Google File System Implementation with SWIM Protocol and Consensus

Reedam Dhake, Shivam Singh, Viraj Agashe
Github Link: COL733-Project

November 17, 2023

Abstract

Google File System is the first distributed storage system which we learnt about in the course. Though it seems intuitive when we discuss it at a high level, when one gets to the implementation, there are a lot of details to be kept in mind. We chose it as a project to get hands-on experience in coding a large system. We also tried out some of our ideas to try and improve the efficiency of GFS as described in the original paper. We also discuss the results, learnings, and the various challenges we encountered.
Presentation Slides: GFS Project

1 Introduction

1.1 Motivation

In our abstract, we discussed the motivations for choosing the Google File System to implement. Here, we motivate our core ideas for the modifications we propose to the Google File System:

1.1.1 Network Faults - SWIM Protocol

In the version of GFS as described in the paper, the master receives *heartbeats* from all the chunkservers to monitor their status - whether they are alive or not. If the master does not receive heartbeats from a chunkserver, it must replace it by bringing up a new chunkserver - since writes will fail until this chunkserver comes up, and also the degree of replication decreases.

However, it may be possible that the network connecting master to the chunkserver may be down, but the chunkserver may actually be alive. The client may still be able to contact it and write to the chunk successfully. Therefore, we may end up wasting effort in replicating the chunk again on another chunkserver and updating it. We will also have to wait to service reads and writes during this period, which is a performance hit from the standpoint of the client.

We claim that this has a relatively simple fix for this problem of a "false positive" due to simple heartbeating. For this, we implement SWIM, a gossip protocol for verifying group membership. We will discuss the details of this protocol in the following sections.

1.1.2 Consensus Protocol

Another direction we wished to explore in this project is the idea of implementing a consensus protocol. The motivation for this is simple - in GFS we need to wait for all of the chunkservers in a group to perform a write, and if all are successful, only then we consider a write successful. However, we wish to explore a paradigm in which we only need to write to a *majority* of chunkservers responsible for a chunk, for a write to be considered complete.

Apriori, this seems like it would increase the performance of GFS in a write-heavy workload. However, reads become complicated. We will attempt to address this problem theoretically in the subsequent sections using concepts from a consensus protocol known as Virtual Consensus, which we discuss in detail in a subsequent section.

1.2 Setup

We have carried out the implementation of GFS in Python. Our setup, as given in Figure 1, is as follows. For an elaborate explanation, one may see Section 4 for the implementation details.

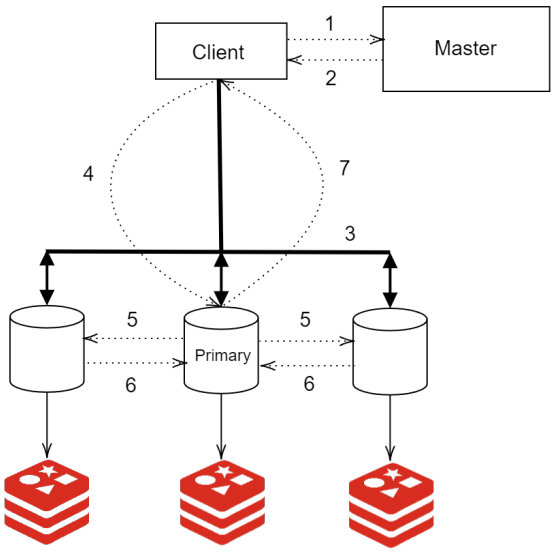

Figure 1: Our Setup

- **Chunkservers:** In our implementation, we store chunks at a chunkserver as key-value pair in a Redis instance, wherein the chunk handle is the key and the chunk data being the value.
- **Master:** Our master mimics the GFS master. It stores all the metadata for any given chunk, including the IP addresses of the chunkservers responsible for it, the primary for a chunk, the version number, and the lease of the primary chunkserver.
- **Client:** Our client provides a wrapper around the GFS functionality and provides a simple API for the GFS end user.

1.3 Terminology

Here is some terminology we have introduced in addition to the GFS paper, for ease of explaining the workflow of our implementation:

1. **Chunk Group:** A group of chunkservers (typically 3) which store a particular chunk.
2. **Chunk Group ID:** A unique identifier for a chunk group.

