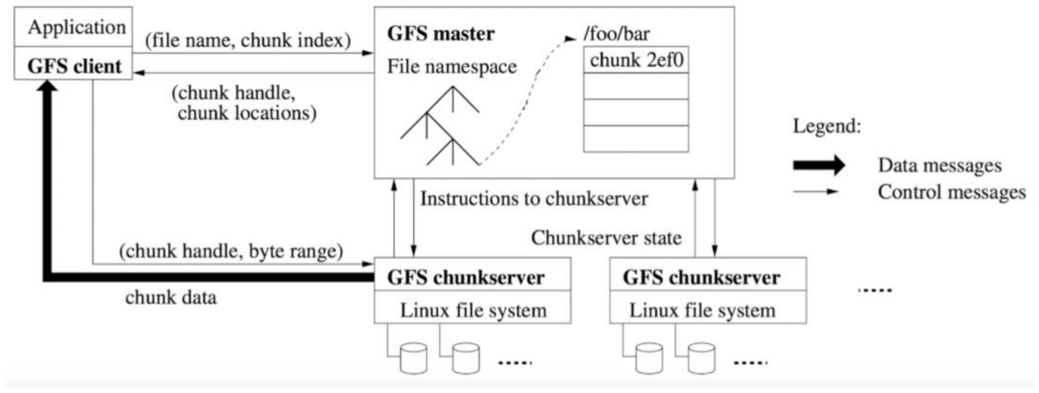
Google File System is a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

Google has designed and implemented a scalable distributed file system for their large distributed data intensive applications. They named it Google File System, GFS. Google

File System was designed by Sanjay Ghemawat, Howard Gobioff and Shun-Tak Leung of Google in 2002-03. GFS provides fault tolerance, while running on inexpensive commodity hardware and also serving a large number of clients with high aggregate performance. Even though the GFS shares many similar goals with previous distributed file systems, the design has been driven by Google's unique workload and environment.Google had to rethink the file system to serve their "very large scale" applications, using inexpensive commodity hardware.

In this context, we have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of huge data processing needs. As it has been mentioned above, the design of GFS has been driven by key observations of the application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions.

The figure below depicts the overall layout of Google File System -:



<u>IMPLEMENTATION</u>

In this project, we have implemented the below functionalities:

1) ***UPLOAD*** a file
2) ***DOWNLOAD*** a file
3) ***UPDATE*** a file
4) Putting ***LEASE*** on a file
5) ***HEARTBEAT*** and Handling ***CHUNKSERVER FAILURE***
6) ***BACKUP MASTER SERVER***

In this implementation we have four chunk servers and two master servers i.e a main master server and a backup master server. The implementation is done in a distributed setting.

All the logic in a nutshell is the master server contains all the metadata in various dictionaries,the details of which are provided below. Now the client or the chunk server has to connect to the master server to get these metadata information and proceed accordingly. For ex: if the client wants to upload a file, it connects to masterserver to get the chunk server id's in which to upload and then connects to these chunk servers for the actual upload process to happen. This is the concept which is followed in the entire project. The master server is only for containing metadata and other checking functionalities. The actual operation for example the actual file transfer is performed between client and chunkserver.

The description of all the global data structures (essentially dictionaries) used in our project are mentioned below. All these data structures are defined globally in the master server file.

dict_chunk_details : key of dictionary is file name and value is the dict of primary and secondary with the key of ip_port mapped to chunk ids.

For ex: dict_chunk_details[file_name]['P'][ip_port]=chunk_ids

dict_chunk_details[file_name]['S'][ip_port]=chunk_ids

P stands for Primary and S stands for secondary

dict_all_chunk_info : key of this dictionary is file name and value is all the chunk ids of this file

dict_size_info : key is file name and value is the file size

dict_status_bit : key is ip port combination[ip:port] and value is the current status of chunk server(whether it is active or not)

dict_file_status : key is file name and value is status of the file (Available or Blocked) and is used for lease operation.

dict_file_hash : key of dictionary is file name and value is the hash of that file

dict_filename_update : key is file name and value is list of multiple files for update feature.

Below there is a deep dive on each of the functionalities mentioned above.

1 <u>UPLOAD</u>

For the upload operation, the client has to enter the command **upload file_name.** Once this is done by the user or the client, a request is sent to the masterserver with the following details:- file name,file size and the hash value (calculated using SHA-1) of the file.

The masterserver stores the details received by the client and updates them in its internal data structures which are mentioned above.

The master server calculates the number of chunks from the file size and by using a **Round Robin Algorithm** decides which chunks of the file will be allocated to which chunkserver. This calculation is done by the master server and is sent back to the client. The chunksize is fixed for our implementation as 512 kB.

Now the client receives the list of chunk servers' ip and port and the list of chunk Id's which have been allocated to the particular chunkserver and based on this information, the client connects to the required chunkserver and sends them the appropriate chunk.

