CSE - 535 Asynchronous Systems Final Project Report

Implementation of Basic Google File System in DistAlgo

Under the guidance of: **Prof. Annie Liu**

Team 10 Members: **Risabh Baheti (111463169), Swethasri Kavuri (111465291), Shashank Rao (111471609)**

1. Problem and Plan

1.1. Introduction

With the huge increase in data being used nowadays, there is a large demand for data storage which cannot be fulfilled using only conventional storage techniques. The data requires newer methodologies to be stored and distributed storage systems is largely being used nowadays, to satisfy this storage need. A distributed data store is a computer network where information is stored on more than one node, often in a replicated fashion [2]. Distributed storage offers the advantages of centralized storage with the scalability and cost base of local storage. Distributed file systems are network file systems where the server can be distributed across several physical computer nodes. File systems that share access to the same block storage are shared disk file systems [3].

Google File System is one such distributed file system. It is scalable and ideal for large, distributed and data-intensive applications. This file system was created to meet the rapidly growing demands of Google’s data processing needs. Thus, there is a marked departure from some of the earlier file system design assumptions, some of them being [1]-

- Component failures are a normal occurrence rather than an exception;
- File sizes are considered to be huge, thus requiring different parameters for I/O and block sizes;
- Most files are considered to be modified by *appending* new data rather than overwriting new data;

Our project involved understanding all aspects of Google File System from the paper [1] and implementing the file system internals as well as the interface extensions for distributed applications using DistAlgo programming language. This report details the functional and technical aspects considered while developing a basic model of GFS using DistAlgo. It also provides performance test results as well as correctness test results obtained by using our implementation.

1.2. Best Existing Implementations

We have found the following existing implementations for Google File Systems on Github:

1. Distributed Systems GFS Implementation by BenningtonCS: https://github.com/BenningtonCS/GFS : This is a well-documented and extensive implementation written in Python language

2. GFS the Google File System in 199 Lines of Python by John Arley Burns: http://clouddbs.blogspot.com/2010/11/gfs-google-file-system-in-199-lines-of.html :

This is a simple implementation written using the Python language that gave us an idea of the basic framework required to be implemented.

We found a lot more implementations on Github, most of them were either very rudimentary to serve our purposes or were extra complicated without proper documentation to be of any use to us. The implementation by BenningtonCS has extensive documentations and clear to understand methods implemented.

1.3. Inputs and Outputs for our Implementation

Inputs:

Our implementation supports the following file system operations for the user:

- Create (fileName)
- Write (fileName, dataToWrite)
- RecordAppend (fileName, dataToWrite)
- Read (fileName)
- Delete (fileName)
- CreateSnapshot (fileName)

Outputs:

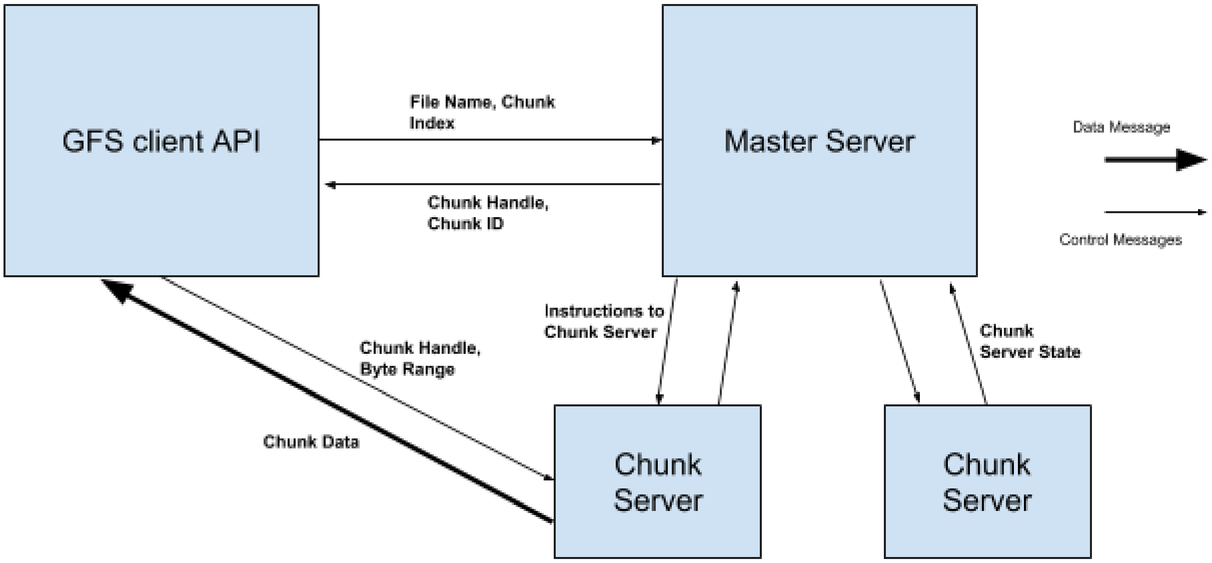
User will obtain the following outputs for the input operations performed:

- Create (fileName) - returns 'CREATED' message to the user if successful, else returns appropriate error message.
- Write (fileName, dataToWrite) - returns 'WRITTEN' message to the user if successful, else returns appropriate error message.
- RecordAppend (fileName, dataToWrite) - returns 'RECORD_APPEND_DONE' message to the user if successful, else returns appropriate error message.
- Read (fileName) - returns 'READ_FILE_DONE' message along with the data read from the file, file name of file read and incremented logical clock to the user if successful, else returns appropriate error message.
- Delete (fileName) - returns 'DELETE_DONE' message to the user if successful, else returns appropriate error message.
- CreateSnapshot (fileName) - returns 'SNAPSHOT_SUCCESS' message to the user if successful, else returns appropriate error message.

1.4. Tasks Performed with Timeline

Week	Risabh	Shashank	Swetha
1	Understand the GFS paper and figure out the design for Master	Understand the GFS paper and figure out the design for Client	Understand the GFS paper and figure out the design for Chunk Server
2	Start building the master and implement message passing between components	Work on Client Server implementation and see how we can implement locking mechanism	Work on Chunk Server implementation and how testing should be done
3	Finish Write ,read API	Work on lease and finish record append	Finish delete API and start on test cases
4	Work on HeartBeat functionality, bug fixing, code comment. Project report	Work on snapshot, performance testing. Project report	Work on bug fixing, correctness testing. Project report

2. System Design



The above figure provides an overview of how the file system is designed. The entire file system consists of three major components - client servers (can be multiple; depending on incoming client operations), Master server (a single server for the entire system) and Chunk servers (has to be multiple in number). Following are the overview of the three components:

1. **Client**: This is the interface that communicates between the underlying file system and the external user. The client APIs is present in one particular server, to which multiple end-users will call. Client supports following operations - **read, write, record append, delete, snapshot** - provided by the user. Each of these operations will communicate with the master server to obtain the file metadata (chunk ID and chunk location); then perform the necessary data operation on the Chunk servers; finally update the metadata back in the Master server as required.

2. **Master Server:** This will be the location where metadata of the entire filesystem is stored. The client requests initiate with the Master server, which returns the stored metadata information regarding the requested file present on the Chunk server, which then allows the Client to talk directly with the individual Chunk server. This prevents the Master server from becoming a bottleneck as metadata is usually short and has low latency. There is a Heartbeat system in order to notify the availability of different Chunk servers to the Master server and the files available on the various chunk servers.

-ref[1] Section 2.4

The master server contains following data maps to hold the metadata in memory for faster operations:

- fileStore : Maps filename to chunkuuids
- chunkStore : Maps chunkuuids to chunk locations.
- chunkServerStore: Maps chunk locations to physical storage on chunk servers.
- snapShotuuids : Maps filename to chunkuuids before snapshot was called.
- lastChunkOfFileStore: Maps fileName to last chunk to which data was writtten for that file.
- deletedChunksList: Maps chunk servers to the list of chunkuuid that need to be deleted from that chunk server.
- leaseQueue: Maps filename with client requests to perform mutations on that file.

3. **Chunk Server:** In GFS, Chunk servers are relatively dumb, meaning that they only know about the chunks, ie, file data broken up into pieces. These servers do not see the whole picture of the entire file, where data is split up across the filesystem, associated metadata, etc. This server will be implemented as simple local storage, in a fixed directory path - ‘\tmp\gfs\chunks\<chunkServerId>’. This path is configurable from the code on Chunk.da. *- ref[1] Section 2.6*

3. Implementation

3.1. System, Language and Tools

Platform: Linux Ubuntu 16.0.4
Processor: 2.7 Ghz Intel Core i7
Memory: 8 GB 1867MHz DDR3
Language: Python 3.6
Libraries: DistAlgo v1.0.0b15
Python builtin modules Tools: Sublime Text and Spyder (Python 3.5) for development, Google doc for documentation

Code Size:
Total Lines of code (including comments): 1981
Total Comments in Code : 457
Total Lines of Code without Comments : 1524
Development and Testing Effort:4 weeks

3.2. Technical Details of Implementation

The sections 3.2.1 and 3.2.2 describe important functions/message receive handlers which will be internally used by Create, Write or Record Append operations.

3.2.1. Implementation and use of 'ALLOC_MASTER' and 'ALLOC_APPEND_MASTER' message handler functions

The **'ALLOC_MASTER'** messages will be passed from the Client server to the Master server during the Create and Write operations, while **'ALLOC_APPEND_MASTER'** messages are passed from Client server to the Master server during Record Append operations. The main function of both these messages are to allocate new chunk UUIDs required for the respective operations.

The message handler for **'ALLOC_MASTER'** messages is defined in Master.da as:

def receive(msg= ('ALLOC_MASTER', isCreate, fileName, numChunks, c),from_= client)

The following are the parameters passed in the message:

- a flag ‘isCreate’ which will distinguish whether the call is for Create operation or for Write operation (in case of Create operation, it will be True; in case of Write operations, it will be False)
- fileName, which is the fileName to be created
- numChunks, which is the number of chunks to be allocated (in case of Create operation, it will be 1; in case of Write operation, it will the calculated number of chunks, required for the data to be written)
- c is the logical clock

This receive handler method will internally call *allocate(fileName, numChunks, isCreate)* function, which will again internally call *allocateChunks(numChunks)* method.

The message handler for **'ALLOC_APPEND_MASTER'** messages is defined in Master.da as:

def receive(msg = ('ALLOC_APPEND_MASTER', fileName, numChunks, c), from_=client)

The following are the parameters passed in the message:

- fileName, which is the fileName to which data is to be appended
- numChunks, which is the number of chunks to be allocated based on calculations done in Client and depending on size of data to append
- c is the logical clock

This receive handler method will internally call *allocate_append(fileName, numChunks)* function, which will also internally call *allocateChunks(numChunks)* method.

This *allocateChunks(numChunks)* method will take in the number of chunks as a parameter, create a unique Chunk UUID for each chunk required, decide which all chunk servers (including which two other servers acting as replicas) will be used for holding the newly allocated chunk, add these decided entries into the *chunkStore* and *chunkServerStore* maps, and finally return the newly created chunk UUID list to the *allocate()* or *allocate_append()* methods.

This *allocate()* function maps the returned *chunkuuid* list to the fileName and adds it to the fileStore map with key as fileName and value as list of chunk UUID to be allocated for the Create or Write operation. However, *allocate_append()* method finds existing chunks for the file (which need to be appended to) from the fileStore map and appends the newly created chunk UUIDs to the end of the value for the given fileName key. Both these functions then return back to their respective receiver handler function, where only the chunks server ids required for the Create/Write/Record Append operation file is added to a list and acknowledgement **‘ALLOC_DONE’** is returned to the Client server in case of Create/Write operations and acknowledgement **‘ALLOC_APPEND_DONE’** is returned to the Client server in case of Record Append operations. The acknowledgement sent by either of the receiver handlers also contains the metadata required for file operations - *chunkuuid* list, *chunkServerStore* list and required *chunkStore* list.

3.2.2. Implementation and use of leaseQueue map

The *leaseQueue* map has been used in the Master server (Master.da file) in order to implement leases as well as locks on files during Write, Record Append and Snapshot operations as mentioned in the paper [1]. We use a map with the key as fileName and value as a list of tuples containing <client Id, operation being performed, logical clock value>. This *leaseQueue* data structure has been used to serialize access to a file, if multiple clients perform different operations on the same file.

Master.da contains a message receiver handle for messages of type **'ADD_TO_LEASE_Q'** defined as:

def receive(msg =('ADD_TO_LEASE_Q', fileName, c, operation), from_= client)

The following are the parameters passed in the message:

- The respective fileName upon which operation is to be done
- a logical clock
- A string containing operation being performed (can be 'WRITE', 'APPEND' or 'SNAPSHOT' depending on the operation being performed)

This message handler will add an entry to the *leaseQueue* map using the parameters passed along with the message as well as using the client Id of the Client passing the message. Once the entry is made into the *leaseQueue* data structure, the method *checkQueueAndReturn()* is internally called. This method will take the take the fileName, client ID, logical time and operation as parameters will check the list mapped with the fileName. If the first entry in the list contains the same client Id, operation and logical clock values, then it means that the current client performing the operation (Write, Append or Snapshot) is the first in the queue and needs to be serviced. In this case, an acknowledgement message **'CLIENT_TURN_IN_LEASE_Q_<WRITE/APPEND/SNAPSHOT>'** is sent back to the client which sent the **'ADD_TO_LEASE_Q'** message. If the first entry in the list does not have the the same client Id as the message passing client, then that client will be kept in a waiting condition until it gets its turn.