1.4 Write Data and Control Flow

As in the paper, we describe the complete workflow for a single write in GFS. Note that the bold arrows represent the data flow, and the dotted arrows represent the control flow.

1. The client sends a tuple `(filename, chunk_number)` to the GFS master, and requests for information on which chunk group is responsible for it.
2. The master finds the metadata corresponding to the given filename and chunk number, and responds with the `chunk_handle` and the other metadata, that is, IP addresses of the chunkservers responsible for it, the primary for the chunk, the version number, and the lease of the primary chunkserver.
3. The client then sends the data for the write to all of the chunkservers. It sends along with this a request ID. We have implemented a data forwarding mechanism for doing this efficiently.

4. After waiting for a fixed timeout, the client sends a write request to the primary. The request contains a request ID, the same as was contained in the message with which the data was pushed earlier to all of the replicas.

5. The primary decides an offset to write the data sent by the client. It updates its own file (Redis) and sends this offset to the secondary replicas.

6. The secondary replicas try to apply the write at the offset provided and sends an acknowledgement to the primary.

7. The primary informs the client about the write - whether it was successful, or one of the secondaries failed to write it. It also returns the offset at which the write was successfully written, so that the client may read this information later.

Note that a read is much simpler - a client simply contacts any one of the chunkservers, provides the byte-range which it wants to read. The chunkserver performs the read from Redis, and returns the requested data to the client.

2 SWIM Protocol

The Scalable Weakly Consistent Infection-style Process Group Membership Protocol [1], SWIM for short, is a gossip protocol which is used for verification of membership of a node in a group of nodes, and the efficient dissemination of membership information. Processes are monitored through an efficient peer-to-peer randomized probing protocol periodically.

The important feature of this protocol is that he expected time to first detection of each process failure, and the expected message load per group member, does not depend on the number of members of the group, and hence this protocol is *highly scalable* as the name suggests.

The SWIM approach consists of two components:

- **Failure Detection Component:** The method by which individual group members detect the failure of other members of the group
- **Dissemination Component:** The method by which individual group members inform other members of the group about the failure of one or more members of the group.

2.1 Failure Detection Component

The SWIM failure detection uses two parameters: protocol period T', and k, which is the size of failure detection subgroups. The algorithm proceeds as follows:

1. After every period T', each node M_1 sends a ping to a random group member M_2.

2. If M_1 receives an ACK from M_2 within worst case RTT, M_2 is marked as "healthy".

3. Else M_1 sends a request to k other members to ping M_2.

4. After the period T' expires, if M_1 has not received an ACK from the k members, it marks M_2 as "failed".

The psuedocode for the algorithm is given:

```
Integer pr; /* Local period number */
Every T' time units at Mi do:
0. pr := pr + 1
1. Select random member Mj
    1.1. Send a ping(Mi, Mj, pr) message to Mj
    1.2. Wait for the worst-ease message round-trip time for an ack(Mi, Mj, pr) message
2. If have not received an ack(Mi, My, pr) message yet:
```

```
    2.1. Select k members randomly from view
    2.2. Send each of them a ping-req(Mi, My, pr) message
    2.3. Wait for an ack(Mi, Mj, pr) message until the end of period pr
3. If have not received an ack(Mi, Mj, pr) message yet
    Declare Mj as failed
```

2.2 Dissemination Component

Upon detecting the failure of another group member, the member which detected the failure multicasts this information to the rest of the group as failed. A member receiving this message deletes the faulty node from its local membership list.

2.3 Example Run of SWIM

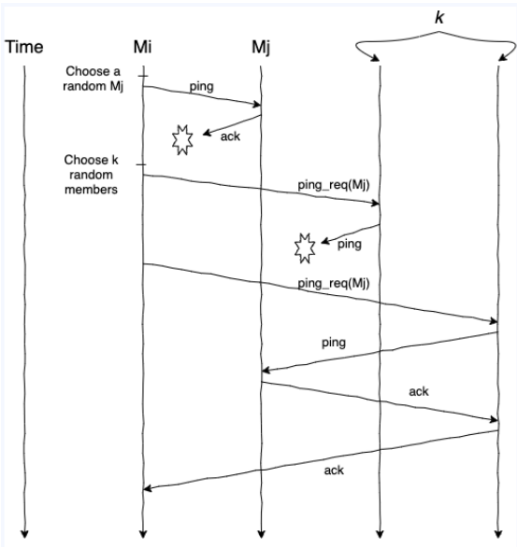

Figure 2: SWIM Protocol in Action

In Figure 2, we see an example run of the SWIM protocol. In the example, $k = 2$. Here, the node M_i chooses the node M_j at the beginning of the round, and pings it. The ACK packet from M_j is dropped. Therefore, M_i sends a ping request to k (here, two) other nodes. One of them is able to ping M_j successfully, and M_i receives an acknowledgement of the same. Therefore, despite several network faults, we are able to successfully detect the membership of M_j in the group.

