Half Baked GFS

A Report for Project in COL-733

Submitted by

Ronak Ladhar	**2021MCS2146**
Sourav Sharma	**2021MCS2154**
Chintan Sheth	**2021MCS2129**

Guided By

Dr. Abhilash Jindal

Indian Institute of Technology - Delhi

1 Introduction

Half baked GFS is a distributed, scalable and fault tolerant storage system. We have made use of Rpyc library to implement remote procedure calls for communication between master, chunkservers and the client. The motive behind choosing this library over others is that it has the provision of asynchronous remote procedure calls which can optimize the operations in our architecture. Our major driving factor behind implementing a mini version of GFS was to get hands-on experience of working on distributed systems and to learn from the challenges faced while implementing the same.

2 Design Assumptions

The master is assumed to be reliable and thus, we have not made the system tolerant to the master failures.

3 Architecture

3.1 Master

The master manages the entire control flow of the system and acts as an initiator of the communication between the client and the chunkservers. It manages the creation, removal and the management of chunks and files entirely and ensures the correct behavior of the system.

3.2 Chunkserver

The chunkservers are involved in the data flow in connection with the clients. They will serve the read or write request for the chunks which are allocated to it. Chunkservers also participates in control flow with the master where they sends periodic heatbeat messages to master and gets stale replicas in return.

3.3 Client

The clients are continuously communicating with the master as well as the chunkservers to get its purpose served. The clients will be the endpoint available to the applications to perform file operations in the Half-baked GFS architecture.

3.4 Metadata

On the client we maintain information on chunks corresponding to the file and its chunk number which includes the chunk id of that chunk number, primary chunkserver, secondary chunkservers and cache timeout at which this information expires. On chunkserver we store information about chunks currently been stored on it which is mapping from chunk id to file name to which the chunk belongs, version number which later helps in stale chunk detection, lease expiry time if replica is elected as a primary for that chunk id, offset from which data can be written in the chunk, and md5 hash of the data stored in it. On master we are maintaining information about both the chunk servers which is a mapping from chunkserver url to list of chunk id's which are currently present on chunkserver and heartbeat time at which request is received from chunkserver, and the chunks which is mapping from chunk id's to file name to which the chunk belongs, primary chunkserver if elected, secondary chunk servers holding replicas of the chunks and current version of the chunk.

4 Master Operations

4.1 Chunk Re-replication

Master has a background thread which periodically runs at an interval of 50 sec, and checks whether the number of replicas corresponding to each chunk is greater than equal to replication factor. If the number of replicas corresponding to any chunk is found to be less than replication factor, then it will compute a list of chunkserver urls which are currently not holding any replica for that chunk and will iterate over it one-by-one till it is successful in replicating the chunk onto desired number of chunk servers. When a request from master is sent to chunserver for replication, it contains the list of replicas currently holding the chunk, version number which can later be verified by chunkserver performing the replication.

4.2 Garbage Collection

The master periodically runs a background thread at an interval of 50 seconds, which checks for the files present in the trash i.e. with prefix "TRASHFILE_" in its cache whether the time has exceeded the restoration timeout of 40 seconds. If it has exceeded, it will delete the entries for the file from its metadata. Sooner, when the chunk servers who are holding the chunks for the file will send the heartbeat to the master, it will return those chunks marked as stale ones and thus, the chunk servers will delete the metadata as well as the files stored on the disk for corresponding chunks.

4.3 Stale Replica Detection

The chunkserver could be holding stale chunks in multiple scenarios. If the chunkserver crashed and got alive after some time, the master would have re-replicated the chunks held by it during the meantime. The chunkserver maintains the persistent state, thus, when it comes up, it will sync with the master by sending the chunks held by it. The master will verify the version number for the chunks, if the chunkserver has the chunk with less version number and will return those chunks marked as stale and the chunkserver will delete the metadata and files corresponding to those chunks. The other case is when the client was not able to connect to the chunkserver for some duration and it assumed that it had crashed and requested the master to remove the chunkserver and ask for a new replica for the chunk. The master will re-replicate the chunk sooner. Now, when the chunk server sends the heartbeat to the master, it identifies that the chunk has been re-replicated and will return the chunks marked as stale and the chunkserver will delete the metadata and files corresponding to those chunks. The last case is when the file has been deleted, which has been described in the Garbage Collection.