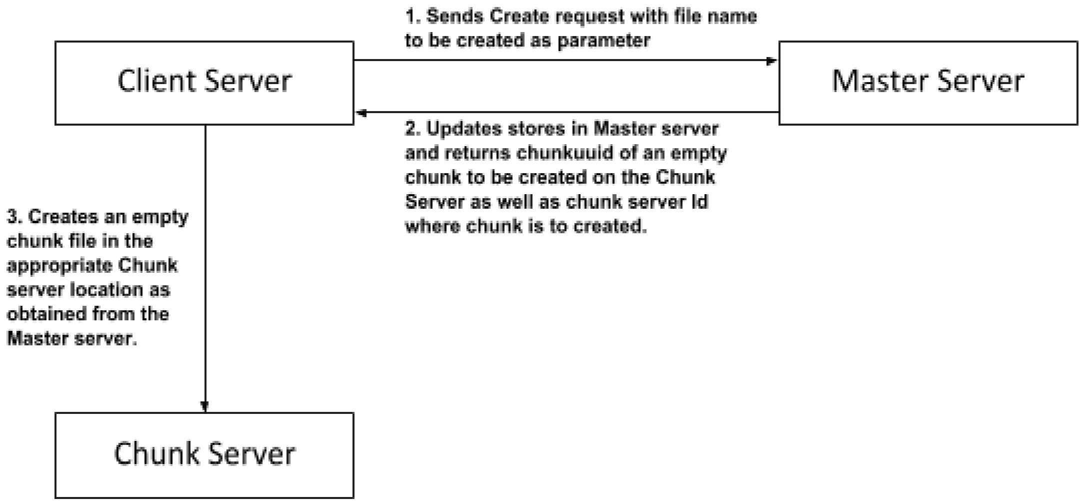
One other important factor we used in implementing the leaseQueue structure is to provide priority for Create Snapshot operations, i.e., if there are multiple Record Append operations already present in the queue for a particular file, and if receive a *SNAPSHOT* operation to be performed on that same file, then this operation request will be added to the beginning of the queue. This means that if there is a Record Append or Write operation currently in progress, then it will be completed but the next operation to be performed will be Snapshot creation, which will then be followed by the Record Append operations on the newly created Snapshot file.

Master.da also contains a message receiver handle for messages of type **'PROCESS_NEXT_IN_LEASE_Q'** defined as:

def receive(msg =('PROCESS_NEXT_IN_LEASE_Q', fileName), from_= client)

This message receiver handler will pop out the first entry in the queue for a given fileName key as well as call *checkQueueAndReturn()* in order to process the next entry present in the queue and thus perform serialized accesses to concurrent operations on fileName.

3.2.3. Create operation


1. Sends Create request with file name to be created as parameter
Client Server
Master Server
2. Updates stores in Master server and returns chunkuuid of an empty chunk to be created on the Chunk Server as well as chunk server Id where chunk is to created.
3. Creates an empty chunk file in the appropriate Chunk server location as obtained from the Master server.
Chunk Server

A brief overview of the working of Create operation is shown in the above figure. The following is an in-depth explanation of the same.

The Create operation will create an empty file in the file system, i.e., one empty chunk in the primary chunk server and two empty chunks each in the secondary replicas chunk servers.

The message handler for Create operation is defined in the Client.da as:

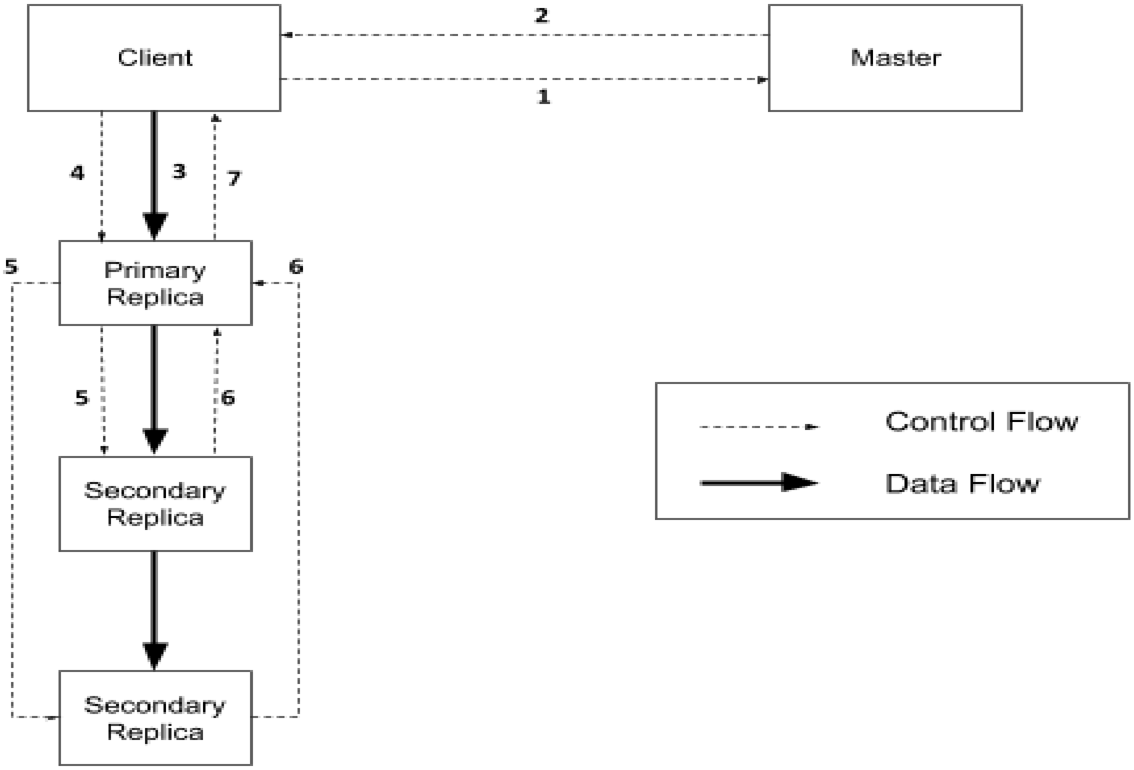
def receive(msg=('CREATE_MSG', fileName))

When messages corresponding to **'CREATE_MSG'** arrive at the Client server, a Create operation is performed. The first check performed is whether or not a file with the same file name already exists on the file system. If it does, then an error message is sent back to the user of the type (**'ERROR_CREATE'**, message), where the message is 'File error, file exists already'. If the file does not exist in the file system, then the control flow for the create operation proceeds further. Next, an **'ALLOC_MASTER'** message is sent to the message handler in Master server and the client waits for an **'ALLOC_DONE'** message from the Master. The details about functioning of **'ALLOC_MASTER'** message is explained in detail in Section 3.2.2 of this report.

Once the Client receives the **‘ALLOC_DONE’** acknowledgement message with *chunkuuid* list, *chunkServerStore* list and *chunkStore* list from the Master (metadata about file to be created), it then calls the *writeChunks()* internal method within Client.da file. This method will take the received chunkuuids, *chunkServerStore* and *chunkStore* lists from the acknowledgement message, an empty string as *dataToWrite* and *fileName* of file to be created as parameters.
Within this *writeChunks()* method, currently since the string is empty, only one empty chunk is created and written on the respective Chunk server. The Master server also gets its *lastChunkOfFileStore* map updated through **‘UPDATE_LAST_CHUNK_DATA’**. These functions are explained in more detail in the section 3.2.4 related to Write operation, since the same functions are used in its case too.

Once an empty chunk is created in the respective Chunk Server as well as its replicas, the message handler for **‘CREATE_MSG’** will send an acknowledgement message of the form (**‘CREATED’**, fileName) to the user, indicating the end of the operation. In case an error is encountered anywhere in between the operation being performed (other than the file exists error), an error message is sent back to the user stating what error has occurred with the message type (**‘ERROR_CREATE’**, message), where the message is ‘Error while creating file’. Message will hold the respective message indicating what the error is. Also if an error occurs after respective maps have been populated, then these changes will be rolled back and corresponding chunkuuids will be added for garbage collection.

3.2.4. Write Operation



The figure above shows the flow in case of a Write operation. This section provides an in-depth explanation for the same. The Write operation writes to a newly created file. If user tries to write to a file which has already been written to or for which data has been appended to, then user will receive a message asking him/her to use Record Append operation.

The message handler for Write operation is defined in the Client.da as:

def receive(msg=('WRITE_MSG', fileName, dataToWrite))

When messages corresponding to **'WRITE_MSG'** arrive at the Client server, a Write operation is performed. The first check performed is whether or not a file with the required fileName is present in the file system or not. If it does not exist, then an error message is sent back to the user of the type (**'ERROR_WRITE'**, message) where the message is 'File error, file doesnt exists'. If the file does exist in the file system, then the control flow for the Write operation proceeds further and next, an **'ADD_TO_LEASE_Q'** message is sent to the message handler function in Master server and the client awaits till it receives **'CLIENT_TURN_IN_LEASE_Q_WRITE'** acknowledgement message from the Master. The explanation regarding these messages have been explained in section 3.2.2 of the report.

Once the waiting Client receives acknowledgement from the Master that it is the turn of the Client to perform the Write operation on a particular file, it sends a **'LATEST_CHUNK_DATA'** message to the Master server. The message handler on the Master server is:

def receive(msg = ('LATEST_CHUNK_DATA', fileName, c), from_=client)

This message handler will simply return the details related to the final chunk of the file stored in the file system through the acknowledgement message **'LATEST_CHUNK_DATA_DONE'**. Upon receiving this acknowledgement, the client then checks whether the latest chunk is of size zero. If the last chunk size is not zero, then it means that after its creation, Record Appends or Writes have already occurred by the users and an error message is sent back to the user of the type (**'ERROR_WRITE'**, message) where the message is 'File is already written, use record append'. Hence, Write functionality in our implementation will only work directly after create, provided there is no Writes or Record Appends on the same file from any user. Once some data has been written into a new file (either from Write or Record Append), further modifications to the data in the file can only be performed through Record Append operations.

If the last chunk size obtained from the Master is zero, then **'ALLOC_MASTER'** message is called with *isCreate* parameter as False. The number of chunks required is calculated based on the chunk size set and the data to be written. **'ALLOC_MASTER'** message is sent to the message handler in Master server and the client waits for an **'ALLOC_DONE'** message from the Master.

Once it receives the **'ALLOC_DONE'** acknowledgement, *writeChunks(chunkuuids, dataToWrite, chunkServerStore, chunkStore, fileName)* method is called. This function performs the main write operations as described in the paper [1], section 3.1. The following are the seven steps as described in the figure above:

1. Request message from Client to Master: This is done during the **'ALLOC_MASTER'** message sent to the Master.

2. Acknowledgement message from Master to Client: This is done with the **'ALLOC_DONE'** message received by the Client.

3. The client pushes the data to all the replicas [1]: This is done within the *writeChunks()* method, where **'STORE_DATA'** message is sent to the nearest Chunk server (can be either Primary or secondary replicas) along with the data to be written. This message when received by the individual Chunk Server, will store the data in local storage as well as send **'STORE_DATA'** message to the next Chunk Server (again, can be primary or secondary replicas), where the same operation happens. Once the data has been propagated to the last replica, all the chunk servers will return **'STORE_DATA_DONE'** acknowledgement messages after which the data flow will be done (solid arrows in the figure).

4. Once all the replicas have acknowledged receiving the data, the client sends a Write request to the Primary [1]: This is done by sending **'WRITE_CHUNK'** message to the primary replica chunk server. The message handler for **'WRITE_CHUNK'** is:

   def receive(msg= ('WRITE_CHUNK',chunkuuid, chunkloc, chunkServerLocal, c, toAppend),from_ = sender)

5. The primary forwards the Write request to all secondary replicas [1]: This is done within the message handler for the message **'WRITE_CHUNK'**. **'WRITE_CHUNK_SECONDARY'** messages are passed to each of the secondary replicas chunk servers. Upon receipt of this message, the secondary replicas will use the data earlier stored in local storage (in step 3), and write to the respective chunk file replicas.

6. The secondaries all reply to the primary indicating that they have completed the operation: This is done using the acknowledgement messages **'WRITE_CHUNK_SECONDARY_DONE'**. Upon receipt of messages from all secondary replicas, the primary replica will also commit the data held by it in local storage.

7. The primary replies to the client [1]: This is done using the acknowledgement message **'WRITE_CHUNK_DONE'**.

Once the **'WRITE_CHUNK_DONE'** acknowledgement is received by the client, it sends a **'UPDATE_LAST_CHUNK_DATA'** message to the master, where the *lastChunkOfFileStore* map is updated with the last chunkuuid of the file as well as size of data written in that chunk. This is used for Record Append or Write operations (this **'UPDATE_LAST_CHUNK_DATA'** message is also called during Create operation, in which case the size is referred to during **‘LATEST_CHUNK_DATA_DONE’** of Write or Record Append operation).