2.4 Theoretical Results

Although the protocol is proabilistic in nature, it provides some very important theoretical guarantees. We will attempt to bound the time of detection of failure.

For any group member M_j, suppose we have that q_f is the fraction of non-faulty nodes. Then we have,

$$P[\text{at least one non-faulty member chooses to ping } M_j \text{ directly in a time interval } T']$$

$$= 1 - \left(1 - \frac{1}{n} \cdot q_f\right)^{n-1}$$

$$\simeq 1 - e^{-q_f} \text{ (since } n \gg 1)$$

Therefore, the probability that no faulty member chooses to ping M_j is roughly e^{-q_f}. So, the expression for expected time between a failure of member M_j and its detection is

$$\mathrm{E}[\mathcal{T}] = T'(1 - e^{-q_f}) + 2T'(1 - e^{-q_f})e^{-q_f} + 3T'(1 - e^{-q_f})(e^{-q_f})^2 + ...$$

Solving, we get:

$$\boxed{\mathrm{E}[\mathcal{T}] = T' \cdot \frac{1}{1 - e^{-q_f}} = T' \cdot \frac{e^{q_f}}{e^{q_f} - 1}}$$

Therefore, we get that the expected time for detection of failure depends only on q_f, which is the fraction of live nodes, which is also likely to be closer to 1. Further, the total number of group members does not show up in this expression. Thus, SWIM is highly efficient, and scalable.

Some other guarantees provided by SWIM Protocol:

- **Strong Completeness:** The crash-failure of any node in the group is eventually detected by all live members
- **Configurable Parameters for Speed and Accuracy:** The parameters k and T' can be configured using mathematical equations to give provable (expected) speed and accuracy to the protocol.
- **Uniform Load:** SWIM ensures that there is a uniform expected send/receive load at all group member.

2.5 Usage of SWIM in GFS

In the GFS implementation, we use the SWIM protocol on the master and a chunk group, whenever the master does not receive regular heartbeats from a chunkserver, which we call as the *suspicious chunkserver*. We run the SWIM protcol for a fixed number of rounds to figure out whether the *suspicious chunkserver* is alive or not, i.e. check the membership of the chunkserver in the chunk group. This process also populates the "routing tables" for the master and the chunkservers, which tells them which chunkservers are able to talk to which chunkservers. We can use this for forwarding of messages in GFS in case a specific network connection is broken.

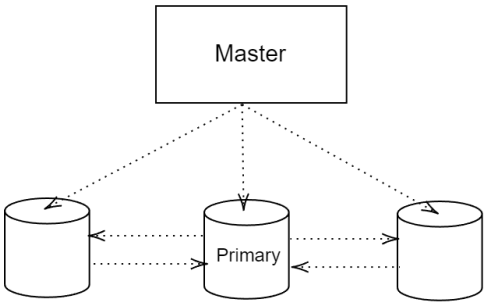

Figure 3: SWIM Protocol in GFS

3 Consensus Protocol - Delos Loglets

As mentioned in previous sections, we also wanted to possibly improve GFS by implementing a consensus algorithm, which would allow writes to go through even if they are successful at a majority of chunkservers. Writes in this case therefore, are simple - we can consider them as successful if a majority succeeds. However, we still need a good system for reads - now, the GFS guarantee that we can read from any of the chunkservers directly is no longer applicable - since different chunkservers could have different data. We need a good way to implement reads for the consensus algorithm.

For this purpose, we went through the paper "Virtual Consensus in Delos" by Mahesh Balakrishnan, et al [2], as suggested by Prof. Abhilash. We discuss our learnings and ideas in this section.