4.4 Lease Extension

Whenever a client wants to commit data to a chunk, it needs information about the primary replica for that chunk. Client requests this information from the master and caches it for a duration of 30sec. Master will check its metadata to see whether the primary is assigned to the requested chunk, and its lease time has not expired then this primary can directly be supplied to the client. Once primary for a particular chunk is elected by master, primary chunkserver can request for lease extension as long as it wants by sending a lease extension request to master with periodic heartbeat messages exchanged between chunkserver and master. Lease expiry interval of 60 sec is chosen to be greater than heartbeat interval of 30 sec, so that lease extension request can be sent before expiration piggybacked with heartbeat request.

4.5 Hearbeat

The chunk servers periodically send heartbeat messages to the master along with the list of chunks held by it. The master verifies whether the chunks held by the chunkserver have become stale and also requests the chunkserver to remove those chunks. It also checks if the chunkserver holds the chunk with the latest version number, but the master doesn't have mapping for the chunkserver with the respective chunkserver, then it will add the chunkserver to the replicas of the chunk, which will save the data transfer time, had the replication happened on different chunkserver which was not holding the chunk earlier, the data had to be transferred from some existing replica to new chunkserver.

5 Client Operations

5.1 Write

End-user passes the data and the filename. Offset in client identify the chunk id to which we need to append the data. Client finds the primary and secondary replicas url for the chunk id. In case the size of data spans multiple chunks it splits the file accordingly.

5.2 Read

Client accepts the read request from the end-user. The end-user passes the client the filename, offset and the number of characters to read from the file. Client computes the chunk id to which this offset belongs, once that is done the client searches its local cache for the filename and chunk id. If it is present, the information related to the url chunk servers which contains this file is utilized to make a random selection among the url for read request. In case if it is not present we first populate the client cache by asking the master for the replicas containing a given chunk id corresponding to the filename. The client sends the request to the chunkserver (randomly picked from the replica url) with the chunk id, offset and amount of data to read in bytes(or characters) from the offset. If a read request spans multiple chunks then the client sends read sequentially. At last the merged result is sent in response to the end-user.

5.3 Delete

When the application sends a delete request to the client along with the file name to be deleted, the client forwards the request to the master, which moves the file to the trash.

5.4 Restore

This request is generated for the files which are deleted. If the request is generated before the delete timeout of the file to be restored, then it is possible for the end-user to restore the file back.

6 Fault Tolerance

6.1 Chunk Replication and Re-replication

The re-replication of the chunk could happen in multiple cases. Each chunkserver runs a background thread to verify the checksum of each chunk held by it. During this validation task, if it finds that the file doesn't exist in the storage due to disk crash or some other reasons, or the checksum of the file changed which signifies that the file got corrupted due to some unwanted scenarios, it will ask the master to remove its mapping for the corresponding chunk which will result in less number of replicas for the chunk. When the periodic re-replication task will be executed on the master, it will perform re-replication of the chunk. When the chunkserver crashes, the periodic heartbeat validation check running on master at an interval of 40 seconds will identify it has not received heartbeat from the chunkserver, and will remove the mappings for all the chunks corresponding to the chunkserver. This will also lead to re-replication of chunks held by that chunkserver. Thus, the implementation is tolerant to disk failures as well as chunkserver failures. Moreover 3 replicas of each chunk are stored onto different chunkserver which also provides resistance towards entire loss of data in case of single or at most two chunkserver failure. But to ensure this we need to make sure that the interval of periodic re-replication at master is not too high so that single chunkserver failure could be detected as soon as it fails.

6.2 Checksum Validation

The checksum validation is performed by the chunkserver periodically for each chunk held by it. We store a md5 hash with each chunk on chunkserver, which is used by the background task of chunkserver to verify integrity of the data stored on the disk. Through checksum validation, the disk failures as well as chunk corruption could be identified and if the chunkserver finds any such corrupted chunk, it requests the master to remove its mappings for the corresponding chunk and also removes mapping from its own metadata.

7 High Availability

7.1 Chunkserver

Any modifications to chunkserver metadata are persistently stored on disk, so that it remains persistent across chunkserver reboot. When a chunkserver reboots, it loads its state from disk if present and then sends a sync chunkserver request to master together with a list of tuples containing chunk ids and version number of chunks stored in it. Master will check its state and will send stale replicas back to chunkserver. Chunkserver on getting a list of stale replicas will remove them from its storage as well as from its state. Moreover, in case of chunkserver failure, the background thread running on master will identify that it had not received heartbeat messages from failed chunkserver then it will remove its entries from chunkserver information and from list of replicas which are stored corresponding to each chunk id. Another background thread will kick in, verifying if the replica count is equal to the replication factor, and scheduling re-replication of chunks where the replica count has fallen below the replication factor.