The chunkserver stores these chunks sent by the client and once the process is complete sends acknowledgment back to the client.

Client then sends acknowledgement to master when upload is done.

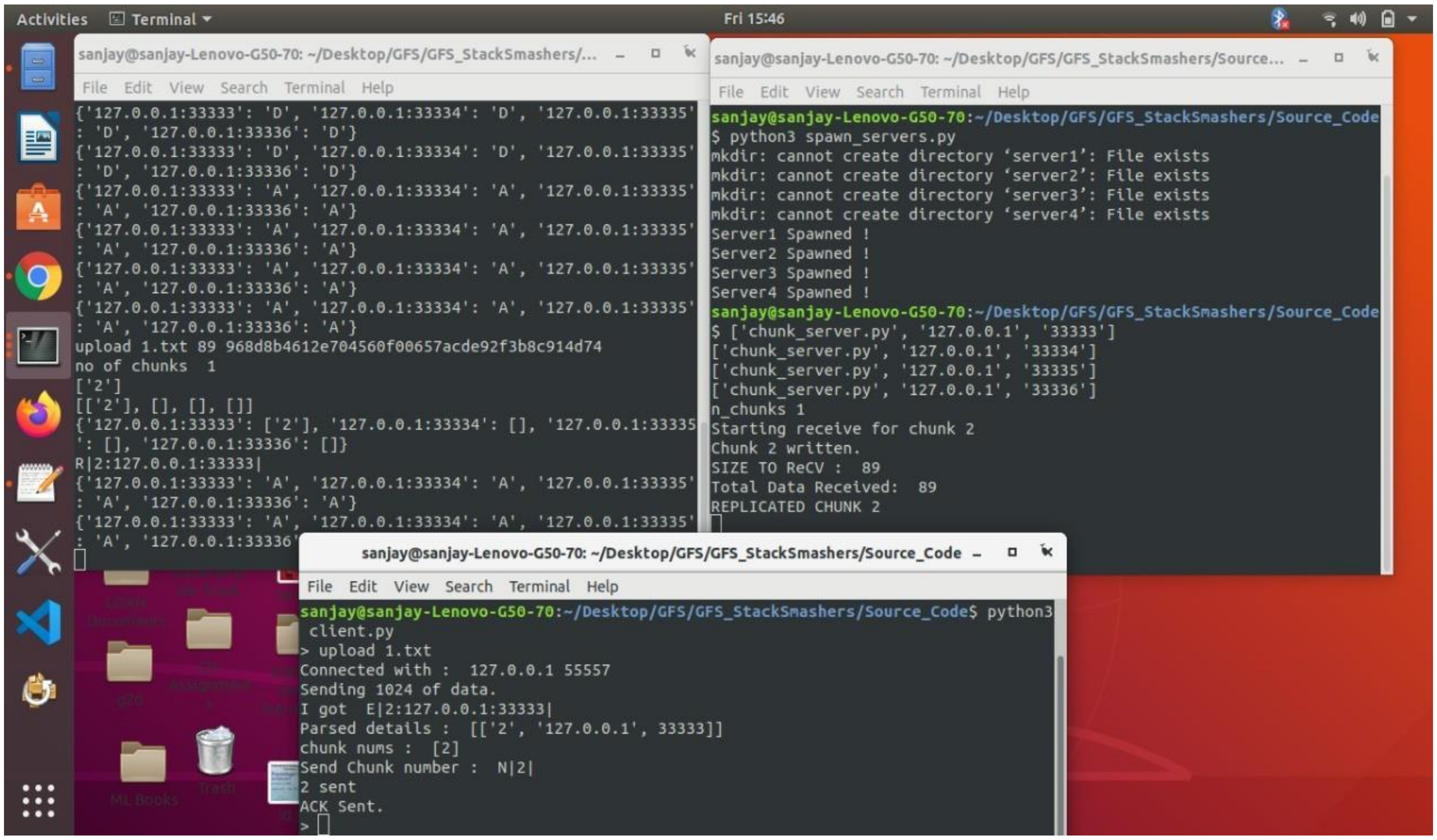
Once the acknowledgment is received, the master server initiates replication of the chunks.

It sends instructions to chunkserver about which chunk Id has to copied from which Chunk server.

The info about primary and secondary chunk details are stored in a dictionary which is *dict_chunk_details* corresponding to the file name.