We may try to use the concept of *Native Loglets* as used for Delos, in the implementation of chunkerservers. In this system, we separate storage from the consensus. Therefore, we are not concerned with maintaining the state, just the log. This allows us to have holes in the log.

Let us suppose there are n LogServers. Each of them has its own local log. There is a global *Tail*, which is the first globally uncommitted log position. We also maintain a *knownTail* variable on each server, which is the global tail it knows about (the commit index). One of the servers acts as a sequencer. Here is how the different operations of the Native Loglet work:

- **Append:** The servers send append requests to the sequencer. The sequencer assigns a position in the log to each command and forwards the request to all LogServers. It considers the append globally committed and sends an ACK to the client once it receives successful responses from the majority of active LogServers.
  Each LogServer locally commits a particular log position n only after the previous position $n-1$ is locally committed on the same LogServer, or globally committed on a majority of LogServers.
- **Seal:** Any Loglet client can seal the NativeLoglet by contacting each LogServer to set its seal bit. If the seal completes successfully – i.e., a majority of LogServers respond – future appends are guaranteed to return an error code. Seal does not go through the sequencer.
- **CheckTail:** This returns the current global tail, essentially the commit index. Once we receive the commit index from a majority, we follow a slightly involved operation which uses a state machine. The following states are possible:
  - *All Sealed:* All LogServers whcih respond are sealed, then we return the local tail t from all the LogServers if it is the same for all. If we receive different tail from different servers, we do a read repair and copy over log entries upto t_{max} on LogServers from the LogServers which have them.
  - *Some Sealed:* Some LogServers which respond are sealed, then we issue a `seal` first, then reissue the `checkTail`.
  - *None Sealed:* In the case where none of the responding LogServers are sealed, the client picks the maximum position t_{max} and then waits for its *knownTail* to reach this position.
- **Reads:** Read restricts us to using a *ReadNext* functionality as follows: First the server tries to perform a fast local read. If it cannot find an entry locally, it issues a read to another LogServer.

As we can see, the implementation logic of `NativeLogLets` is much simpler than other consensus algorithm implementations such as RAFT. Further, the NativeLoglet concept fits in perfectly with GFS, since we are only concerned with appends to files, which can be seen as a log, rather than operations on data. Therefore, this is an extremely promising idea for the implementation of GFS via a consensus protocol. There are a few drawbacks of this approach: the sequencer is a single point of failure of the implementation. The GFS master however, can help with sequencer fault tolerance, similar to the way it helps with primary fault tolerance in vanilla Google File System.

4 Implementation Details

4.1 Master

The following are the implementation details of the GFS Master:

- `__init__`: This function initializes the Master object with the given host, port, and Redis port. It sets up two TCP sockets for communication with chunk servers and handling heartbeats. It initializes data structures for storing chunk metadata, and also starts threads for listening as well as heartbeating.

- `populate_maps`: This function populates the initial chunk group map using information from constants.py. It iterates over chunk server IPs and ports, creates chunk group IDs, and selects a primary server for each group. It also allots a lease time to each chunk group primary.
- `listen`: The listen function runs in an infinite loop, continuously receiving requests on the Master's main socket.It checks the type of the received request and calls corresponding handler functions
- `metadata_request_handler` This function handles and responds to metadata requests received from clients. If the requested file and chunk number combination is not in the files dictionary, it creates a new metadata record using `create_meta_record`.
- `create_meta_record`: This function creates a new metadata record for a given filename and chunk number. It generates a unique chunk handle and associates it with a randomly selected chunk group ID.
- `heartbeat`: The heartbeat function manages the periodic sending of heartbeat pings to all registered chunk servers. It keeps track of the number of consecutive missed heartbeats for each server in a map, `miss_history`. If a chunk server misses three consecutive heartbeats, it triggers switching to a new server, or in case of SWIM mode - running the SWIM protocol.
- `run_swim`: This function sends a request to all chunk servers in a chunk group to run the SWIM protocol to detect whether the suspicious node is alive or not, and also to populate their own routing tables.
- `swim_handler`: This function handles responses from chunk servers participating in the SWIM process. It marks the suspected chunk server as still operational based on the received response.
- `switch_server`: This function is responsible for handling the process of switching to a new server when a chunk server fails. It selects a new IP and port for the replacement server, updates data structures, and calls `updates_handler` to propagate the changes to other chunk servers.
- `update_primary_and_lease`: This function checks and updates the primary server and lease time for chunk groups based on lease expiration. If the lease time has expired, it updates the primary server and lease time and calls `updates_handler` to notify other chunk servers.
- `updates_handler` function: This function handles the process of updating chunk group information after a change in the primary server, i.e. version number, and/or the lease time. It updates the chunk group map, and then, for each chunk server in the group, it sends an update message containing the new information.