8 Benchmarking

8.1 Hardware Specification

8.1.1 Chunkserver

- Processor Cores - 2 cores of Intel Xeon E3-12xx v2 (Ivy Bridge, IBRS)
- RAM - 8 GB

8.1.2 Master

- Processor - 8 cores of Intel Core Processor (Haswell, no TSX, IBRS)
- RAM - 8 GB

8.1.3 Client

- Processor - 8 cores of Intel Core Processor
- RAM - 8 GB

8.2 Write

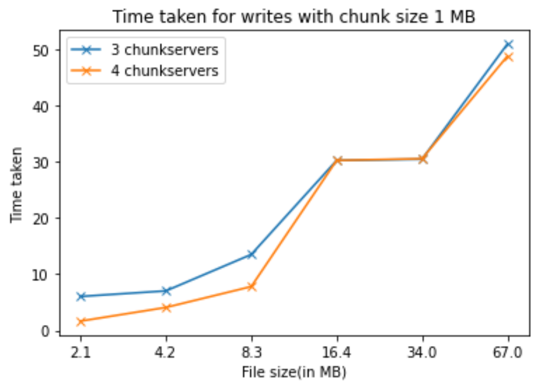

Figure 1: Time taken for writes with chunk size 1 MB

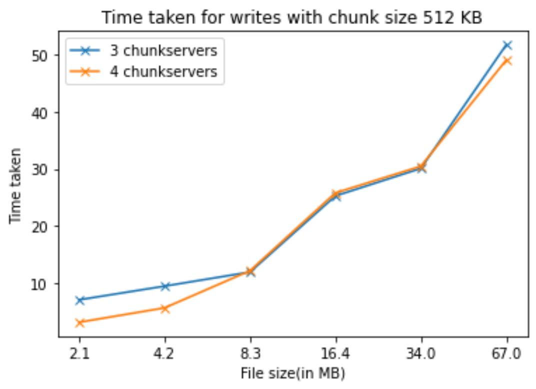

Figure 2: Time taken for writes with chunk size 512 KB

8.3 Read

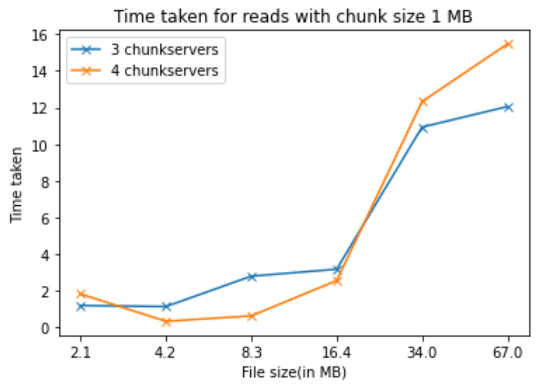

Figure 3: Time taken for reads with chunk size 1 MB

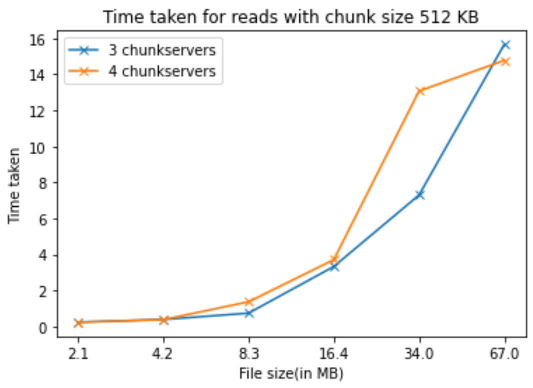

Figure 4: Time taken for writes with chunk size 512 KB

8.4 Benchmarking with HDFS

8.4.1 Write

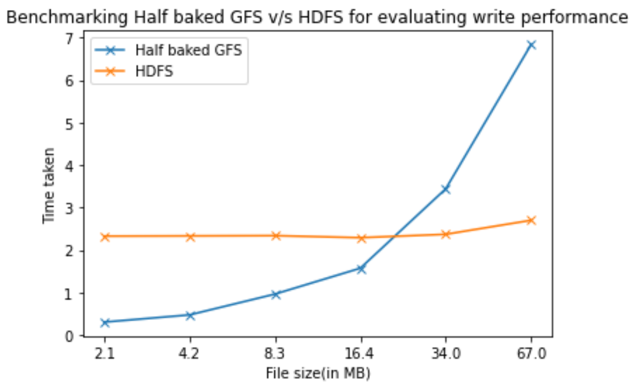

Figure 5: Benchmarking write performance of Half baked GFS v/s HDFS

8.4.2 Read

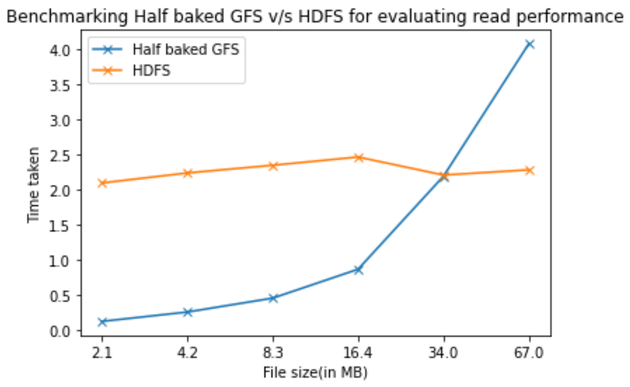

Figure 6: Benchmarking write performance of Half baked GFS v/s HDFS

9 Observations

It can be observed from the graphs that on increasing the number of chunkservers, the write rate improves which is because of even distribution of chunks by the master over the chunkservers.
It can also be observed that on increasing the chunk sizes, the write rate improves which is due to less number of chunks being created in case of large chunk sizes which reduces the creation and management overhead of the chunks.
The read and write rates are quite lower than our expectations because we have implemented synchronous writes as well as reads. If implemented in asynchronous manner, we can expect better performance of reads and writes.
When comparing Half-Baked GFS to HDFS, it is evident that Half-Baked GFS outperforms HDFS for medium file sizes, but as file sizes become larger, Half-Baked GFS' performance diminishes. This could be due to chunk formation and maintenance cost. Furthermore, in the case of Half-Baked GFS, read and write operations are synchronous, as opposed to HDFS, which has a negative influence on performance when numerous chunks are involved.

10 Test Cases

10.1 Writing to GFS

The application requests the client to write to a file by providing the file name along with data and the offset at which the contents are to be written. The write flow is divided into three steps.

10.1.1 Chunk Creation

At first, the file is to be created in the system if it doesn't exist. The client sends the create request to the master, which in turn creates a new file and the first chunk corresponding to the file and stores it in its metadata cache. The newly created chunk is assigned three replicas which are then returned to the client, so that it can initiate appending to the file.

10.1.2 Append to Log