This call to function *writeChunks()* method occurs during Create, Write and Record Append operations. During Create, the *dataToWrite* parameter is passed empty, which creates an empty chunk. In case of both other operations, *datatoWrite* contains data to be written onto the file.

In case an error occurs at any place during execution of Write operation, the entire process which has taken place till the error occurs is rolled back (by sending **‘ERROR_WRITE’** message to Master server, where the receiver handler in Master cleans up half written files) and an error message is passed back to the user of the type (**‘ERROR_WRITE’**, message), where message sent is ‘Error writing file’.

Within the finally section of the message handler, **‘PROCESS_NEXT_IN_LEASE_Q’** message is sent which will trigger the next operation in queue to be processed.

3.2.5. Record Append Operation

The Record Append operation performs appends of passed data to the end of a given existing file.

The Record Append receive handler in Client.da is defined as follows:

def receive(msg= ('RECORD_APPEND_MSG', fileName, dataToWrite))

Similar to all previous operations, the first check performed is to check whether the file exists or not. If the fileName is not present in the file system, an error message is passed back to the user of the type (**‘ERROR_APPEND’**, message) where the message sent is ‘File Doesn't Exist to do record append’.

If the file exists, then client sends **‘ADD_TO_LEASE_Q’** message and waits for acknowledgement message **‘CLIENT_TURN_IN_LEASE_Q_APPEND’** from the Master server. Upon receiving this acknowledgement, the number of chunks required for the data to be appended is calculated.

If the required number of chunks is greater than 1, then the control flow follows logic similar to that of Write operation - sending **‘ALLOC_APPEND_MASTER’** message and waiting for **‘ALLOC_APPEND_DONE’** followed by calling *writeChunks()* method. The working of **‘ALLOC_APPEND_MASTER’** message is explained in section 3.2.1 and explanation of *writeChunks()* method is given in section 3.2.4.

If the required number of chunks is equal to 1, then it means that the data passed by the user can be appended to the end of the existing chunk in the file, provided the data fits the remaining size of chunk file. This information regarding size of last chunk file is fetched using

‘GET_LAST_CHUNK_DATA’ message to the Master server. The message handler of the Master server is as follows:

def receive(msg = ('GET_LAST_CHUNK_DATA', fileName, c), from_=client)

The Client will wait for acknowledgement **'GET_LAST_CHUNK_DATA_DONE'** from the Master, which will have the required information regarding last chunk of the file obtained from last *lastChunkOfFileStore* map. On receiving last chunk data, if it is found that append data cannot be fit into the last chunk then once again similar procedure as that for Write operation is performed, though only for one chunk to be created and added to the *fileStore* map. However, if the data to be appended can be fit into the last chunk of the file, then *addToExistingChunkFile()* method is called within Client.da file.

Within the *addToExistingChunkFile()* method, **‘STORE_DATA’** and **‘WRITE_CHUNK’** messages which are required for writing to a chunk file (steps 3 to 7 explained in Write operation) are directly sent using the information obtained from **'GET_LAST_CHUNK_DATA_DONE'** acknowledgement. Thus, allocation of chunk for this particular data to be appended can be skipped in case of appends. It can also reduce internal fragmentation, since most Record Append operations are generally small enough to be fit in the remaining size of the last chunk of the file.

In case an error occurs at any place during execution of Write operation, the entire process which has taken place till the error occurs is rolled back (by sending **‘ERROR_RECORD_APPEND’** message to Master server, where the receiver handler in Master cleans up half written files) and an error message is passed back to the user of the type (**‘ERROR_APPEND’**, message), where message sent is ‘Error while record append + <Exception which occurred>’.

Within the finally section of the message handler, **‘PROCESS_NEXT_IN_LEASE_Q’** message is sent which will trigger the next operation in queue to be processed.

3.2.6. Read Operation

Read operation is used to read the contents of the file.

The read receive handler in Client.da is defined as follows:

def receive(msg= ('READ_MSG',clock, fileName))

When a **‘READ_MSG’** is received, along with the fileName, at first a check is performed to see if a file exists with the fileName that’s received with the **‘READ_MSG’** message. If no such file exists, the client responds to the parent process with (**‘ERROR_READ’**, message) where the message sent is 'File Doesn’t Exist to read'.

If the file exists, then Client sends **‘READ_MASTER’** message to master, and the receive handler in the master is defined as:

def receive(msg = ('READ_MASTER', fileName,c), from_=client)

Master upon receiving this message, gets the chunkuuids from FileStore and the chunk location from chunkStore and returns them to the client with **‘READ_MASTER_DONE’** message.

The client upon receiving the **‘READ_MASTER_DONE’** message from the master, will send a **‘READ_CHUNK’** message to all the chunk servers available and then awaits a reply from one of the chunk servers.

If any of the available chunk servers replies client with (**'READ_CHUNK_DONE'**,chunk,c2), we append the data received to a list and send the request to all chunkservers for the next chunkuuid. Once we have received the data for all chunkuuids, then all the received data is appended to one, and is returned to the user as the *dataRead* along with **‘READ_FILE_DONE’** acknowledgement message.

3.2.7. Delete Operation

The Delete operation is meant so that the user cannot read/record append to this file. The delete process doesn’t remove the physical chunks right aways, but rather marks them that these chunks need to be deleted during the next garbage collection cycle.

The Delete receive handler in client.da is defined as follows :

def receive(msg= ('DELETE', fileName))

When a **‘DELETE’** message along with the fileName is received at the Client Server, at first, a check is performed to see if any file exists with the fileName that’s received with the **‘DELETE’** Message. If no such file exists, the client respond to the parent process with (**‘ERROR_DELETE’**, message) where the message sent is ‘File Doesn’t Exist to Delete’

If the file exists, then Clients sends **‘DELETE_FILE_MASTER’** message to master and the receive handler for it is defined as:

def receive(msg = ('DELETE_FILE_MASTER', fileName, c), from_=client)

Master upon receiving this message, calls *addToGarbageCollection()* method with the list of all chunkuuid’s mapped to this fileName from fileStore. This method adds the chunkuuids to the corresponding chunkserver in which it is stored, in *deletedChunksList* map which is later used for garbage collection. If there is entry in *snapShotuuids* for the file, we call addToGarbageCollection with the *chunkuuids* from *snapShotuuids*. Then we delete the entry for the filename from fileStore and *snapShotuuids*.

Once this is done, Master acknowledges to client with the message **'DELETE_FILE_MASTER_DONE'** and the client responds back to the user with the acknowledgement **'DELETE_DONE'**. - *ref[1] Section 4.4.1*

3.2.8. Snapshot Operation

The Snapshot receive handler is defined in Client.da as follows:

def receive(msg= ('CREATE_SNAPSHOT', fileName))

Snapshot operation creates a copy of the file almost instantaneously with minimal interruptions on the ongoing operations. When messages corresponding to **'CREATE_SNAPSHOT'** arrive at the Client server, a Snapshot operation is performed. The first check performed is whether or not a file with the same file name already exists on the file system. If it does not, then an error is thrown back to the user stating that the particular file does not exists. If the file exists in the file system, then the control flow for the create operation proceeds further and a **'ADD_TO_LEASE_Q'** message is sent to the Master server.

The paper states that the Snapshot process should get priority over all other processes. To provide priority to it, we add it to the lease queue at index 1 if atleast one other entry is in the lease queue. This allows for the current process to execute its operations and the next operation to be performed on the file will be Snapshot.

Once the Snapshot process obtains the lease on the file, it sends a **'READ_MASTER'** request for the file. Master provides the client with the chunkuuids and its storage locations. Client sends each of the chunkuuids at the various locations a **'WRITE_CHUNK_SNAPSHOT'** message which is defined as follows:

def receive(msg = ('WRITE_CHUNK_SNAPSHOT', chunkuuid, c), from_=client)

The chunk servers, read the chunk corresponding to the passed chunkuuid and create a local copy of the chunk naming it chunkuuid-SNAPSHOT. If there is an error while writing or chunk is not written we send an error back to the user which has the type (**'ERROR_SNAPSHOT'**, message).

If the snapshot gets created, then the Client sends an **'UPDATE_SNAPSHOT'** request to master which is defined as:

def receive(msg=('UPDATE_SNAPSHOT', fileName, c), from_=client)

This handler updates the metadata for that file stored in master, to point to the new versions created by snapshot process. It updates the lastChunkOfFileStore, fileStore and chunkStore.

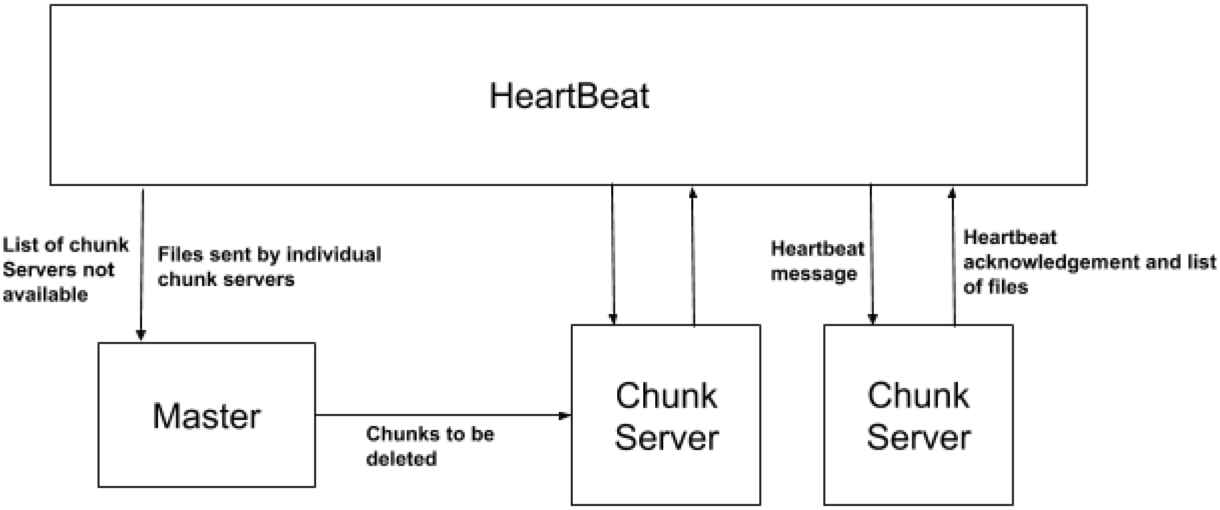
It also updates/creates a new entry in the *snapShotuuids* to add the previous chunkuuids for this file (so that they could be removed, once the file is deleted). Once this is done Master replies with a **'UPDATE_SNAPSHOT_DONE'** message for that file, at which point the Snapshot process is complete. Before exiting the function it sends a message to master to process the next process in the lease queue for that file. After snapshot, any further mutations/read will happen from the new snapshot chunks. *-ref[1] Section 3.4*

3.2.9. Heartbeat Operation

Heartbeat is a very important process in GFS. It serves two main purposes:

- Checking which chunk servers are alive
- Garbage collection

- *ref[1] Section 4.4*


HeartBeat
List of chunk Servers not available
Files sent by individual chunk servers
Heartbeat message
Heartbeat acknowledgement and list of files
Master
Chunks to be deleted
Chunk Server
Chunk Server

Checking which chunk servers are alive

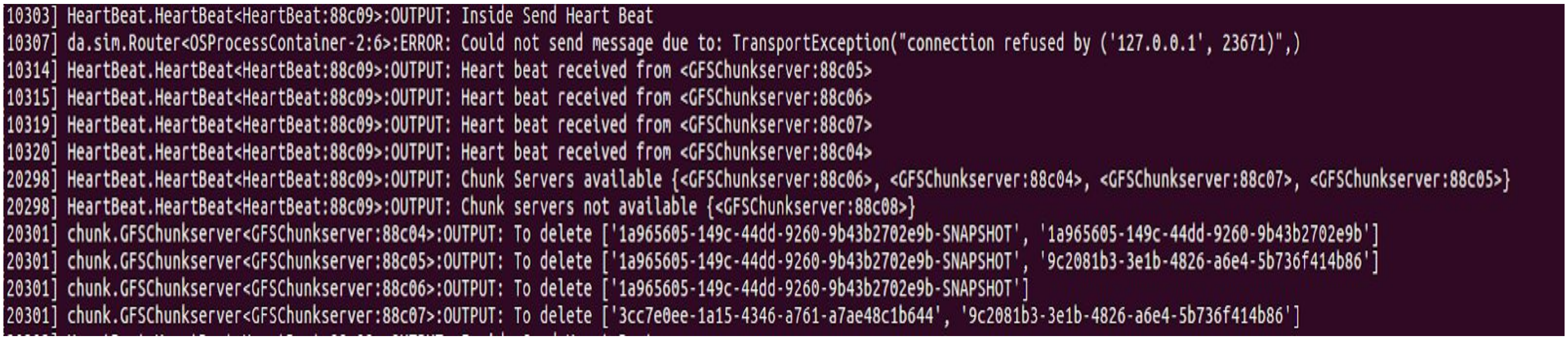
In our implementation at regular intervals(10 secs for now), Heartbeat process sends a **'HEARTBEAT'** message to all chunk servers. The receive handler on chunk server side is defined as:

def receive(msg =('HEARTBEAT',), from_= heartBeat)