4.2 ChunkServer

The following are the implementation details of the ChunkServer:

- `__init__`: This function initializes a ChunkServer object with the provided host, port, Redis port, and other information about its chunk group. It sets up TCP sockets for communication with other chunkservers and handling heartbeats. It also spawns separate threads for listening (listen) and handling heartbeats (heartbeat).
- `listen` function: This function continuously listens for incoming requests on the ChunkServer's listener TCP socket. Based on the type of the received request, it calls the corresponding handler functions. (eg. `master_update_handler`, `write_fwd_handler`, etc)
- `client_read_handler`: This function handles read requests from clients. It retrieves the requested data from Redis based on the chunk handle and byte range, then sends a response to the client.

- `client_write_handler`: This function handles write requests from clients. It checks the sender's version, if there is a mismatch, it rejects the write. If the server is not the primary server, it forwards the write request to the primary server. Otherwise, it appends the data to the chunk in Redis, updates the offset, and notifies other servers in the chunk group.
- `write_fwd_handler`: Handles forwarded write requests from the primary. If the sender's version matches the server's version, it retrieves the data from the buffer and appends it to the chunk in Redis. Responds to the sender with the primary of the operation.
- `write_fwd_resp_handler`: This is a function for the primary, to handle responses to forwarded write requests to secondaries. If the response indicates success, it updates the pending write forward count. Once all forwarded writes are confirmed, it sends a *success* response to the client.
- `master_update_handler`: This function handles updates from the master server: it updates the ChunkServer's information, including the primary IP and port, version number, lease time, and IP list.
- `heartbeat`: It listens for incoming heartbeat requests from other servers. Based on the type of the heartbeat, it triggers actions such as starting the SWIM process (`swim_start`), handling SWIM requests (`swim_handler`), and handling SWIM responses (`swim_resp_handler`).
- `swim_start`: Initiates the SWIM process by sending a SWIM ping request to a *suspicious IP* provided by master..

4.3 Client

The GFS Client implements a client-side functionality for interacting with the Google File System. It handles metadata retrieval, write and read requests to chunk servers, and forwards data for storage. The implementation details are as follows:

- `__init__` function: This function initializes a GFS Client object with the provided host, port, master IP, and master port. It sets up a TCP socket for communication with the master server. It initializes an LRU cache to store metadata information about chunks, and also a map to track pending responses from the master. It spawns a separate thread for listening to messages.
- `listen`: This function listens continuously for incoming responses from the master server and the chunkservers, and based on the type of the received response, it calls the corresponding handler functions (`metadata_handler`, `write_response_handler`, etc).
- `metadata_handler`: This function handles metadata responses from the master. ON receiving info, we update the cache with metadata information, including the chunk handle, IP list, primary IP, version number, and lease time.
- `write_response_handler`: This function handles responses to write requests. First, it deletes the corresponding entry from the `pending_responses` dictionary. If the write operation is successful, it returns a `success` message along with the offset to the end user, else it triggers a metadata request and returns an error message.
- `read_response_handler`: This method handles responses to read requests. It deletes the corresponding entry from the `pending_responses` dictionary. Then, if the read operation is successful, it returns a success message along with the retrieved data; otherwise, it returns an error message.
- `metadata_request`: This function sends a metadata request to the master server for information about a specific chunk identified by the filename and chunk number.
- `write_request`: This function checks if the metadata needed for the operation is present in the cache; if not, it triggers a metadata request. Once the metadata is available, it sends a write request to the appropriate chunk server, including the data request ID and other necessary

information. It also tracks the request by inserting a unique ID in the `pending_responses` dictionary.