If the file has already been created, the client will first check if it holds the chunk id and chunkserver urls corresponding to the file and the chunk number. If it holds the information in the cache, it will directly send the data to a random chunkserver out of the replicas for the chunk. If it doesn't hold the information, it will request the master to send the chunk id, primary and replicas, and then it will send the append request to any random chunkserver out of the replicas. The first chunkserver which gets the append request from the client will write it to the cached log and forward it to a random replica out of the replicas sent by the client. The same procedure will be followed synchronously until the data reaches the last replica. Each replica will wait for the response from the next replica. Once the first chunkserver gets the success from the next replica, it will send a success message to the client. If the request fails for any of the chunkserver, the client will request the master to remove the mappings of the chunkserver.

10.1.3 Commit Append

In case of success received from the chunkserver, the client will send the commit append request to the primary which will append the data present in the cached log to the persistent local disk. The primary will have the offset at which the contents were last written on the chunk, which will then be sent along with the commit request to both the secondary replicas. The secondary replicas will append the log to the persistent local storage and will reply successfully to the primary. If the primary receives success from both the secondaries and it is also able to append the log successfully, it will return success to the client. If any of the replicas is not able to commit, the primary will return failure to the client.

10.1.4 Writes spanning over multiple chunks

If the data is large enough to fit into one chunk, the client divides the data such that the data is spanned over multiple chunks. In such a case, after successfully writing to one chunk, the client will request the master to create a new chunk and assign replicas to it. After receiving success from the master for chunk creation, the client will send the append request to a random replica and the same append flow as described above will be followed until the entire data is not written to the system.

10.2 Reading from GFS

Application calls for reading from a filename a given number of characters to read from a given offset. The client computes the chunk number which contains the data at the given offset. The client checks in its cache for the data of the chunk that belongs to the filename. Also, the chunk servers which contain this chunk are saved in the cache. So the client connects randomly to a chunk server and asks for the data from the chunk. The client also passes what amount of data to read from the given chunk. Client read requests can span multiple chunks, and thus each time the client supplies the offset and the amount of data to read from the chunk. In case the cache present at the client doesn't contain data for the given chunk for the file passed by the user, it first asks the master for the data and then cache that data.