The below figure shows the upload process in our implementation:

This is the status of the client(bottom), chunk servers (top right) and master_server(top left) after uploading a file named 1.txt.



```
{'127.0.0.1:33333': 'D', '127.0.0.1:33334': 'D', '127.0.0.1:33335'
: 'D', '127.0.0.1:33336': 'D'}
{'127.0.0.1:33333': 'D', '127.0.0.1:33334': 'D', '127.0.0.1:33335'
: 'D', '127.0.0.1:33336': 'D'}
{'127.0.0.1:33333': 'A', '127.0.0.1:33334': 'A', '127.0.0.1:33335'
: 'A', '127.0.0.1:33336': 'A'}
{'127.0.0.1:33333': 'A', '127.0.0.1:33334': 'A', '127.0.0.1:33335'
: 'A', '127.0.0.1:33336': 'A'}
{'127.0.0.1:33333': 'A', '127.0.0.1:33334': 'A', '127.0.0.1:33335'
: 'A', '127.0.0.1:33336': 'A'}
{'127.0.0.1:33333': 'A', '127.0.0.1:33334': 'A', '127.0.0.1:33335'
: 'A', '127.0.0.1:33336': 'A'}
upload 1.txt 89 968d8b4612e704560f00657acde92f3b8c914d74
no of chunks  1
['2']
[['2'], [], [], []]
{'127.0.0.1:33333': ['2'], '127.0.0.1:33334': [], '127.0.0.1:33335
': [], '127.0.0.1:33336': []}
R|2:127.0.0.1:33333|
{'127.0.0.1:33333': 'A', '127.0.0.1:33334': 'A', '127.0.0.1:33335'
: 'A', '127.0.0.1:33336': 'A'}
{'127.0.0.1:33333': 'A', '127.0.0.1:33334': 'A', '127.0.0.1:33335'
: 'A', '127.0.0.1:33336'
```

```
sanjay@sanjay-Lenovo-G50-70:~/Desktop/GFS/GFS_StackSmashers/Source_Code
$ python3 spawn_servers.py
mkdir: cannot create directory 'server1': File exists
mkdir: cannot create directory 'server2': File exists
mkdir: cannot create directory 'server3': File exists
mkdir: cannot create directory 'server4': File exists
Server1 Spawned !
Server2 Spawned !
Server3 Spawned !
Server4 Spawned !
sanjay@sanjay-Lenovo-G50-70:~/Desktop/GFS/GFS_StackSmashers/Source_Code
$ ['chunk_server.py', '127.0.0.1', '33333']
['chunk_server.py', '127.0.0.1', '33334']
['chunk_server.py', '127.0.0.1', '33335']
['chunk_server.py', '127.0.0.1', '33336']
n_chunks 1
Starting receive for chunk 2
Chunk 2 written.
SIZE TO ReCV :  89
Total Data Received:  89
REPLICATED CHUNK 2
```

```
sanjay@sanjay-Lenovo-G50-70:~/Desktop/GFS/GFS_StackSmashers/Source_Code$ python3
 client.py
> upload 1.txt
Connected with :  127.0.0.1 55557
Sending 1024 of data.
I got  E|2:127.0.0.1:33333|
Parsed details :  [['2', '127.0.0.1', 33333]]
chunk nums :  [2]
Send Chunk number :  N|2|
2 sent
ACK Sent.
> 
```

2 DOWNLOAD

For the download operation, the client has to enter the command **download file_name.**

Once this is done a request to the masterserver is sent with the file name from the client.

Masterserver checks if the file is available i.e not in lease.

Masterserver picks the Primary entry of the *dict_chunk_details* dictionary containing chunk Ids and ip_port list and sends these details to the client.