- `read_request`: This is similar to the `write_request` function but for read requests. It checks the metadata, triggers a metadata request if necessary, and sends a read request to the appropriate chunk server. It also tracks the request with a unique ID in the `pending_responses` dictionary.
- `data_forward`: This function forwards data to the chunk servers by writing directly to them. First, it ensures it has the correct metadata for a chunk. Generates a unique data request ID for tracking the data forwarding process. It uses a Lua script to read a specified amount of data from the file (`DATA_SIZE` bytes) starting from a given offset (`start_offset`). It sends this data to each chunk server in the IP list by using the `MIGRATE` functionality of Redis. The Lua script performs operations on the Redis database, including setting a new key, migrating the data, and then deleting the temporary key. It returns the generated data request ID for further tracking.
- `write_data`: This function is used for bulk writing of data. This reads data from a file specified by the filename, breaks it into partitions of size `DATA_SIZE`, and forwards each partition to the `data_forward` function. For each partition, it calls a `write_request` to the chunkserver in the group responsible for the chunk which this partition is a part of.

5 Results

All experiments were run keeping a chunk size of 20MB, with each chunkserver on 3 Baadal VMs, and Client and Master on a local machine (since they have lighter loads).

5.1 Normal GFS Implementation

To test our GFS implementation for end-to-end runtimes, we ran an experiment with a single client as well as with four clients, making writes concurrently. The plot is given below.

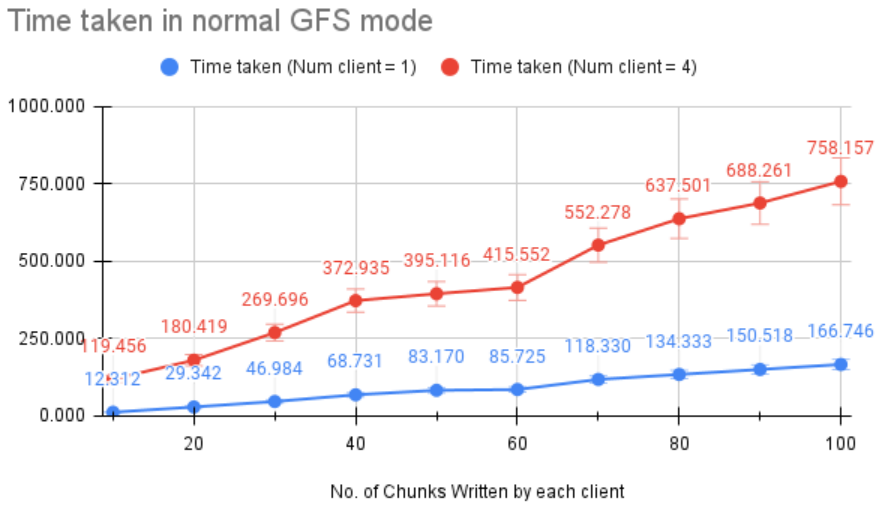

Figure 4: Time taken (in s) as a function of no. of chunks written ($n_{crashes} = 0$)

5.2 Recovery Times in Normal GFS vs GFS-SWIM

Here, we begin to compare our GFS-SWIM implementation with the vanilla GFS implementation to see where we can improve. For this purpose, we break a network link between master and chunkserver, and measure the time needed to "recover" from this fault. In case of vanilla GFS, we will need to replicate the chunk again, which will involve copying of the entire data into another chunkserver. There will also be a minor cost of sending updates to the chunkservers in the chunkgroup about the

new added chunkserver.

We can avoid this cost by using SWIM to detect that the chunkserver is indeed still alive, and it is the network that is down. In fact, this gives several orders of magnitude lower recovery time for larger data sizes, and remains nearly constant, which can be seen in the graph below.