It waits till it receives response from all chunk servers or till the message timeout. If it receives acknowledgement from all chunk servers we log "all chunk servers are responding' and move on to garbage collection. If a chunkserver fails to respond we send a message to master informing which chunk servers are not responding, so that master can update the non Available Servers list and take it into consideration while allocating chunks. The receive handler on master side is as follows:

def receive(msg = ('UPDATE_AVAILABLE_CHUNK_SERVERS', nonAvailableChunkServers), from_= heartBeat):

Here the master updates the list of *nonAvailableChunkServers* which is used by master while allocating chunk for new writes/ record append, garbage collection and other operations.



```
10303] HeartBeat.HeartBeat<HeartBeat:88c09>:OUTPUT: Inside Send Heart Beat
10307] da.sim.Router<OSProcessContainer-2:6>:ERROR: Could not send message due to: TransportException("connection refused by ('127.0.0.1', 23671)",)
10314] HeartBeat.HeartBeat<HeartBeat:88c09>:OUTPUT: Heart beat received from <GFSChunkserver:88c05>
10315] HeartBeat.HeartBeat<HeartBeat:88c09>:OUTPUT: Heart beat received from <GFSChunkserver:88c06>
10319] HeartBeat.HeartBeat<HeartBeat:88c09>:OUTPUT: Heart beat received from <GFSChunkserver:88c07>
10320] HeartBeat.HeartBeat<HeartBeat:88c09>:OUTPUT: Heart beat received from <GFSChunkserver:88c04>
20298] HeartBeat.HeartBeat<HeartBeat:88c09>:OUTPUT: Chunk Servers available {<GFSChunkserver:88c06>, <GFSChunkserver:88c04>, <GFSChunkserver:88c07>, <GFSChunkserver:88c05>}
20298] HeartBeat.HeartBeat<HeartBeat:88c09>:OUTPUT: Chunk servers not available {<GFSChunkserver:88c08>}
20301] chunk.GFSChunkserver<GFSChunkserver:88c04>:OUTPUT: To delete ['1a965605-149c-44dd-9260-9b43b2702e9b-SNAPSHOT', '1a965605-149c-44dd-9260-9b43b2702e9b']
20301] chunk.GFSChunkserver<GFSChunkserver:88c05>:OUTPUT: To delete ['1a965605-149c-44dd-9260-9b43b2702e9b-SNAPSHOT', '9c2081b3-3e1b-4826-a6e4-5b736f414b86']
20301] chunk.GFSChunkserver<GFSChunkserver:88c06>:OUTPUT: To delete ['1a965605-149c-44dd-9260-9b43b2702e9b-SNAPSHOT']
20301] chunk.GFSChunkserver<GFSChunkserver:88c07>:OUTPUT: To delete ['3cc7e0ee-1a15-4346-a761-a7ae48c1b644', '9c2081b3-3e1b-4826-a6e4-5b736f414b86']
```

Garbage collection

In every Heartbeat message the chunk server along with the acknowledgement sends a subset of the chunks on its storage(5 for now). It can be configured by changing the parameter *heartBeatNumFiles* in chunk.da. The receive handler on HeartBeat.da is defined as follows:

def receive(msg=('HEARTBEAT_ACKNOWLEDGED',files), from_=chunkServer):

The handler appends the *chunkuuid* list(files) obtained to the map *filesList* which maps *chunkServer* to list of *chunkuuids* sent by it. The heartbeat process sends the *filesList* map to master. After sending *filesList* map to master it resets the *chunkuuids* list to empty list for each chunkServer. The receive handler on master side is defined as follows:

def receive(msg = ('DELETE_ORPHANED_CHUNKS', fileList), from_=heartBeat):

The master looks into its metadata store and checks which of these chunkuuid have been deleted and sends to the chunk server that these chunkuuid needs to be deleted from their storage. We can configure the interval after which we want garbage collection to happen. At present we have configured it so that it runs on every cycle but we can change it by varying the parameter *self.timeForGarbage* in the HeartBeat.da file. Above is a small snippet showing the message when chunk servers receive message to delete the chunks.

- *ref[1] Section 4.4*

4. Testing and Execution

4.1. Execution

The source code is present in gfs-distalgo directory . All the source files and test files must be present in the same directory and the DistAlgo package should have been extracted into this directory. Note that all code is written to be compatible with Python 3 and hence, should be executed using Python 3.x, preferably version 3.6(updated)

Instructions to run the code:

To run main process:

python3 -m da main.da

Main.da file was used for testing during our local development. It contains the message formats for the different operations that we support and a complete end to end flow with different operations.

To run Correctness Testing:

python3 -m da test.da <testCase Number> (testCases are numbered from 0-16)

Example: python3 -m da test.da 4

TestCase Number mapping with TestCase (Negative test case means failure of operation and positive test case means operation working as expected)

0 = Create - Negative Test Case
1 = Write - Negative Test Case
2 = Delete - Negative Test Case
3 = Read - Negative Test Case
4 = RecordAppend - Negative Test Case
5 = Snapshot - Negative Test Case
6 = Create - Positive Test Case
7 = Write - Positive Test Case
8 = Delete - Positive Test Case
9 = Read - Positive Test Case
10 = RecordAppend - Positive Test Case

11 = Snapshot - Positive Test Case

12 = End-To-End Testing (Create, Write, RecordAppend, RecordAppend, Snapshot, Snapshot, Read, Delete)

13 = End-To-End Testing - Multiple Reads (Create, Create, Write, Snapshot, Read, Read, Read, Read)

14 = End -To-End Testing (Create, Create, Write, RecordAppend, Snapshot, Read, Read, Delete)

15 = Consistency Testing -(Multiple Record Appends,Snapshot)

16 = Availability Testing - (Reading file before and after killing one of the chunkserver)

To run Performance Testing:

```
python3 -m da PerformanceTesting.da <NumberOfClients> <TestCaseNumber>
```

TestCase Number mapping with TestCase

1 = This test case will create the user entered number of clients, create a single file, write a 670 character string to it and will try to **read** the same file from each of the created clients (N concurrent reads)

2 = This test case will create the user entered number of clients, create user entered number of files, and try to **write** the 679 character string into each of the created file using each of the client created (N concurrent writes)

3 = This test case will create the user entered number of clients, create a single file, and **append** 670 character string into the same file from the user entered number of clients (N concurrent Record Appends)

Example: python3 -m da PerformanceTesting.da 500 2 → Running this command will run 500 concurrent Writes from 500 clients

4.2. Performance Testing

We have performed performance testing using our implementation for the following cases - when N clients try to perform a Record Append operation, when N clients try to perform a Write Operation, and when N clients try to perform a Read operation - each with different sizes for the chunks to be stored in the Chunk server. *- ref[1] Section 6.1*

The different test cases performed had the following constant parameters -

Number of Chunk Servers = 5

Size of input string used in all cases = 679 bytes

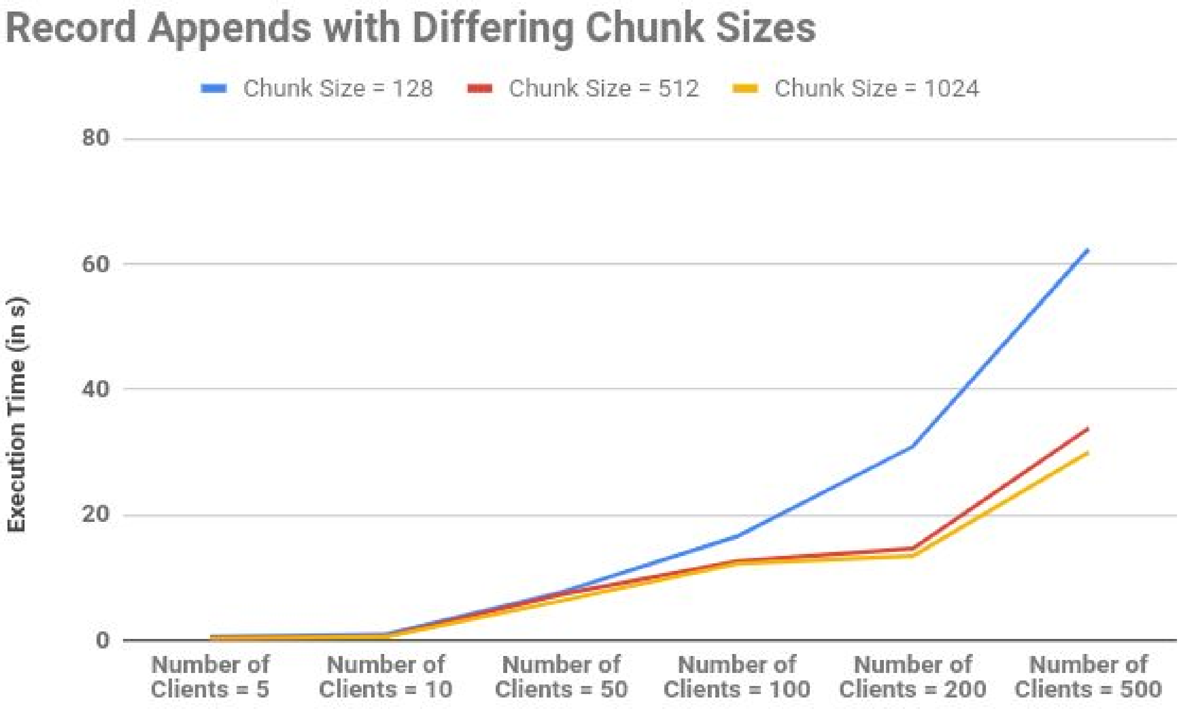
Number of replicas used for each chunk = 3

We have also created a Process Monitor (ProcessMonitor.da) which will capture and calculate the execution times required for performance testing.

4.2.1. Results for N clients performing concurrent Record Appends

The following table lists the results obtained by varying number of clients along with chunk size to perform Record Append operations of a given set string, i.e., a string of 679 characters was appended to a file by each of the N clients performing the Record Append operation. Three different trials were performed for each iteration and the obtained execution time for all three trials were averaged out. The table has also been graphically represented.

Number of Clients	Chunk Size	Record Append Trial 1 Execution Time (in s)	Record Append Trial 2 Execution Time (in s)	Record Append Trial 3 Execution Time (in s)	Average Execution Time Taken Over 3 Trials (in s)
5	128	0.57	0.58	0.568	0.572
10	128	0.977	0.964	0.918	0.953
50	128	8.036	7.328	7.615	7.659
100	128	17.261	16.359	16.169	16.596
200	128	29.983	32.346	30.475	30.934
500	128	62.524	62.378	62.388	62.43
5	512	0.321	0.324	0.333	0.326
10	512	0.635	0.654	0.642	0.643
50	512	7.324	7.464	7.148	7.312
100	512	12.58	12.726	12.504	12.603
200	512	14.279	14.744	14.982	14.668
500	512	33.498	34.125	33.692	33.771
5	1024	0.288	0.286	0.29	0.288
10	1024	0.579	0.582	0.568	0.576
50	1024	6.295	6.364	6.319	6.326
100	1024	12.218	12.222	12.184	12.208
200	1024	13.874	13.53	12.838	13.414
500	1024	29.976	30.158	29.889	30.007

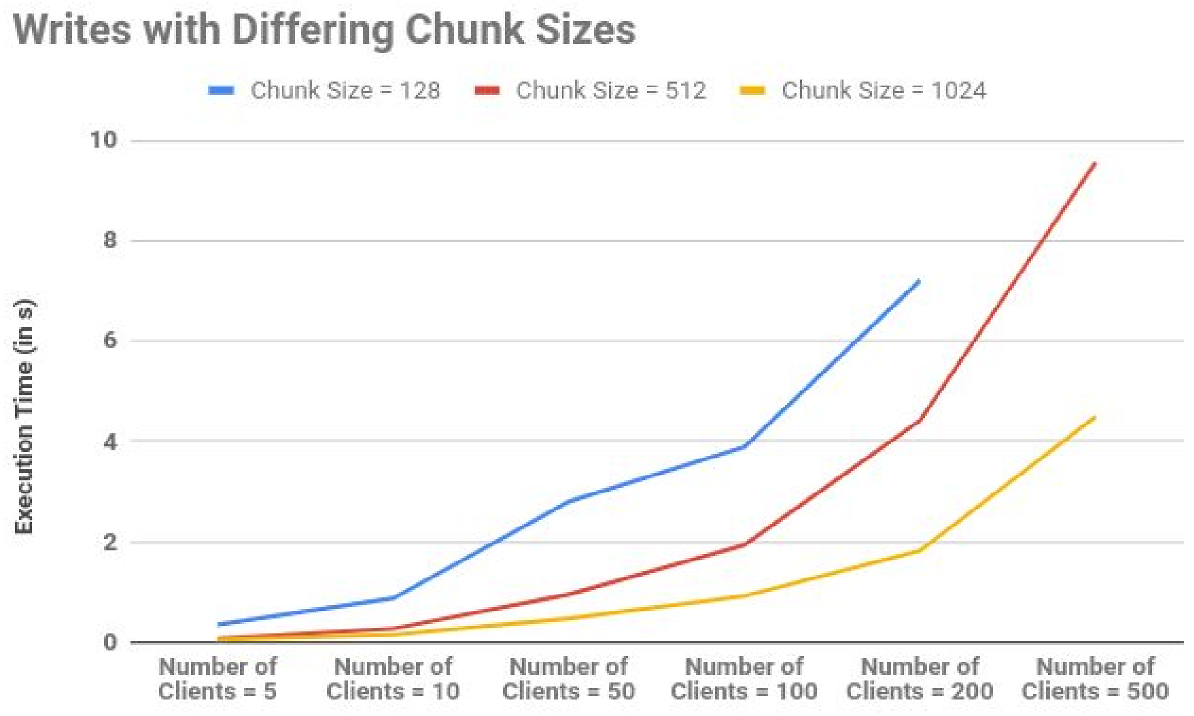


The results show that with increase in number of clients, the execution time taken for N Record Append operations also increases, which is pretty intuitive. The interesting result, however, is that with increase in chunk size per chunk, the execution time decreases considerably. As can be seen in the graph above, the execution time is pretty high when the chunk size used is 128, as compared to when the chunk size is 1024. We relate this behavior to the fact that when the chunk size is small, a Record Append operation of a large string causes many chunks (as well as their replicas) required to be created/updated, since a large string may not fit into a single chunk. This requires a large number of file creates and writes on every chunk servers, thus causing an increase in the execution time for small chunks. However, for a large chunk file, multiple record appends can take place in a single already existing file, thus lowering the internal file system operations required on the individual chunk servers. It can also be seen from the graph, that this effect is more pronounced as the number of clients increases, i.e., with small number of clients, execution time variation between using different chunk sizes is minimal; however, with large number of clients, execution time variation between different chunk sizes varies considerably.