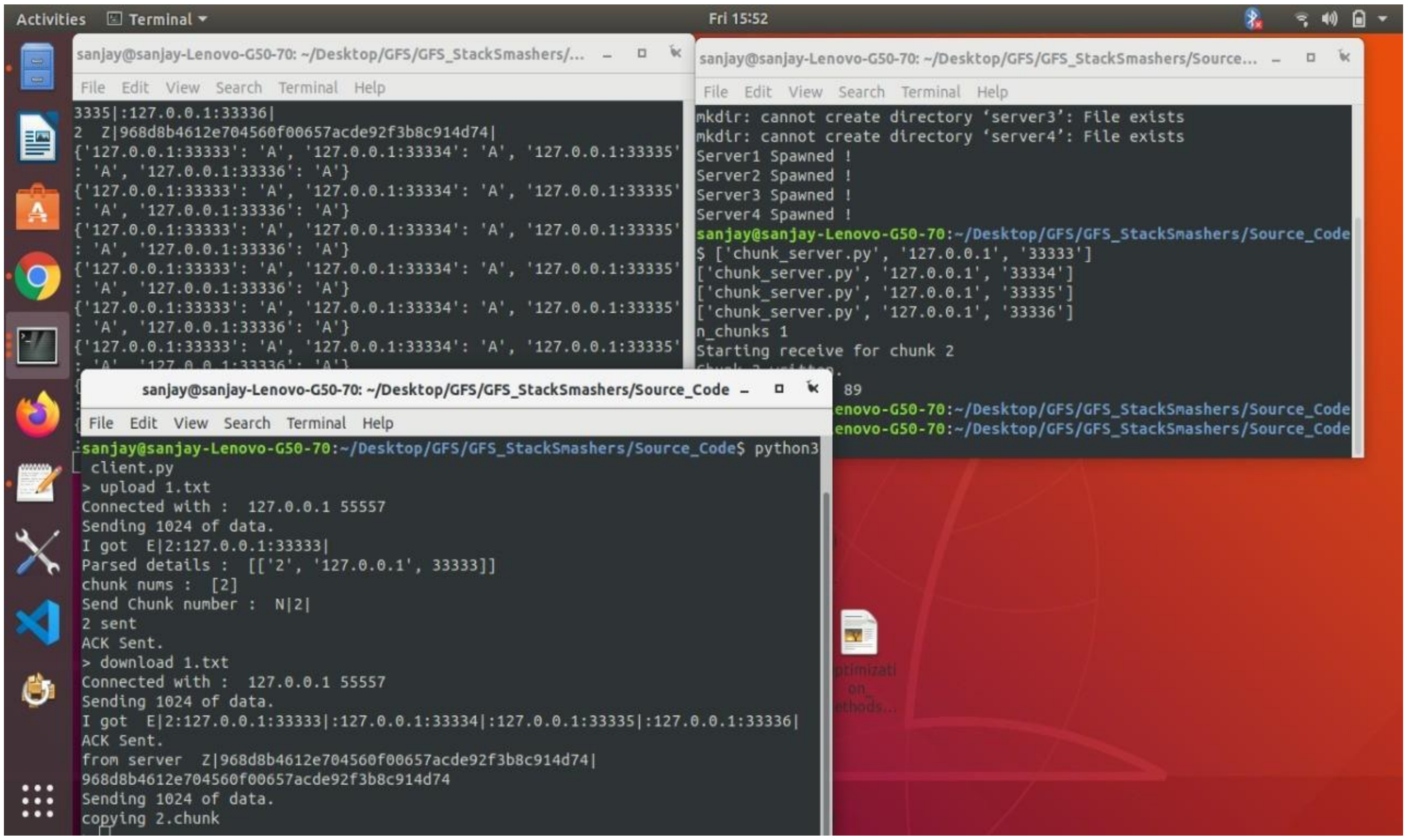
Client after receiving the list of chunk Ids and ip_port connects to each chunk server independently and parallely downloads the chunks of the file from each of the chunk servers.

After obtaining all the chunks, the client merges the chunks into a single file and names it with the given file name.

Also it will receive the hash value of the original file from the master server. After merging the chunks, we check the hash value of the merged chunks with the one we got from the master server and check if they match or not. This is done to authenticate if we are able to download correctly or not.

Finally after the process is over, the client sends acknowledgment to the master server.

The below figure shows the download process happening. This is the status of the client(bottom), chunk servers (top right) and master_server(top left) after downloading a file named 1.txt.


Activities
Terminal
Fri 15:52
sanjay@sanjay-Lenovo-G50-70: ~/Desktop/GFS/GFS_StackSmashers/...
File Edit View Search Terminal Help

```
3335|:127.0.0.1:33336|
2  Z|968d8b4612e704560f00657acde92f3b8c914d74|
{'127.0.0.1:33333': 'A', '127.0.0.1:33334': 'A', '127.0.0.1:33335'
: 'A', '127.0.0.1:33336': 'A'}
{'127.0.0.1:33333': 'A', '127.0.0.1:33334': 'A', '127.0.0.1:33335'
: 'A', '127.0.0.1:33336': 'A'}
{'127.0.0.1:33333': 'A', '127.0.0.1:33334': 'A', '127.0.0.1:33335'
: 'A', '127.0.0.1:33336': 'A'}
{'127.0.0.1:33333': 'A', '127.0.0.1:33334': 'A', '127.0.0.1:33335'
: 'A', '127.0.0.1:33336': 'A'}
{'127.0.0.1:33333': 'A', '127.0.0.1:33334': 'A', '127.0.0.1:33335'
: 'A', '127.0.0.1:33336': 'A'}
{'127.0.0.1:33333': 'A', '127.0.0.1:33334': 'A', '127.0.0.1:33335'
: 'A', '127.0.0.1:33336': 'A'}
```

sanjay@sanjay-Lenovo-G50-70: ~/Desktop/GFS/GFS_StackSmashers/Source...
File Edit View Search Terminal Help

```
mkdir: cannot create directory 'server3': File exists
mkdir: cannot create directory 'server4': File exists
Server1 Spawned !
Server2 Spawned !
Server3 Spawned !
Server4 Spawned !
sanjay@sanjay-Lenovo-G50-70:~/Desktop/GFS/GFS_StackSmashers/Source_Code
$ ['chunk_server.py', '127.0.0.1', '33333']
['chunk_server.py', '127.0.0.1', '33334']
['chunk_server.py', '127.0.0.1', '33335']
['chunk_server.py', '127.0.0.1', '33336']
n_chunks 1
Starting receive for chunk 2
 89
enovo-G50-70:~/Desktop/GFS/GFS_StackSmashers/Source_Code
enovo-G50-70:~/Desktop/GFS/GFS_StackSmashers/Source_Code
```