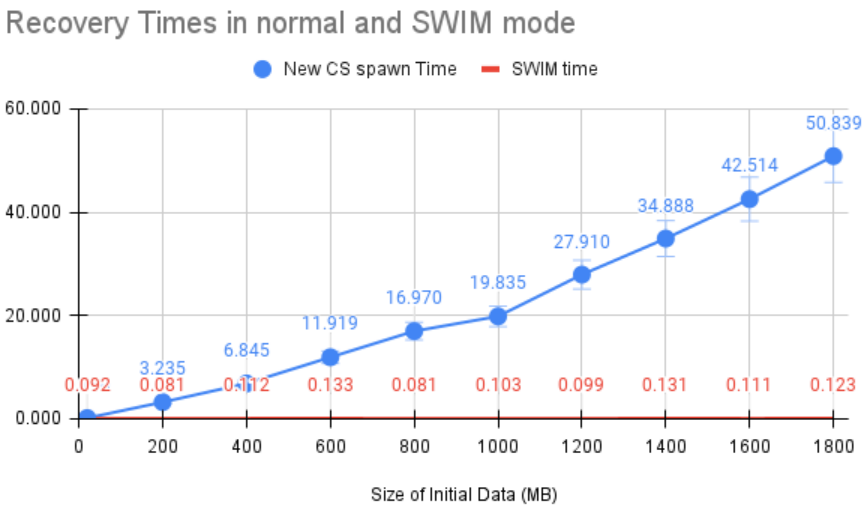

Figure 5: Recovery Time taken (in s) as a function of data size

5.3 GFS vs GFS-SWIM with Chunkserver Crashes

In the presence of chunkserver crashes, GFS and GFS-SWIM give nearly identical performance. This is because, if a chunkserver crashes, in GFS-SWIM we will only take a few extra RTTs to detect that the chunkserver actually crashed. Thus as we can see in the below graphs, the plots for GFS and GFS-SWIM nearly overlap.

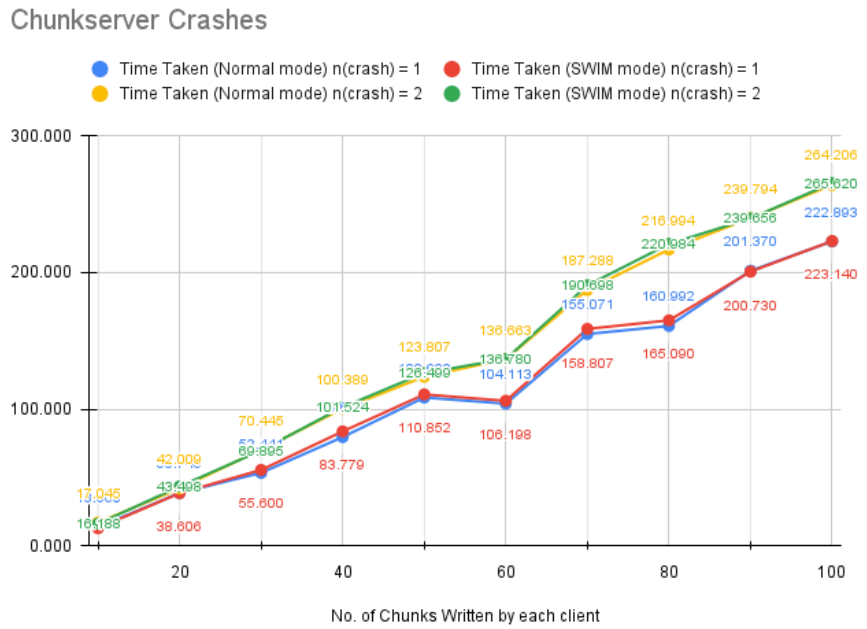

Figure 6: Time taken (in s) as a function of no. of chunks written

5.4 GFS vs GFS-SWIM with Network Crashes

In the presence of network crashes, GFS and GFS-SWIM give starkly different performance. This is because, if the network crashes, in GFS-SWIM we will be able to detect the same within a few RTTs but vanilla GFS will have to assume that the chunkserver actually crashed. Therefore, we see that GFS-SWIM gives much better performance in presence of crashes.