4.2.2. Results for N clients performing concurrent Writes

The following table lists the results obtained by varying number of clients along with chunk size to perform Write operations of a given set string, i.e., a string of 679 characters was written to a newly created file by each of the N clients performing the Write operation. Three different trials were performed for each iteration and the obtained execution time for all three trials were averaged out. The table has also been graphically represented.

Number of Clients	Chunk Size	Write Trial 1 Execution Time (in s)	Write Trial 2 Execution Time (in s)	Write Trial 3 Execution Time (in s)	Average Execution Time Taken Over 3 Trials (in s)
5	128	0.405	0.33	0.318	0.351
10	128	0.883	0.859	0.886	0.876
50	128	2.407	2.983	3.014	2.801
100	128	3.856	3.912	3.929	3.899
200	128	7.298	6.893	7.457	7.216
5	512	0.07	0.068	0.068	0.068
10	512	0.286	0.27	0.257	0.271
50	512	1.004	0.909	0.956	0.956
100	512	1.859	1.958	2.001	1.939
200	512	4.702	4.231	4.34	4.424
500	512	9.815	9.71	9.168	9.564
5	1024	0.053	0.057	0.053	0.054
10	1024	0.116	0.161	0.17	0.149
50	1024	0.454	0.464	0.497	0.471
100	1024	0.93	0.941	0.89	0.920
200	1024	1.868	1.802	1.797	1.822
500	1024	4.461	4.558	4.485	4.501



The results show similar behavior as that of N Record Append operations. The execution time increases with increase in number of clients and decreases with increase in chunk size. The reasoning behind this behavior is once again similar to that of Record Append operations, since larger chunk size results in fewer files needed to be created. The only anomaly occurred in the execution of Write operations by 500 clients with a chunk size of 128. We believe this anomaly occurred due to extremely large number of chunks required to be written, which caused overload in the network buffer. Despite this anomaly, the trend of the curves can be seen clearly from the graphs.

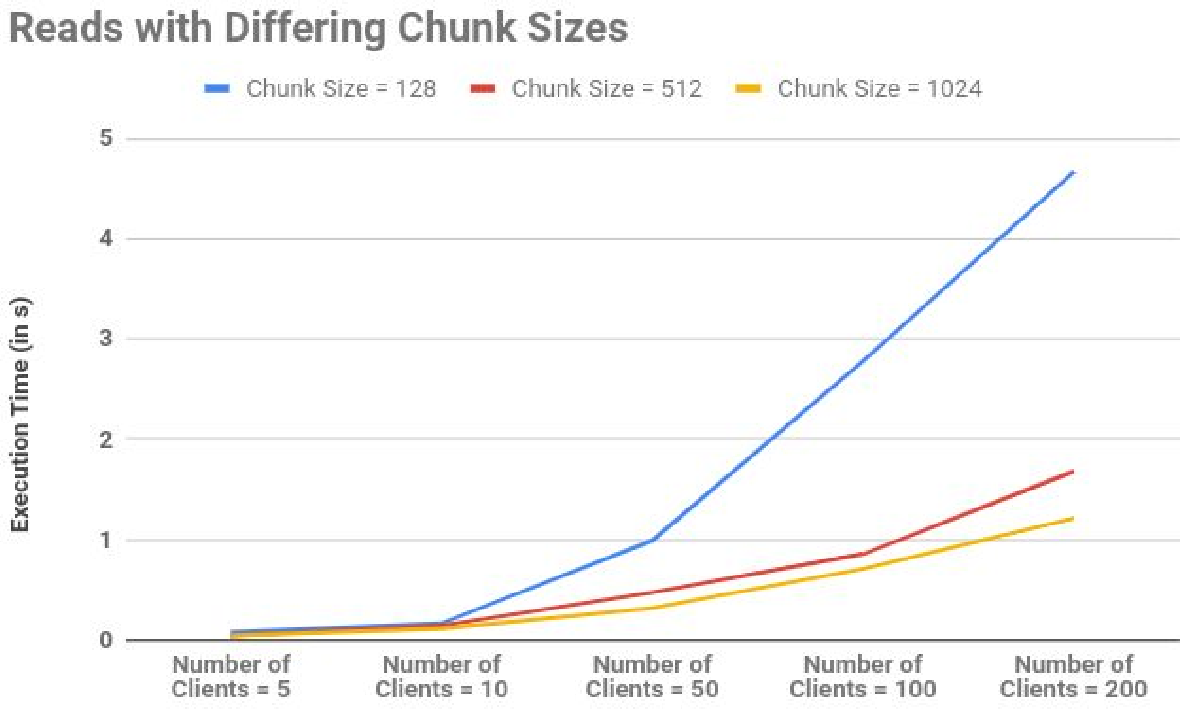
4.2.3. Record append operations Vs. Write operations

As can be seen in the graphs, the execution times for Record Appends in much higher than that for Writes. The reason for this discrepancy is that in case of a Record Append operation, there are multiple checks performed with regards to available space within the existing chunk which increases the time. In case of a Write operation, the string is directly written into the file. The other major reason is that in case of a Write operation, every file written is separate, whereas for a Record Append operation, the same file is appended to. This means that in case of a Write operation, the entries in the lease queue are separate for separate files created, while for a Record Append operation, entries are added to the same lease queue, which means that the concurrent Record Append operations internally operate sequentially on one file in the system, whereas the Write operations operations occur parallelly between multiple files in the system.

4.2.4. Results for N clients performing concurrent Reads

The following table lists the results obtained by varying number of clients along with chunk size to perform Read operations of a given set string, i.e., a string of 679 characters was written to a newly created file and this file was read by N clients concurrently. Three different trials were performed for each iteration and the obtained execution time for all three trials were averaged out. The table has also been graphically represented.

Number of Clients	Chunk Size	Read Trial 1 Execution Time (in s)	Read Trial 2 Execution Time (in s)	Read Trial 3 Execution Time (in s)	Average Execution Time Taken Over 3 Trials (in s)
5	128	0.078	0.079	0.076	0.077
10	128	0.159	0.193	0.158	0.17
50	128	0.919	1.027	1.037	0.994
100	128	2.97	2.487	2.897	2.784
200	128	5.164	4.416	4.442	4.674
5	512	0.046	0.043	0.045	0.044
10	512	0.146	0.138	0.15	0.144
50	512	0.584	0.415	0.427	0.475
100	512	0.816	0.826	0.931	0.857
200	512	1.74	1.665	1.645	1.683
5	1024	0.044	0.041	0.049	0.044
10	1024	0.124	0.111	0.105	0.113
50	1024	0.301	0.322	0.336	0.319
100	1024	0.706	0.842	0.582	0.71
200	1024	1.236	1.213	1.198	1.215



Once again the behavior displayed is similar to the above two test cases. Small chunk sizes result in higher execution times since larger number of different chunk files must be read to obtain the required data from the file. However, we were unable to run read operations with 500 clients with any chunk size, due to stack overflow errors caused by system limitations. We were able to observe the trends of the varying parameters using the remaining client sizes.

4.2.5. Conclusions From Performance Testing

One of the major conclusions we have come to after analyzing the performance tests is that of the chunk size. We have observed that larger chunk sizes increase the execution speeds of different operations. Even though such large chunk file sizes may cause internal fragmentation, the performance gain trade-offs far offsets the fragmentation concerns. It is for this reason that Google recommends using chunk file size of 64 MB.

4.3. Correctness Testing

We have implemented Availability, Consistency, Functionality testing on the File System Operations of this project. In total, we have implemented total 17 test cases which includes positive and negative test cases for Create, Read, Delete, RecordAppend, Write, SnapShot, End-to-End Testing of FOps, Consistency and Availability Testing. Below is the brief description of few of the test cases we have implemented.

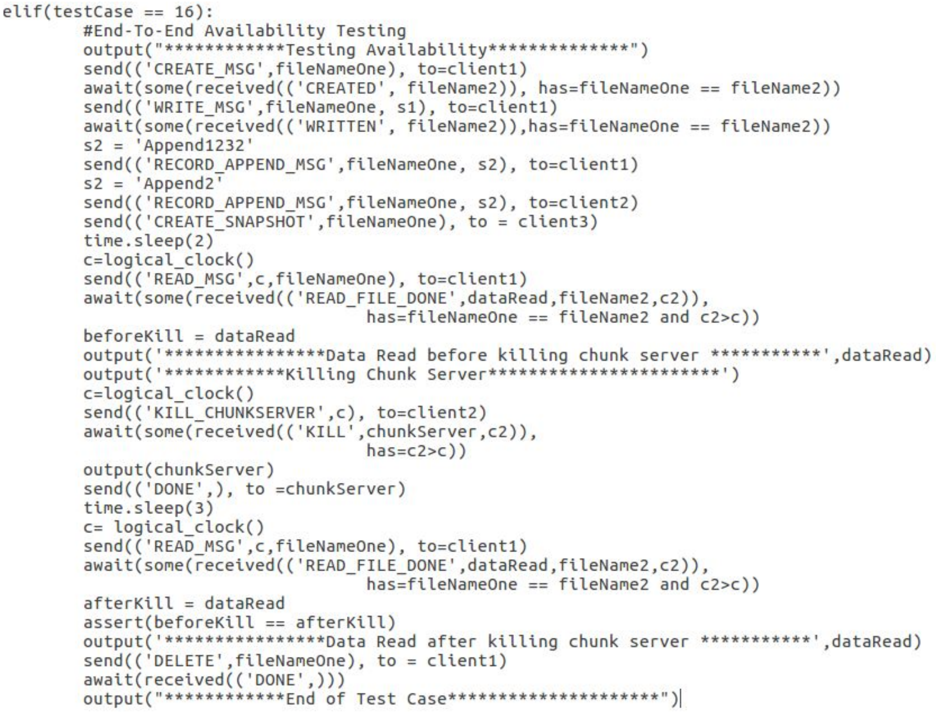
4.3.1. Availability

We tested Availability by reading the data from file before and after killing one of the chunk servers.

We created the file, and have done write and read operations on it.

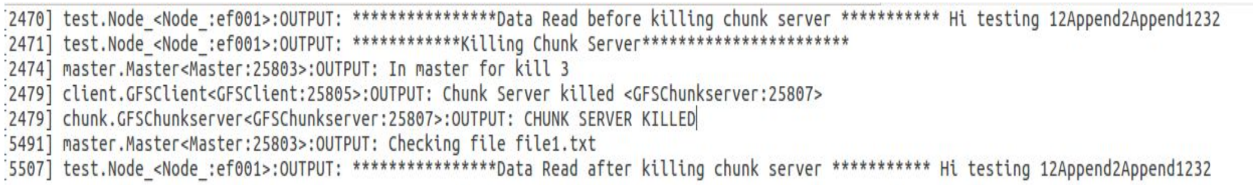
We then killed one of the chunk servers,and did read operation on the file. The data read before and after killing chunk server is compared and observed to be the same.