sanjay@sanjay-Lenovo-G50-70: ~/Desktop/GFS/GFS_StackSmashers/Source_Code
File Edit View Search Terminal Help

```
sanjay@sanjay-Lenovo-G50-70:~/Desktop/GFS/GFS_StackSmashers/Source_Code$ python3
 client.py
> upload 1.txt
Connected with :  127.0.0.1 55557
Sending 1024 of data.
I got  E|2:127.0.0.1:33333|
Parsed details :  [['2', '127.0.0.1', 33333]]
chunk nums :  [2]
Send Chunk number :  N|2|
2 sent
ACK Sent.
> download 1.txt
Connected with :  127.0.0.1 55557
Sending 1024 of data.
I got  E|2:127.0.0.1:33333|:127.0.0.1:33334|:127.0.0.1:33335|:127.0.0.1:33336|
ACK Sent.
from server  Z|968d8b4612e704560f00657acde92f3b8c914d74|
968d8b4612e704560f00657acde92f3b8c914d74
Sending 1024 of data.
copying 2.chunk
```

3 <u>UPDATE</u>

For the update operation, the client has to enter the command **update old file_name new file_name.**

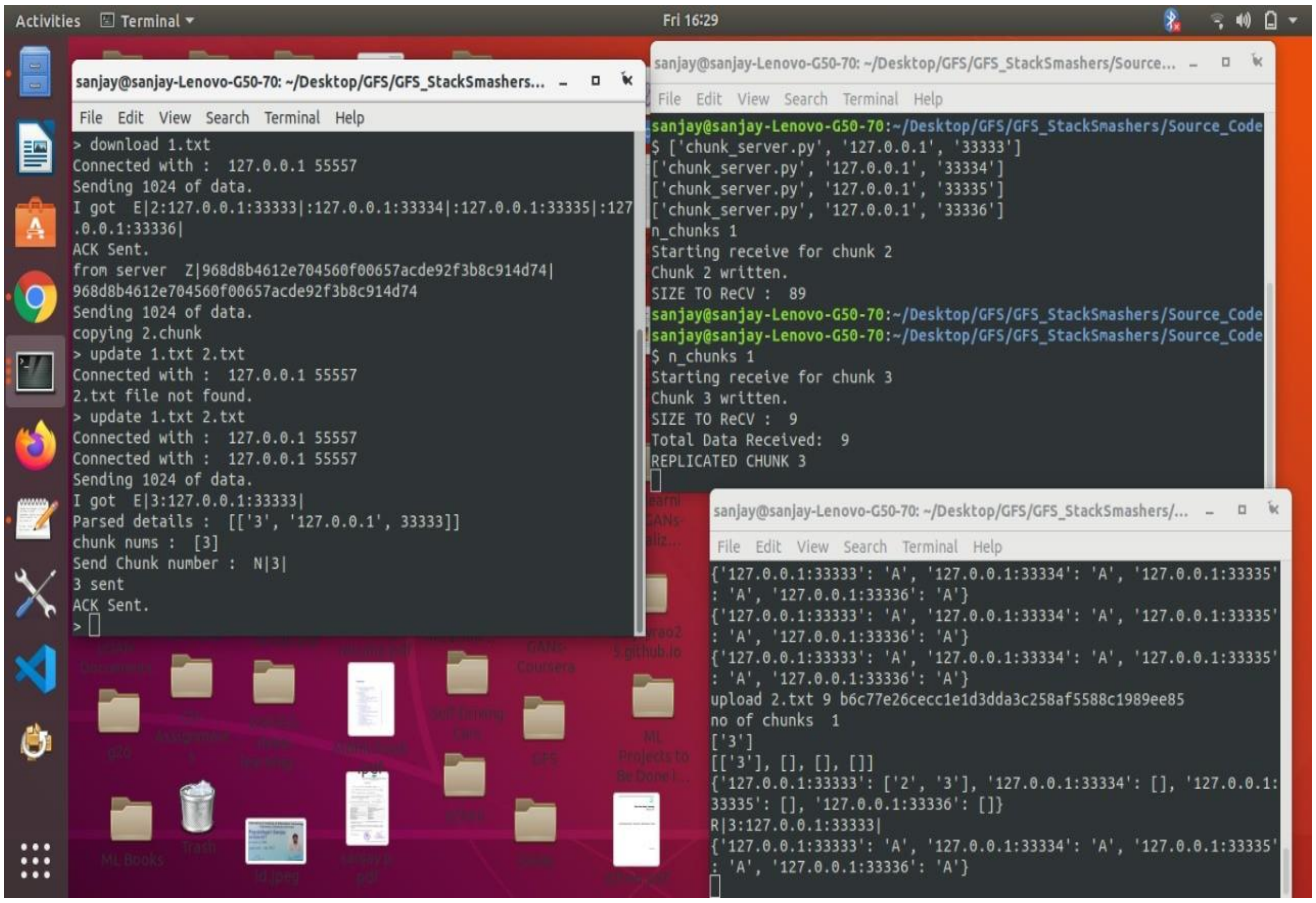
Once this is done a request to the masterserver is sent with the file name by the client. The master server on receiving the update request appends the new file_name mapped to the old_filename in one of its dictionaries *dict_filename_update*.