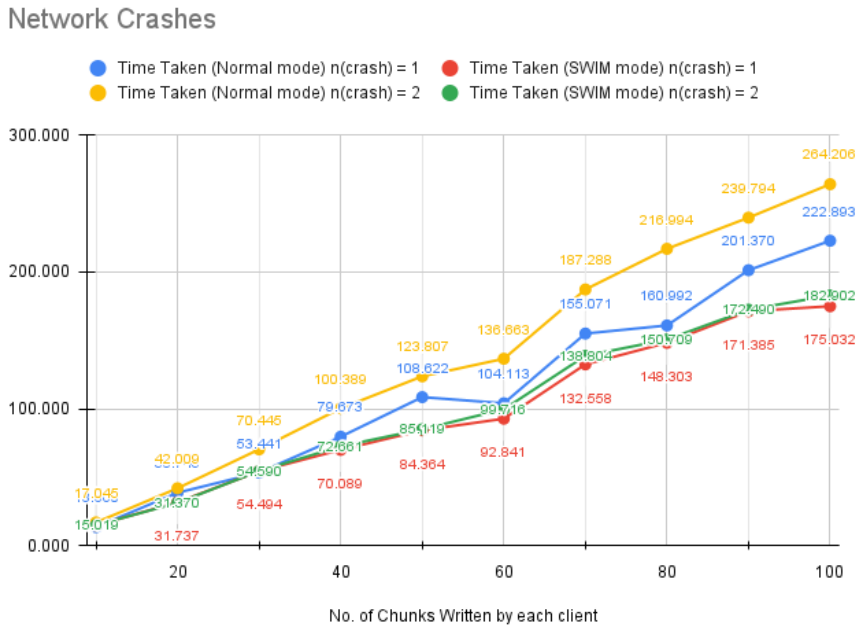

Figure 7: Time taken (in s) as a function of no. of chunks written

6 Pros and Cons

In this section, we will discuss the various pros and cons of the design decisions we took during the design and implementation of our modified Google File System:

6.1 Pros

1. Due to the implementation of the SWIM protocol, we are able to detect within a few RTTs whether a chunkserver is alive or dead. This saves us the effort of replicating the chunkserver again, which takes order of magnitudes higher, as seen in the results.

2. We are also able to implement data forwarding thanks to the SWIM protocol, using which we are able to construct our own routing tables, which may be used if a particular network connection is broken.

3. Using the consensus algorithm as used by NativeLoglets, we will be able to see a higher availability of the system, since we will be able to write even when write to *all* servers fail, as long as we are able to write to the majority.

4. We have ensured that we implemented GFS modularly. We have kept the networking portion independent from the control and data part of the GFS infrastructure.

5. We have used a minimal number of threads for our implementation of Master and Chunkservers, to prevent the overhead of thread creation and context switching from interfering with our end-to-end testing.

6.2 Cons

1. We did not model the network topology in our implementation of GFS. Therefore, we do not optimize for sending writes to the "closest" chunkserver by the client.
2. In some cases, the end-to-end runtime may not be optimized due to data forwarding. It may be the case that forwarding may be slower than actually retrying the writes.
3. Without deletion and garbage collection, the size of the Redis instances may keep increasing, due to which we may not be able to handle extremely high write workload.
4. The Consensus Implementation is not completely fault tolerant, due to the presence of the sequencer. We would need to handle sequencer fault tolerance using the Master to make it entirely fault tolerant like GFS.

7 Learning

In this section, we discuss the various learnings which came out of this project:

- **Understanding of GFS:** Via this project, we got a very good understanding of the intricacies involved in the implementation of the Google File System. We had to reread the GFS paper very carefully to ensure we did not miss any details, and we got an opportunity to think carefully about how to handle the corner cases for fault tolerance. This enhanced our understanding of GFS and justified its design choices.
- **SWIM Protocol:** We got a brief flavour of gossip or epidemic protocols in the course when learning about Dynamo, and learning about the SWIM protocol helped build upon that knowledge. Gossip protocols seem hard to argue about intuitively, but on reading the paper on SWIM, we also got to understand how to formally argue the effectiveness of such protocols.
- **Consensus & Delos NativeLoglets:** Thanks to Prof. Abhilash's suggestions, we also got to learn about a powerful technique for consensus called virtual consensus, which is able to walk the tightrope between efficiency as well as fault tolerance. In particular, we learnt about the implementation of consensus protocol in `NativeLoglet` for maintaining a shared log. We also enjoyed thinking about how to extend the implementation described in the Delos paper to Google File System.

8 Future Work

We leave the following scope for work to be done to improve the GFS implementation in the future:

- The implementation of the consensus protocol as in `NativeLoglet` is incomplete. We would like to complete of this implementation including fault tolerance of sequencer.
- Implementation of garbage collection and support for file deletion
- Use of FTP for transferring of large data
- Implementation of Master Fault Tolerance

References

[1] Das, A. and Gupta, I. and Motivala, A. (2002) *SWIM: scalable weakly-consistent infection-style process group membership protocol*, Proceedings International Conference on Dependable Systems and Networks.

[2] Mahesh Balakrishnan et al. *Virtual Consensus in Delos* 14th USENIX Symposium on Operating Systems Design and Implementation