At any time, if one chunk server is down, we will still be able to retrieve data from other available servers,hence the data is not lost.



```
elif(testCase == 16):
        #End-To-End Availability Testing
        output("************Testing Availability**************")
        send(('CREATE_MSG',fileNameOne), to=client1)
        await(some(received(('CREATED', fileName2)), has=fileNameOne == fileName2))
        send(('WRITE_MSG',fileNameOne, s1), to=client1)
        await(some(received(('WRITTEN', fileName2)),has=fileNameOne == fileName2))
        s2 = 'Append1232'
        send(('RECORD_APPEND_MSG',fileNameOne, s2), to=client1)
        s2 = 'Append2'
        send(('RECORD_APPEND_MSG',fileNameOne, s2), to=client2)
        send(('CREATE_SNAPSHOT',fileNameOne), to = client3)
        time.sleep(2)
        c=logical_clock()
        send(('READ_MSG',c,fileNameOne), to=client1)
        await(some(received(('READ_FILE_DONE',dataRead,fileName2,c2)),
                                    has=fileNameOne == fileName2 and c2>c))
        beforeKill = dataRead
        output('****************Data Read before killing chunk server ***********',dataRead)
        output('************Killing Chunk Server***********************')
        c=logical_clock()
        send(('KILL_CHUNKSERVER',c), to=client2)
        await(some(received(('KILL',chunkServer,c2)),
                                    has=c2>c))
        output(chunkServer)
        send(('DONE',), to =chunkServer)
        time.sleep(3)
        c= logical_clock()
        send(('READ_MSG',c,fileNameOne), to=client1)
        await(some(received(('READ_FILE_DONE',dataRead,fileName2,c2)),
                                    has=fileNameOne == fileName2 and c2>c))
        afterKill = dataRead
        assert(beforeKill == afterKill)
        output('****************Data Read after killing chunk server ***********',dataRead)
        send(('DELETE',fileNameOne), to = client1)
        await(received(('DONE',)))
        output("************End of Test Case**********************")
```

Output for the above Testcase:



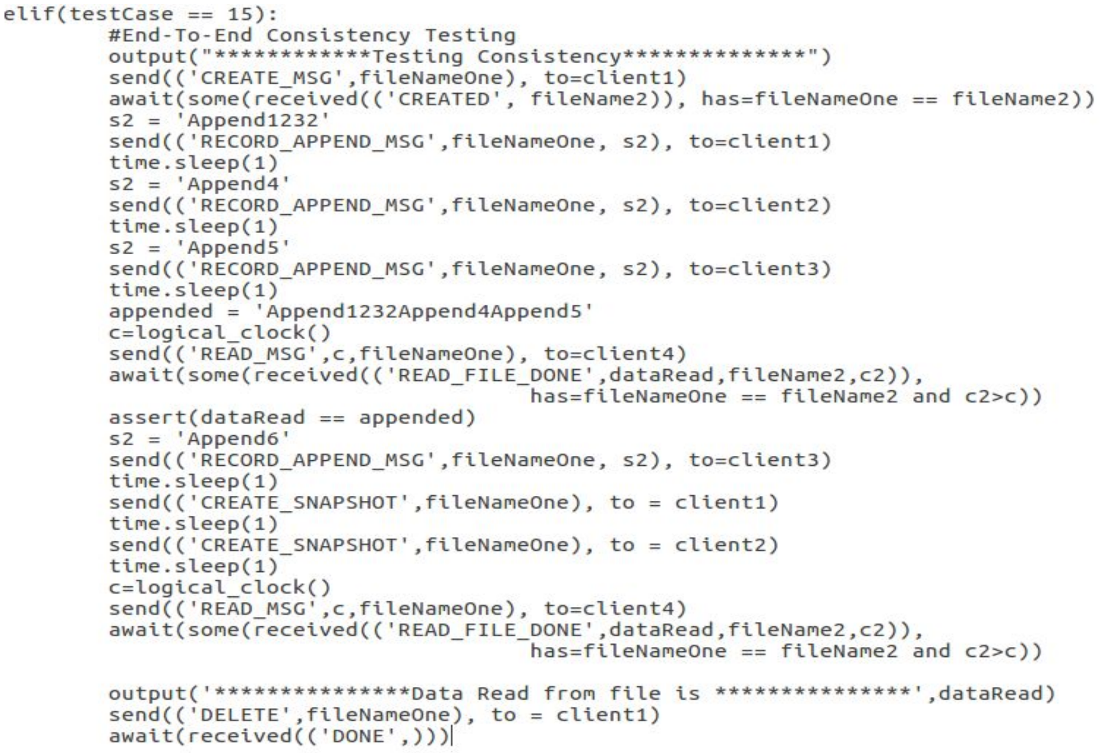
```
2470] test.Node_<Node_:ef001>:OUTPUT: ****************Data Read before killing chunk server *********** Hi testing 12Append2Append1232
2471] test.Node_<Node_:ef001>:OUTPUT: ************Killing Chunk Server**********************
2474] master.Master<Master:25803>:OUTPUT: In master for kill 3
2479] client.GFSClient<GFSClient:25805>:OUTPUT: Chunk Server killed <GFSChunkserver:25807>
2479] chunk.GFSChunkserver<GFSChunkserver:25807>:OUTPUT: CHUNK SERVER KILLED
5491] master.Master<Master:25803>:OUTPUT: Checking file file1.txt
5507] test.Node_<Node_:ef001>:OUTPUT: ****************Data Read after killing chunk server *********** Hi testing 12Append2Append1232
```

4.3.2. Consistency

We have testing consistency on multiple record appends and snapshot operation.

4.3.2.1. Record Appends

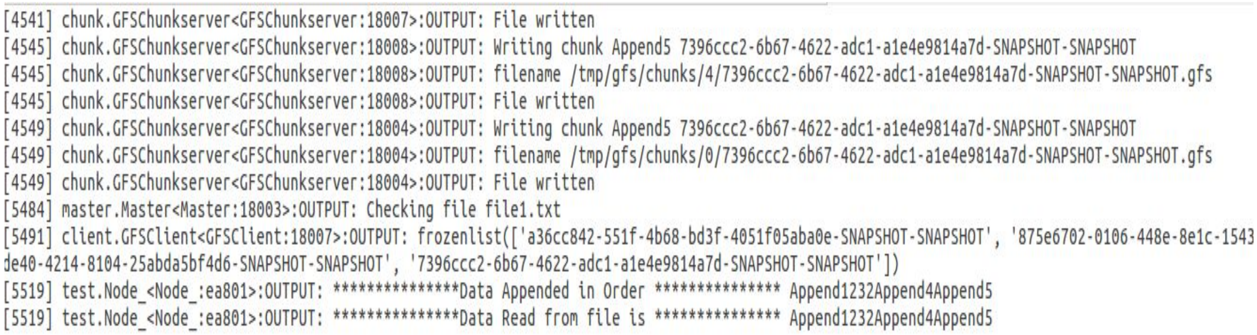
Multiple record appends should be executed in the order they arrive.
We have created a file and executed multiple record appends on it
We have performed read operation from the file and compared the order of data read from the file to the order of data appended to it.
The order of data read and data appended was observed to be the same.



```
elif(testCase == 15):
        #End-To-End Consistency Testing
        output("************Testing Consistency**************")
        send(('CREATE_MSG',fileNameOne), to=client1)
        await(some(received(('CREATED', fileName2)), has=fileNameOne == fileName2))
        s2 = 'Append1232'
        send(('RECORD_APPEND_MSG',fileNameOne, s2), to=client1)
        time.sleep(1)
        s2 = 'Append4'
        send(('RECORD_APPEND_MSG',fileNameOne, s2), to=client2)
        time.sleep(1)
        s2 = 'Append5'
        send(('RECORD_APPEND_MSG',fileNameOne, s2), to=client3)
        time.sleep(1)
        appended = 'Append1232Append4Append5'
        c=logical_clock()
        send(('READ_MSG',c,fileNameOne), to=client4)
        await(some(received(('READ_FILE_DONE',dataRead,fileName2,c2)),
                                        has=fileNameOne == fileName2 and c2>c))
        assert(dataRead == appended)
        s2 = 'Append6'
        send(('RECORD_APPEND_MSG',fileNameOne, s2), to=client3)
        time.sleep(1)
        send(('CREATE_SNAPSHOT',fileNameOne), to = client1)
        time.sleep(1)
        send(('CREATE_SNAPSHOT',fileNameOne), to = client2)
        time.sleep(1)
        c=logical_clock()
        send(('READ_MSG',c,fileNameOne), to=client4)
        await(some(received(('READ_FILE_DONE',dataRead,fileName2,c2)),
                                        has=fileNameOne == fileName2 and c2>c))

        output('***************Data Read from file is ***************',dataRead)
        send(('DELETE',fileNameOne), to = client1)
        await(received(('DONE',)))
```

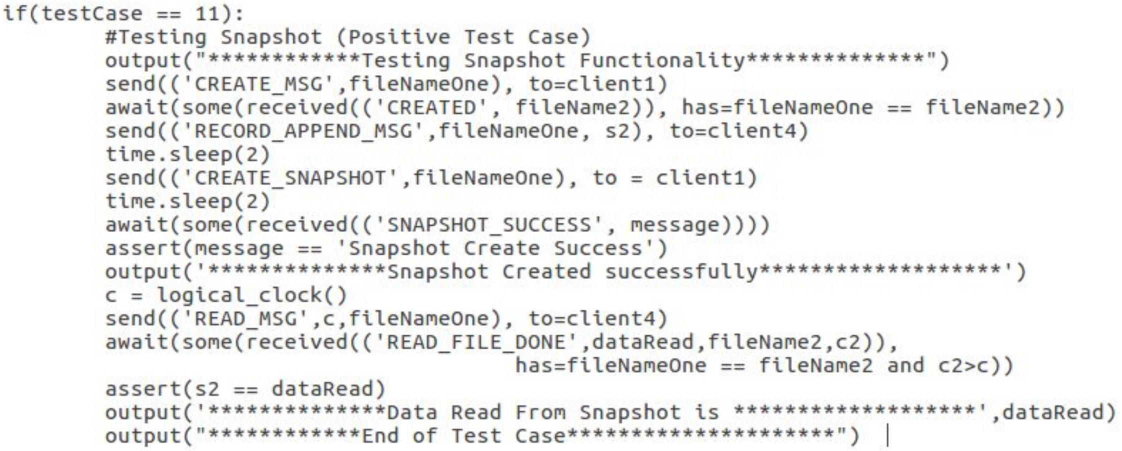
Output for above Testcase:



```
[4541] chunk.GFSChunkserver<GFSChunkserver:18007>:OUTPUT: File written
[4545] chunk.GFSChunkserver<GFSChunkserver:18008>:OUTPUT: Writing chunk Append5 7396ccc2-6b67-4622-adc1-a1e4e9814a7d-SNAPSHOT-SNAPSHOT
[4545] chunk.GFSChunkserver<GFSChunkserver:18008>:OUTPUT: filename /tmp/gfs/chunks/4/7396ccc2-6b67-4622-adc1-a1e4e9814a7d-SNAPSHOT-SNAPSHOT.gfs
[4545] chunk.GFSChunkserver<GFSChunkserver:18008>:OUTPUT: File written
[4549] chunk.GFSChunkserver<GFSChunkserver:18004>:OUTPUT: Writing chunk Append5 7396ccc2-6b67-4622-adc1-a1e4e9814a7d-SNAPSHOT-SNAPSHOT
[4549] chunk.GFSChunkserver<GFSChunkserver:18004>:OUTPUT: filename /tmp/gfs/chunks/0/7396ccc2-6b67-4622-adc1-a1e4e9814a7d-SNAPSHOT-SNAPSHOT.gfs
[4549] chunk.GFSChunkserver<GFSChunkserver:18004>:OUTPUT: File written
[5484] master.Master<Master:18003>:OUTPUT: Checking file file1.txt
[5491] client.GFSClient<GFSClient:18007>:OUTPUT: frozenlist(['a36cc842-551f-4b68-bd3f-4051f05aba0e-SNAPSHOT-SNAPSHOT', '875e6702-0106-448e-8e1c-1543
de40-4214-8104-25abda5bf4d6-SNAPSHOT-SNAPSHOT', '7396ccc2-6b67-4622-adc1-a1e4e9814a7d-SNAPSHOT-SNAPSHOT'])
[5519] test.Node_<Node_:ea801>:OUTPUT: **************Data Appended in Order ************** Append1232Append4Append5
[5519] test.Node_<Node_:ea801>:OUTPUT: **************Data Read from file is ************** Append1232Append4Append5
```

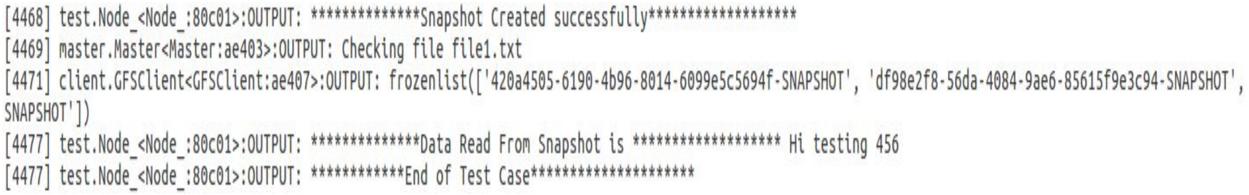
4.3.2.2. Snapshot

Data should be read correctly after Snapshot from the Snapshot chunk.
We have tested this by creating the Snapshot of a file in the chunkserver and then performed a read operation on it.
The data read is observed to be from the snapshot chunk



```
if(testCase == 11):
        #Testing Snapshot (Positive Test Case)
        output("************Testing Snapshot Functionality**************")
        send(('CREATE_MSG',fileNameOne), to=client1)
        await(some(received(('CREATED', fileName2)), has=fileNameOne == fileName2))
        send(('RECORD_APPEND_MSG',fileNameOne, s2), to=client4)
        time.sleep(2)
        send(('CREATE_SNAPSHOT',fileNameOne), to = client1)
        time.sleep(2)
        await(some(received(('SNAPSHOT_SUCCESS', message))))
        assert(message == 'Snapshot Create Success')
        output('**************Snapshot Created successfully*******************')
        c = logical_clock()
        send(('READ_MSG',c,fileNameOne), to=client4)
        await(some(received(('READ_FILE_DONE',dataRead,fileName2,c2)),
                                        has=fileNameOne == fileName2 and c2>c))
        assert(s2 == dataRead)
        output('**************Data Read From Snapshot is *******************',dataRead)
        output("************End of Test Case**********************")  |
```