After that the client uploads the new file.

Now while downloading we go through all the file names in *dict_filename_update* and download all the chunks from the correct chunk servers and merge them into one file.

So for example, if the request by client is update 1.txt 2.txt, then the result of this operation will be the contents of 2.txt appended to the end of the contents of 1.txt and the downloaded result being 1.txt.

The below figure shows the update process happening.This is the status of the client(top left), chunk servers (top right) and master_server(bottom right) after updating a file named 1.txt 2.txt.


Activities
Terminal
Fri 16:29
sanjay@sanjay-Lenovo-G50-70: ~/Desktop/GFS/GFS_StackSmashers...
File Edit View Search Terminal Help
sanjay@sanjay-Lenovo-G50-70: ~/Desktop/GFS/GFS_StackSmashers/Source...
sanjay@sanjay-Lenovo-G50-70: ~/Desktop/GFS/GFS_StackSmashers/...

```
> download 1.txt
Connected with :  127.0.0.1 55557
Sending 1024 of data.
I got  E|2:127.0.0.1:33333|:127.0.0.1:33334|:127.0.0.1:33335|:127
.0.0.1:33336|
ACK Sent.
from server  Z|968d8b4612e704560f00657acde92f3b8c914d74|
968d8b4612e704560f00657acde92f3b8c914d74
Sending 1024 of data.
copying 2.chunk
> update 1.txt 2.txt
Connected with :  127.0.0.1 55557
2.txt file not found.
> update 1.txt 2.txt
Connected with :  127.0.0.1 55557
Connected with :  127.0.0.1 55557
Sending 1024 of data.
I got  E|3:127.0.0.1:33333|
Parsed details :  [['3', '127.0.0.1', 33333]]
chunk nums :  [3]
Send Chunk number :  N|3|
3 sent
ACK Sent.
>
```

```
sanjay@sanjay-Lenovo-G50-70:~/Desktop/GFS/GFS_StackSmashers/Source_Code
$ ['chunk_server.py', '127.0.0.1', '33333']
['chunk_server.py', '127.0.0.1', '33334']
['chunk_server.py', '127.0.0.1', '33335']
['chunk_server.py', '127.0.0.1', '33336']
n_chunks 1
Starting receive for chunk 2
Chunk 2 written.
SIZE TO ReCV :  89
sanjay@sanjay-Lenovo-G50-70:~/Desktop/GFS/GFS_StackSmashers/Source_Code
sanjay@sanjay-Lenovo-G50-70:~/Desktop/GFS/GFS_StackSmashers/Source_Code
$ n_chunks 1
Starting receive for chunk 3
Chunk 3 written.
SIZE TO ReCV :  9
Total Data Received:  9
REPLICATED CHUNK 3
```

```
{'127.0.0.1:33333': 'A', '127.0.0.1:33334': 'A', '127.0.0.1:33335'
: 'A', '127.0.0.1:33336': 'A'}
{'127.0.0.1:33333': 'A', '127.0.0.1:33334': 'A', '127.0.0.1:33335'
: 'A', '127.0.0.1:33336': 'A'}
{'127.0.0.1:33333': 'A', '127.0.0.1:33334': 'A', '127.0.0.1:33335'
: 'A', '127.0.0.1:33336': 'A'}
upload 2.txt 9 b6c77e26cecc1e1d3dda3c258af5588c1989ee85
no of chunks  1
['3']
[['3'], [], [], []]
{'127.0.0.1:33333': ['2', '3'], '127.0.0.1:33334': [], '127.0.0.1:
33335': [], '127.0.0.1:33336': []}
R|3:127.0.0.1:33333|
{'127.0.0.1:33333': 'A', '127.0.0.1:33334': 'A', '127.0.0.1:33335'
: 'A', '127.0.0.1:33336': 'A'}
```

4 LEASE

The lease functionality is used for blocking a client from downloading a file. This is done to ensure that it is not allowed for a client to download a file which is being updated or changed by another client. Essentially this is done to handle synchronization problems between various clients while downloading a file.