10.3 Invalid Checksum

The chunkserver periodically performs checksum validation which checks for the occurrence of disk crash or file corruption. If it finds any such case, it will request the master to remove its mappings for the corresponding chunk in case of file corruption or for all chunks in case of disk crash. The master will remove the mappings for the chunkserver in the metadata accordingly and as soon as the re-replication thread will kick in, it will ask a chunkserver which doesn't hold the chunk to replicate the chunk by getting the data from one of those holding it.

10.4 Re-replicate chunks when replica count is below replication factor

In case of a chunkserver crash, background thread running at master will detect the chunkserver failure if it didn't receive the heartbeat from chunkserver in the heartbeat interval. As a result its mapping from the replica list corresponding to all the chunks if present are removed and re-replication of those chunks would be done by background re-replication thread running periodically on master.

10.5 Deletion of file, moves to trash and from trash permanently deleted on garbage collection

The delete request generated by the end-user gives the filename. First remove all the entries (filename, chunk_x) from the cache of the client. The request for the master to remove the filename and all chunks from its metadata. The master assigned deleted_timeout for the file. It removes the filename and all its chunks from its metadata. And temporarily for the deleted timeout, it maintains a filename with the prefix of TRASHFILE_in the metadata. So if the request for the restore the file arrives before the deleted_timeout then we will restore the file metadata back to the master

10.6 Restoring file from trash before delete timeout expires

If the client sends the restoration request for the file deleted before the expiration of the trash timeout, the master will update the file name in its metadata from TRASHFILE_filename to filename, and thus, the older state of the file will be restored.

10.7 Persistent state of chunkserver

The chunkserver maintains its state on the local disk, so that it can be kept tolerant to chunkserver failures or reboots. When the chunkserver becomes alive after reboot, it will sync with the master by sending the chunks held by it. The master will in turn check whether the chunks held by it are of latest version, if they are then it will add the chunkserver to the replicas for the chunk, otherwise it will send the chunks marked as stale to the chunkserver which will then remove it from the cache and also remove the corresponding files from the disk.

10.8 Stale replicas

When chunkserver reboots, it checks to see whether it has some state stored on the local disk, if it finds one, it loads its state from it and send sync chunkserver request to master. Master will validate the chunk list sent by chunkservers by verifying the version numbers for the respective chunks and returns stale replicas to the chunkservers. The chunkserver will remove the metadata present in its cache as well as data files present in its storage disk corresponding to the stale chunks held by it. Also with each heartbeat message chunkserver sends list of chunk_id with their corresponding version number. If the file has been deleted from the system and the chunkserver is not aware of it, the master will mark the chunks as stale corresponding to the file and will return those to the chunkserver.

11 Future Work

11.1 Persistent Master

To make the state of the master persistent across power failures or crashes. Also, replicating the master's state periodically to make it available even during master failure.

11.2 Rebalancing Chunks over Chunkservers

The master will periodically re-balance the chunks based on the workload of the chunkservers.

11.3 Atomic Record Appends

The current implementation doesn't support concurrent writes, but this can be done using transactional appends.

11.4 Creation of Directories

Currently, single level files creation is only supported, will extend support for hierarchical directories.

12 Contributions

Initially, we all came down to create a high-level framework for our code, which included deciding on various APIs, their parameters, and developing high-level workflows for the various GFS functionality. And later all of us had contributed in building presentation, and report. Tested the rpyc library to validate its use in our implementation. Came up with the idea of using the rpyc library which supports asynchronous remote procedure calls which could improve the performance of our implementation by huge extent.

Ronak Ladhar implemented the workflows for periodic verification of checksum where background thread periodically checks for any corruption of chunks and disk failures, this workflow together with re-replication background thread at master had added high level of fault tolerance to our system against any of the failures and testing of various failure scenarios related to these flows. Also, contributed in installation and benchmarking marking of HDFS.

Sourav Sharma implemented the read and came up with the idea of maintaining the cache at the client to avoid unnecessary requests to the master. Implemented the workflow of deleting the files and the periodically running thread at master for the garbage collection.

Chintan Sheth implemented the workflow of the three-step writes. Initially tried to implement asynchronous workflow of the writes, but were not able to achieve the desired behaviour. Finally, came up with the synchronous workflow. Also contributed in developing scripts to perform benchmarking of reads and writes. The workflow was initially built using static lease extensions, and then the workflows were later enhanced with dynamic lease extensions in conjunction with heartbeats, resulting in a significant increase in our write performance.

13 References

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

14 Code & Presentation Link

Presentation Slides Half-baked-GFS
Link to Code Half-baked-GFS