Output of the above test case:



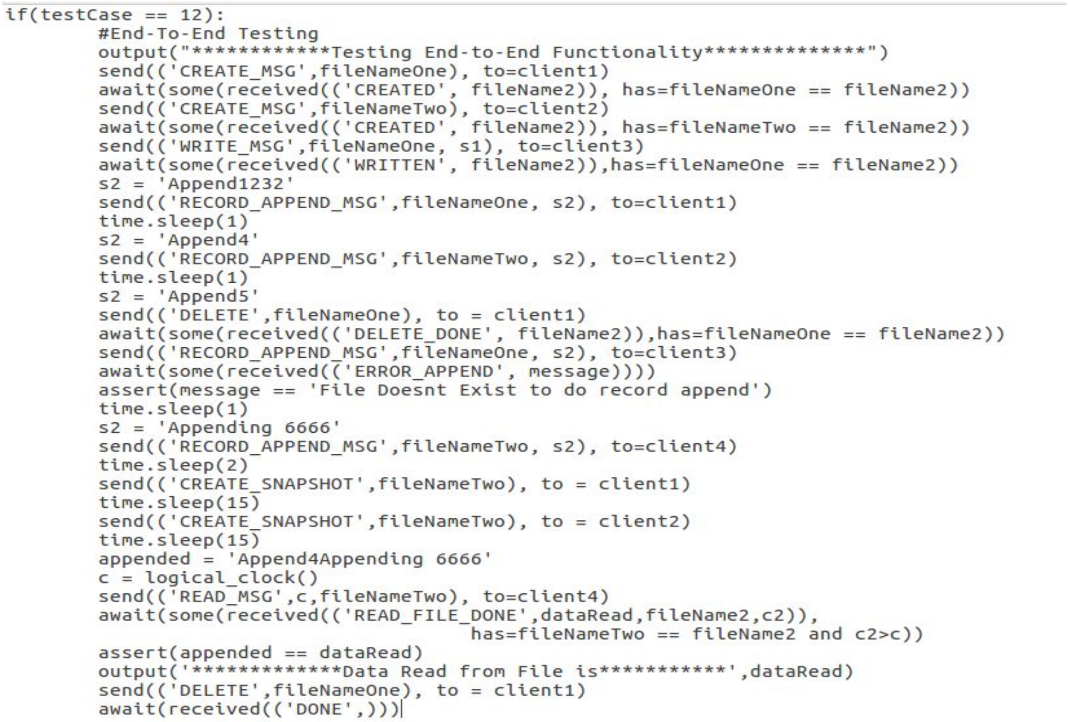
```
[4468] test.Node_<Node_:80c01>:OUTPUT: **************Snapshot Created successfully*******************
[4469] master.Master<Master:ae403>:OUTPUT: Checking file file1.txt
[4471] client.GFSClient<GFSClient:ae407>:OUTPUT: frozenlist(['420a4505-6190-4b96-8014-6099e5c5694f-SNAPSHOT', 'df98e2f8-56da-4084-9ae6-85615f9e3c94-SNAPSHOT',
SNAPSHOT'])
[4477] test.Node_<Node_:80c01>:OUTPUT: **************Data Read From Snapshot is ******************* Hi testing 456
[4477] test.Node_<Node_:80c01>:OUTPUT: ************End of Test Case**********************
```

4.3.3. <u>Functional Testing</u>

Implemented multiple test cases to verify the correctness of **individual** File System Operations as well as **end-to-end** execution of different Fops.

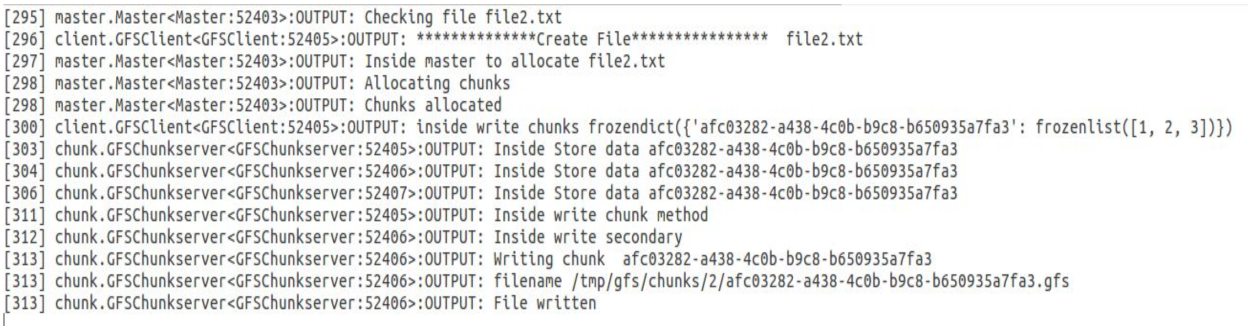
Created Positive and Negative Unit Test Cases for Create, Delete, Read, Write

Below is the code snippet of one of the end-to-end test cases we have implemented.



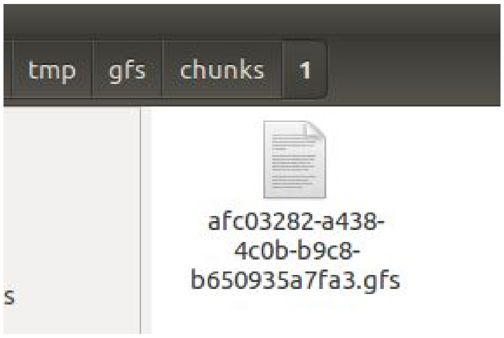
```
if(testCase == 12):
        #End-To-End Testing
        output("************Testing End-to-End Functionality**************")
        send(('CREATE_MSG',fileNameOne), to=client1)
        await(some(received(('CREATED', fileName2)), has=fileNameOne == fileName2))
        send(('CREATE_MSG',fileNameTwo), to=client2)
        await(some(received(('CREATED', fileName2)), has=fileNameTwo == fileName2))
        send(('WRITE_MSG',fileNameOne, s1), to=client3)
        await(some(received(('WRITTEN', fileName2)),has=fileNameOne == fileName2))
        s2 = 'Append1232'
        send(('RECORD_APPEND_MSG',fileNameOne, s2), to=client1)
        time.sleep(1)
        s2 = 'Append4'
        send(('RECORD_APPEND_MSG',fileNameTwo, s2), to=client2)
        time.sleep(1)
        s2 = 'Append5'
        send(('DELETE',fileNameOne), to = client1)
        await(some(received(('DELETE_DONE', fileName2)),has=fileNameOne == fileName2))
        send(('RECORD_APPEND_MSG',fileNameOne, s2), to=client3)
        await(some(received(('ERROR_APPEND', message))))
        assert(message == 'File Doesnt Exist to do record append')
        time.sleep(1)
        s2 = 'Appending 6666'
        send(('RECORD_APPEND_MSG',fileNameTwo, s2), to=client4)
        time.sleep(2)
        send(('CREATE_SNAPSHOT',fileNameTwo), to = client1)
        time.sleep(15)
        send(('CREATE_SNAPSHOT',fileNameTwo), to = client2)
        time.sleep(15)
        appended = 'Append4Appending 6666'
        c = logical_clock()
        send(('READ_MSG',c,fileNameTwo), to=client4)
        await(some(received(('READ_FILE_DONE',dataRead,fileName2,c2)),
                                        has=fileNameTwo == fileName2 and c2>c))
        assert(appended == dataRead)
        output('*************Data Read from File is***********',dataRead)
        send(('DELETE',fileNameOne), to = client1)
        await(received(('DONE',)))
```

<u>Testing Create Function</u> **:** *send(('CREATE_MSG','file2.txt'), to=client1)*

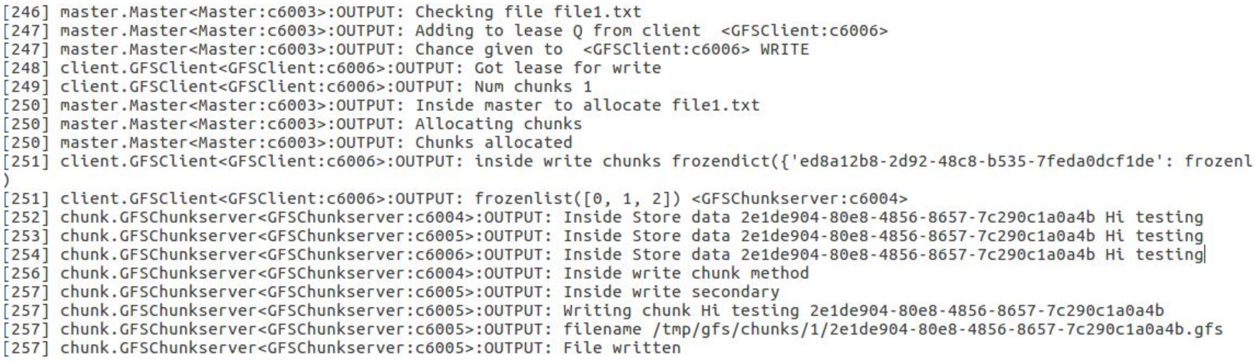


```
[295] master.Master<Master:52403>:OUTPUT: Checking file file2.txt
[296] client.GFSClient<GFSClient:52405>:OUTPUT: **************Create File****************  file2.txt
[297] master.Master<Master:52403>:OUTPUT: Inside master to allocate file2.txt
[298] master.Master<Master:52403>:OUTPUT: Allocating chunks
[298] master.Master<Master:52403>:OUTPUT: Chunks allocated
[300] client.GFSClient<GFSClient:52405>:OUTPUT: inside write chunks frozendict({'afc03282-a438-4c0b-b9c8-b650935a7fa3': frozenlist([1, 2, 3])})
[303] chunk.GFSChunkserver<GFSChunkserver:52405>:OUTPUT: Inside Store data afc03282-a438-4c0b-b9c8-b650935a7fa3
[304] chunk.GFSChunkserver<GFSChunkserver:52406>:OUTPUT: Inside Store data afc03282-a438-4c0b-b9c8-b650935a7fa3
[306] chunk.GFSChunkserver<GFSChunkserver:52407>:OUTPUT: Inside Store data afc03282-a438-4c0b-b9c8-b650935a7fa3
[311] chunk.GFSChunkserver<GFSChunkserver:52405>:OUTPUT: Inside write chunk method
[312] chunk.GFSChunkserver<GFSChunkserver:52406>:OUTPUT: Inside write secondary
[313] chunk.GFSChunkserver<GFSChunkserver:52406>:OUTPUT: Writing chunk  afc03282-a438-4c0b-b9c8-b650935a7fa3
[313] chunk.GFSChunkserver<GFSChunkserver:52406>:OUTPUT: filename /tmp/gfs/chunks/2/afc03282-a438-4c0b-b9c8-b650935a7fa3.gfs
[313] chunk.GFSChunkserver<GFSChunkserver:52406>:OUTPUT: File written
```

File is Created in the folder /tmp/gfs/chunks/1

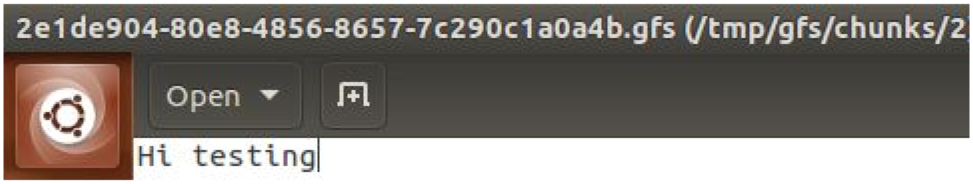


<u>Testing Write Function</u>: *send(('WRITE_MSG','file1.txt', ‘Hi Testing’), to=client3)*