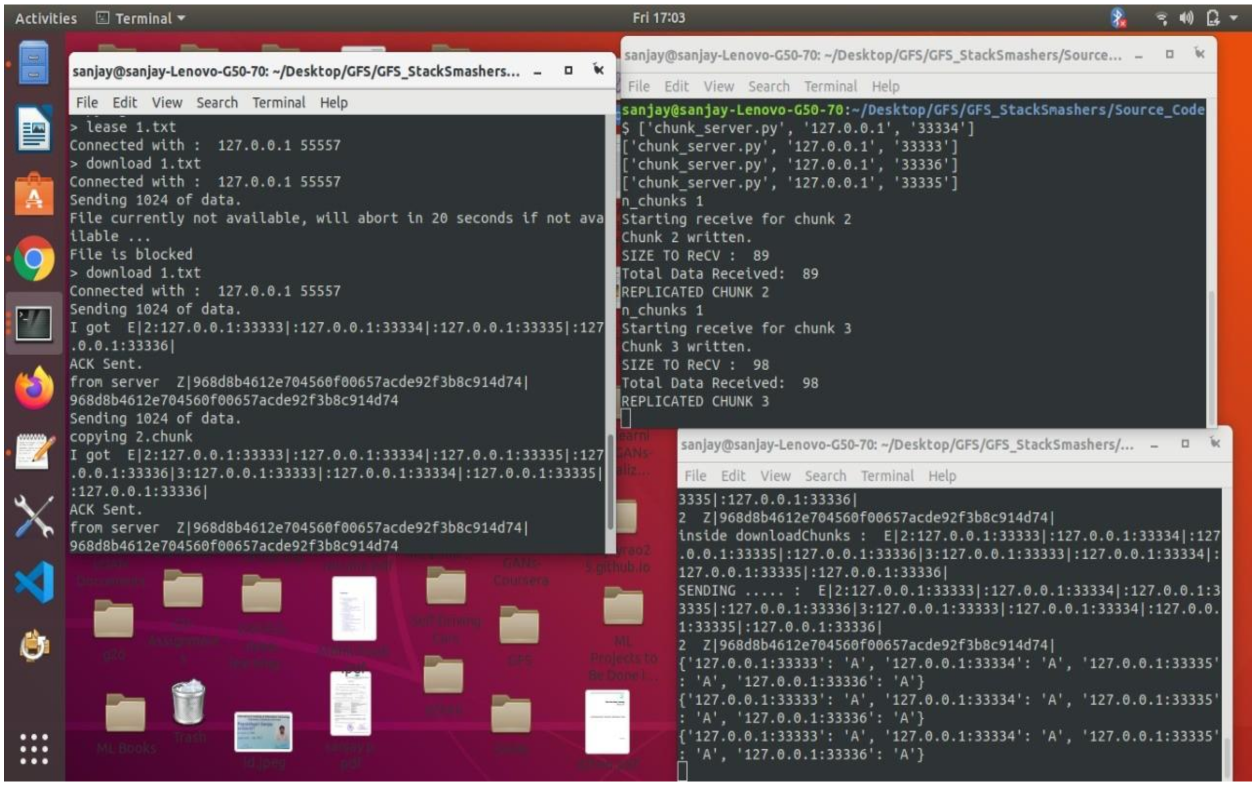
For the lease operation, the client has to enter the command **lease file_name.**

The masterserver on receiving this request changes the status of that file to unavailable in the file_status table for 60 seconds. For this purpose *dict_file_status* dictionary is used in the master server.The masterserver changes the status of the file to **'B'** meaning blocked and the file remains in that state for 60 seconds.

The master server will only let a client download a file when this file status in *dict_file_status* dictionary is showing as 'A'.

Now when a client is trying to download something that has the status as 'B', it will not let it download the file. It is made to wait for 20 seconds and then again try to download. After these 20 seconds, again the file status is checked. If the file status is 'A', then downloading is resumed normally, otherwise downloading is blocked and this message is propagated to the client.

The below figure shows the lease operation happening.This is the status of the client(top left), chunk servers (top right) and master_server(bottom right) after lease of a file named 1.txt.



Activities
Terminal
Fri 17:03
sanjay@sanjay-Lenovo-G50-70: ~/Desktop/GFS/GFS_StackSmashers...
File Edit View Search Terminal Help

```
> lease 1.txt
Connected with :  127.0.0.1 55557
> download 1.txt
Connected with :  127.0.0.1 55557
Sending 1024 of data.
File currently not available, will abort in 20 seconds if not ava
ilable ...
File is blocked
> download 1.txt
Connected with :  127.0.0.1 55557
Sending 1024 of data.
I got  E|2:127.0.0.1:33333|:127.0.0.1:33334|:127.0.0.1:33335|:127
.0.0.1:33336|
ACK Sent.
from server  Z|968d8b4612e704560f00657acde92f3b8c914d74|
968d8b4612e704560f00657acde92f3b8c914d74
Sending 1024 of data.
copying 2.chunk
I got  E|2:127.0.0.1:33333|:127.0.0.1:33334|:127.0.0.1:33335|:127
.0.0.1:33336|3:127.0.0.1:33333|:127.0.0.1:33334|:127.0.0.1:33335|
:127.0.0.1:33336|
ACK Sent.
from server  Z|968d8b4612e704560f00657acde92f3b8c914d74|
968d8b4612e704560f00657acde92f3b8c914d74
```

sanjay@sanjay-Lenovo-G50-70: ~/Desktop/GFS/GFS_StackSmashers/Source...
File Edit View Search Terminal Help

```
sanjay@sanjay-Lenovo-G50-70:~/Desktop/GFS/GFS_StackSmashers/Source_Code
$ ['chunk_server.py', '127.0.0.1', '33334']
['chunk_server.py', '127.0.0.1', '33333']
['chunk_server.py', '127.0.0.1', '33336']
['chunk_server.py', '127.0.0.1', '33335']
n_chunks 1
Starting receive for chunk 2
Chunk 2 written.
SIZE TO ReCV :  89
Total Data Received:  89
REPLICATED CHUNK 2
n_chunks 1
Starting receive for chunk 3
Chunk 3 written.
SIZE TO ReCV :  98
Total Data Received:  98
REPLICATED CHUNK 3
```

sanjay@sanjay-Lenovo-G50-70: ~/Desktop/GFS/GFS_StackSmashers/...
File Edit View Search Terminal Help

```
3335|:127.0.0.1:33336|
2  Z|968d8b4612e704560f00657acde92f3b8c914d74|
inside downloadChunks :  E|2:127.0.0.1:33333|:127.0.0.1:33334|:127
.0.0.1:33335|:127.0.0.1:33336|3:127.0.0.1:33333|:127.0.0.1:33334|:
127.0.0.1:33335|:127.0.0.1:33336|
SENDING ..... :  E|2:127.0.0.1:33333|:127.0.0.1:33334|:127.0.0.1:3
3335|:127.0.0.1:33336|3:127.0.0.1:33333|:127.0.0.1:33334|:127.0.0.
1:33335|:127.0.0.1:33336|
2  Z|968d8b4612e704560f00657acde92f3b8c914d74|
{'127.0.0.1:33333': 'A', '127.0.0.1:33334': 'A', '127.0.0.1:33335'
: 'A', '127.0.0.1:33336': 'A'}
{'127.0.0.1:33333': 'A', '127.0.0.1:33334': 'A', '127.0.0.1:33335'
: 'A', '127.0.0.1:33336': 'A'}
{'127.0.0.1:33333': 'A', '127.0.0.1:33334': 'A', '127.0.0.1:33335'
: 'A', '127.0.0.1:33336': 'A'}
```

5 HEARTBEAT and CHUNKSERVER FAILURE

The HEARTBEAT functionality is used to check the status of the chunk servers after a regular interval (10 seconds).

For checking the status of the chunk servers, the heartbeat thread tries to connect to the chunk servers and send some dummy messages and receive acknowledgments every 10 seconds to check which chunk servers are running.

Here we have used three status bits for this purpose (A: for "alive", C: for "Coma" and D: for "Down").

For handling the case where a chunk server goes down, what we do is **REPLICATION** i.e we replicate the chunks in a chunk server to the next chunk server.

Replication is initiated after the upload operation for a file is finished. Client signals the master that the file is uploaded successfully and the 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.

Now if a chunk server goes down, then the heartbeat thread detects it.

All the chunks which were present in this chunk server which got down get allocated to the next chunk server. The secondary for these chunks will now become primary and the next chunk server to which they got allocated will now become secondary. The next chunk server to which this chunk should get allocated is calculated by the the formula:

i=(i+1)%(Number of active chunk servers)

So for example if chunk server 1 got down and it had chunk number 1. Now suppose this chunk number 1 has it’s secondary in chunk server 2. Now this chunk will get allocated to chunk server 3 and for this chunk number 1, now the primary will be chunk server 2 and the secondary will be chunk number 3.

6 BACKUP MASTER SERVER

For reliability we are having some redundancy and we have a backup Master Server as well which is running all the time along with the primary master Server.

The primary Master server writes all it’s internal dictionaries(meta data) into json files with a frequency of 15 seconds.

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

When the primary master is down , the client sends the request to the backup Master and things are processed accordingly. How it is done is with every operation the client tries to connect first with the master server, if it is not available, then it tries to connect to the backup Master Server.

APPENDIX

File Running Details

1) Run the Master Server : python3 master_server_v1.py
2) Run the Backup Master Server : python3 backup_master_server_v1.py
3) Run all the four chunk servers : python3 spawn_servers.py
4) Run the client : python3 client.py

Then one can run any of the operations which they want to perform via the client.

The details of how to run these operations are given below:

upload file_name

download file_name

[Before downloading, ensure that there is no file with the same file name as the one you are trying to download]

lease file_name

update old_file_name new_file_name

exit

Killing a Chunkserver :

sudo netstat -nlp | grep chunk server port→ Run this command to get the process id of the chunk server you want to kill.

Now kill -9 pid(which you got from the above step) to kill the particular chunk server.

REFERENCES

https://research.google.com/archive/gfs-sosp2003.pdf [The original GFS Research paper]