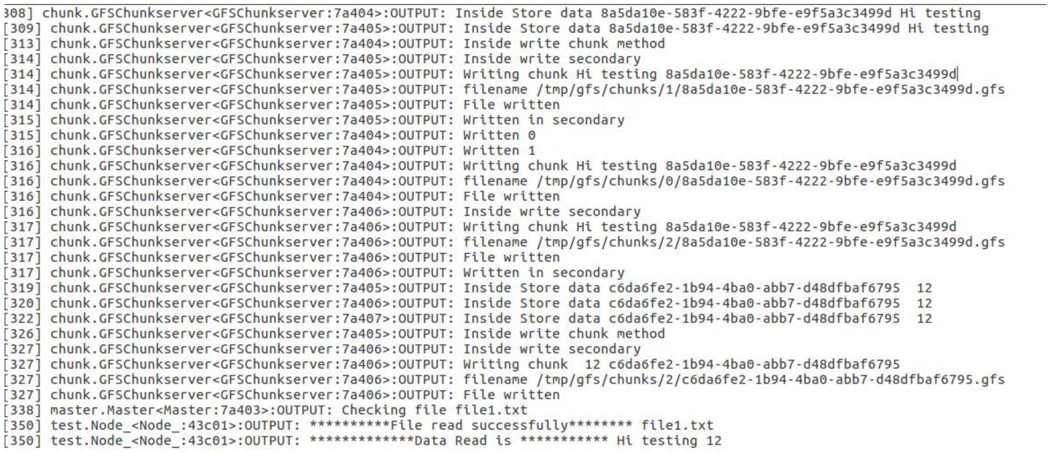
```
[246] master.Master<Master:c6003>:OUTPUT: Checking file file1.txt
[247] master.Master<Master:c6003>:OUTPUT: Adding to lease Q from client  <GFSClient:c6006>
[247] master.Master<Master:c6003>:OUTPUT: Chance given to  <GFSClient:c6006> WRITE
[248] client.GFSClient<GFSClient:c6006>:OUTPUT: Got lease for write
[249] client.GFSClient<GFSClient:c6006>:OUTPUT: Num chunks 1
[250] master.Master<Master:c6003>:OUTPUT: Inside master to allocate file1.txt
[250] master.Master<Master:c6003>:OUTPUT: Allocating chunks
[250] master.Master<Master:c6003>:OUTPUT: Chunks allocated
[251] client.GFSClient<GFSClient:c6006>:OUTPUT: inside write chunks frozendict({'ed8a12b8-2d92-48c8-b535-7feda0dcf1de': frozenl
)
[251] client.GFSClient<GFSClient:c6006>:OUTPUT: frozenlist([0, 1, 2]) <GFSChunkserver:c6004>
[252] chunk.GFSChunkserver<GFSChunkserver:c6004>:OUTPUT: Inside Store data 2e1de904-80e8-4856-8657-7c290c1a0a4b Hi testing
[253] chunk.GFSChunkserver<GFSChunkserver:c6005>:OUTPUT: Inside Store data 2e1de904-80e8-4856-8657-7c290c1a0a4b Hi testing
[254] chunk.GFSChunkserver<GFSChunkserver:c6006>:OUTPUT: Inside Store data 2e1de904-80e8-4856-8657-7c290c1a0a4b Hi testing
[256] chunk.GFSChunkserver<GFSChunkserver:c6004>:OUTPUT: Inside write chunk method
[257] chunk.GFSChunkserver<GFSChunkserver:c6005>:OUTPUT: Inside write secondary
[257] chunk.GFSChunkserver<GFSChunkserver:c6005>:OUTPUT: Writing chunk Hi testing 2e1de904-80e8-4856-8657-7c290c1a0a4b
[257] chunk.GFSChunkserver<GFSChunkserver:c6005>:OUTPUT: filename /tmp/gfs/chunks/1/2e1de904-80e8-4856-8657-7c290c1a0a4b.gfs
[257] chunk.GFSChunkserver<GFSChunkserver:c6005>:OUTPUT: File written
```

File Written in the folder /tmp/gfs/chunks/2



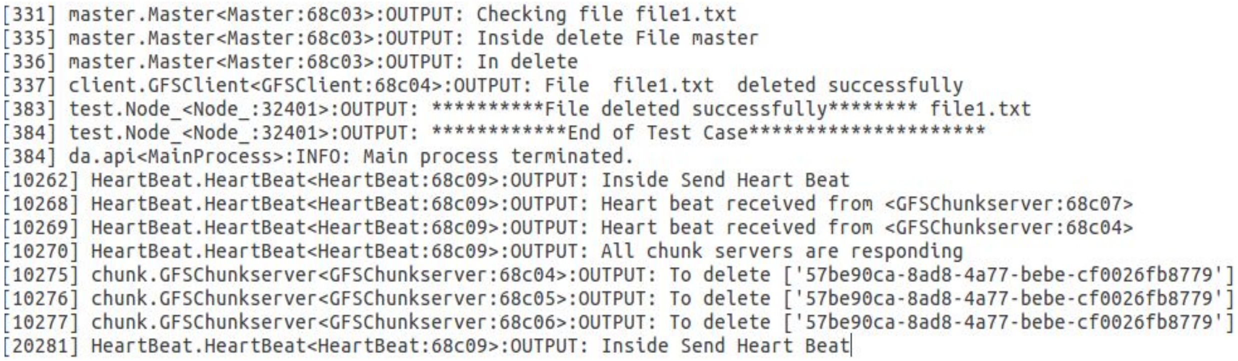
```
Hi testing
```

<u>Testing Read Function</u>**:** *send(('READ_MSG',c,'file1.txt'), to=client4)*



```
308] chunk.GFSChunkserver<GFSChunkserver:7a404>:OUTPUT: Inside Store data 8a5da10e-583f-4222-9bfe-e9f5a3c3499d Hi testing
[309] chunk.GFSChunkserver<GFSChunkserver:7a405>:OUTPUT: Inside Store data 8a5da10e-583f-4222-9bfe-e9f5a3c3499d Hi testing
[313] chunk.GFSChunkserver<GFSChunkserver:7a404>:OUTPUT: Inside write chunk method
[314] chunk.GFSChunkserver<GFSChunkserver:7a405>:OUTPUT: Inside write secondary
[314] chunk.GFSChunkserver<GFSChunkserver:7a405>:OUTPUT: Writing chunk Hi testing 8a5da10e-583f-4222-9bfe-e9f5a3c3499d
[314] chunk.GFSChunkserver<GFSChunkserver:7a405>:OUTPUT: filename /tmp/gfs/chunks/1/8a5da10e-583f-4222-9bfe-e9f5a3c3499d.gfs
[314] chunk.GFSChunkserver<GFSChunkserver:7a405>:OUTPUT: File written
[315] chunk.GFSChunkserver<GFSChunkserver:7a405>:OUTPUT: Written in secondary
[315] chunk.GFSChunkserver<GFSChunkserver:7a404>:OUTPUT: Written 0
[316] chunk.GFSChunkserver<GFSChunkserver:7a404>:OUTPUT: Written 1
[316] chunk.GFSChunkserver<GFSChunkserver:7a404>:OUTPUT: Writing chunk Hi testing 8a5da10e-583f-4222-9bfe-e9f5a3c3499d
[316] chunk.GFSChunkserver<GFSChunkserver:7a404>:OUTPUT: filename /tmp/gfs/chunks/0/8a5da10e-583f-4222-9bfe-e9f5a3c3499d.gfs
[316] chunk.GFSChunkserver<GFSChunkserver:7a404>:OUTPUT: File written
[316] chunk.GFSChunkserver<GFSChunkserver:7a406>:OUTPUT: Inside write secondary
[317] chunk.GFSChunkserver<GFSChunkserver:7a406>:OUTPUT: Writing chunk Hi testing 8a5da10e-583f-4222-9bfe-e9f5a3c3499d
[317] chunk.GFSChunkserver<GFSChunkserver:7a406>:OUTPUT: filename /tmp/gfs/chunks/2/8a5da10e-583f-4222-9bfe-e9f5a3c3499d.gfs
[317] chunk.GFSChunkserver<GFSChunkserver:7a406>:OUTPUT: File written
[317] chunk.GFSChunkserver<GFSChunkserver:7a406>:OUTPUT: Written in secondary
[319] chunk.GFSChunkserver<GFSChunkserver:7a405>:OUTPUT: Inside Store data c6da6fe2-1b94-4ba0-abb7-d48dfbaf6795  12
[320] chunk.GFSChunkserver<GFSChunkserver:7a406>:OUTPUT: Inside Store data c6da6fe2-1b94-4ba0-abb7-d48dfbaf6795  12
[322] chunk.GFSChunkserver<GFSChunkserver:7a407>:OUTPUT: Inside Store data c6da6fe2-1b94-4ba0-abb7-d48dfbaf6795  12
[326] chunk.GFSChunkserver<GFSChunkserver:7a405>:OUTPUT: Inside write chunk method
[327] chunk.GFSChunkserver<GFSChunkserver:7a406>:OUTPUT: Inside write secondary
[327] chunk.GFSChunkserver<GFSChunkserver:7a406>:OUTPUT: Writing chunk  12 c6da6fe2-1b94-4ba0-abb7-d48dfbaf6795
[327] chunk.GFSChunkserver<GFSChunkserver:7a406>:OUTPUT: filename /tmp/gfs/chunks/2/c6da6fe2-1b94-4ba0-abb7-d48dfbaf6795.gfs
[327] chunk.GFSChunkserver<GFSChunkserver:7a406>:OUTPUT: File written
[338] master.Master<Master:7a403>:OUTPUT: Checking file file1.txt
[350] test.Node_<Node_:43c01>:OUTPUT: **********File read successfully******** file1.txt
[350] test.Node_<Node_:43c01>:OUTPUT: *************Data Read is *********** Hi testing 12
```

<u>Testing Delete Function</u>**:** *send(('DELETE','file1.txt'), to = client1)*



```
[331] master.Master<Master:68c03>:OUTPUT: Checking file file1.txt
[335] master.Master<Master:68c03>:OUTPUT: Inside delete File master
[336] master.Master<Master:68c03>:OUTPUT: In delete
[337] client.GFSClient<GFSClient:68c04>:OUTPUT: File  file1.txt  deleted successfully
[383] test.Node_<Node_:32401>:OUTPUT: **********File deleted successfully******** file1.txt
[384] test.Node_<Node_:32401>:OUTPUT: ************End of Test Case**********************
[384] da.api<MainProcess>:INFO: Main process terminated.
[10262] HeartBeat.HeartBeat<HeartBeat:68c09>:OUTPUT: Inside Send Heart Beat
[10268] HeartBeat.HeartBeat<HeartBeat:68c09>:OUTPUT: Heart beat received from <GFSChunkserver:68c07>
[10269] HeartBeat.HeartBeat<HeartBeat:68c09>:OUTPUT: Heart beat received from <GFSChunkserver:68c04>
[10270] HeartBeat.HeartBeat<HeartBeat:68c09>:OUTPUT: All chunk servers are responding
[10275] chunk.GFSChunkserver<GFSChunkserver:68c04>:OUTPUT: To delete ['57be90ca-8ad8-4a77-bebe-cf0026fb8779']
[10276] chunk.GFSChunkserver<GFSChunkserver:68c05>:OUTPUT: To delete ['57be90ca-8ad8-4a77-bebe-cf0026fb8779']
[10277] chunk.GFSChunkserver<GFSChunkserver:68c06>:OUTPUT: To delete ['57be90ca-8ad8-4a77-bebe-cf0026fb8779']
[20281] HeartBeat.HeartBeat<HeartBeat:68c09>:OUTPUT: Inside Send Heart Beat
```

4.3.4. <u>Conclusion From Correctness testing</u>

We ran a lot of scenarios(17 of them) positive and negative to see if the functionalities are working as expected. The correctness testing helped us to find bugs in our code and fix them. It helped us to provide good exception handling so that the user could get proper messages back in case of errors and even our code will not get stuck(as we added timeout in all places).

4.4. Comparison with Best Implementation

We did not find any implementation in DistAlgo. The one we found was in python but it is spread across various servers and our implementation is on local machine. Compared to the implementation we found in Python we implemented the HeartBeat process in the way that the paper describes [1] (section 4.4.1), where the heartbeat response from chunkServer brings a list of files and the master sends to the chunk server which of these files needs to be deleted.

5. Obstacles faced and fixes done in current implementation

1. We were not able to do the Heartbeat functionality from Master directly as it was causing the master to remain paused while it was waiting for acknowledgement for heartbeat. So we created a new heartbeat process from it which does the required job as we had explained earlier.
2. We were randomly getting MessageTooBigException. After many trials and error, we think(still not sure) it was coming, as we were passing too much data in the received queue(we were passing entire maps from master). We reduced the amount of data we were passing during message passing and that helped to get rid of this error at some places, but sometimes we still do get this error.

6. Conclusion

Implementation of Google File System allowed us to gain in-depth knowledge regarding distributed storage systems. It also gave us good experience on how to program on asynchronous systems. DistAlgo, being a very versatile and dynamic language, allowed us to very easily implement our work as we had wanted to. Working with the DisAlgo language simplified many of our needs to access system level calls, and we only required to work with the higher-level interface (send and receive methods, in particular) provided by the language. Thus, despite the time constraint we were under, we were not only able to easily able to get the hang of DistAlgo and implement the Google File System in an efficient and quick manner, but also learn about asynchronous programming and distributed file systems thoroughly.

7. Future Work

1. We did not implement open and close as separate api as they are internally implemented in the all the other file system operations we implemented. Also in the implementations we searched none of them had implemented it. Owing to time limitations we were not able to implement un-delete functionality.
2. For Snapshot we assumed that if snapshot request is made the user will never try to undo that process(rollback).
3. We did not implement re-replication, rebalancing, stale-replica detection.
4. The current implementation is on local storage but it could easily be expanded to multiple servers.
5. We need to figure out a way to remove messages from the queue, as it sometimes causes issues giving MessageTooBigException.

8. References

[1] “The Google File System”, written by Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

[2] Wikipedia - https://en.wikipedia.org/wiki/Distributed_data_store

[3] Wikipedia - https://en.wikipedia.org/wiki/Category:Distributed_file_systems

[4] Distributed Systems GFS Implementation by BenningtonCS:
https://github.com/BenningtonCS/GFS

[5] GFS the Google File System in 199 Lines of Python by John Arley Burns:
http://clouddbs.blogspot.com/2010/11/gfs-google-file-system-in-199-lines-of.html