GoLFS: A Fault-tolerant GFS Implementation

Tony Mu
Stanford University
tonymu@stanford.edu

Abstract— **The Google File System is a classic distributed file system that is both scalable and fault-tolerant. However, they have not open-sourced their code and the software is not available for public research and use [1]. The GoLFS system attempts to create an implementation of the Google File System using Golang. Due to the short duration of the course, however, GoLFS only contains most of the basic features, including all file operations, garbage collection, chunk replication, and master state recovery. Aside from features, unit tests are in place to ensure correctness. For evaluation, this paper mainly focuses on exploring the throughput and fault-tolerant properties of the system, using a probabilistic failure model to quantify master availability and simulate failures.**

I. Introduction

In the domain of distributed systems, system errors are frequent and inherent occurrences. Factors include network delays and packet losses, hardware failures, and human oversight. Consequently, in contexts where the CAP theorem finds relevance, it is common for modern distributed systems to prioritize partition-tolerance, while making trade-offs between availability and consistency. A notable example illustrating these principles is the Google File System (GFS), a robust distributed file system engineered to accommodate large files on commodity hardware [1].

The Google File System presents several wise design decisions and desirable properties. First, GFS is simple and easy to understand. At its core, GFS features a single master server serving as a centralized state manager. Additionally, GFS deliberately aims to reduce the interactions between the master server and file servers (referred to as chunk servers in the original paper). For state changes (such as filename update), the system employs a flexible lease mechanism, which hides away much complexity within the master server [1]. These decisions contribute to the straightforward and error-proof implementation of GFS.

Second, GFS tolerates numerous points of failure. Each file chunk is replicated across multiple chunk servers, typically three times as outlined in the original paper, with each replica residing on a distinct server (preferably in geographical regions). Furthermore, through the frequent exchanges of heartbeat messages between master and chunk servers, their states are updated and any failures is detected within the system. Upon receiving a deletion request from a client, the master merely marks the file as deleted, while the physical file is kept safe for a period of time before undergoing garbage collection. Although the master node represents a potential single point of failure, it mitigates this risk by only committing its state changes after persisting the change onto a hard disk [1].

Finally, GFS is strongly consistent. The design decision of a single master means there is no need for master replication. Consequently, this also eliminates the necessity for synchronization among master replicas. While this design choice may constrain throughput and availability, it guarantees the consistency of the file system state [1].

Although GFS has many commendable properties and design decisions, its initial design by Google engineers dates back to 2003, now over two decades ago [1]. In hindsight, certain design choices, such as single master, appear outdated and hinder GFS's ability to scale. Furthermore, while the system's design and architecture have been well-documented in an academic paper, access to the source code remains restricted and unavailable to the public. Thus, this paper attempts to demonstrates some of aforementioned properties through an implementation in Golang. This paper will also attempt to delve into some of the potentially

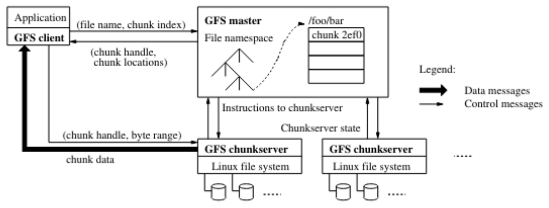

Fig. 1. GFS/GoLFS Architecture [1]

antiquated design decisions in later discussion.

II. System Architecture

The system architectures are well-documented in the original paper. Thus, this section will only briefly touch on some highlights in the original architecture. The rest of the section will mainly focus on differences in the original implementation and GoLFS implementation of GFS.

In the original Google File System (GFS) implementation, three primary components were specified: a single master node, chunk server nodes, and clients. The master node manages various aspects of the file system, including file metadata, chunk server metadata, and chunk metadata. Additionally, it handles tasks such as garbage collection of files and of failed chunk servers. Master is monitored and recovered by external processes. Clients communicate exclusively with the master for metadata operations and push physical file updates to chunk servers. Chunk servers establish connections with the master using a well-known IP or domain name within the cluster. Master and chunk servers frequently exchange heartbeat messages to update states and help master to detect failed chunk servers and out-dated chunks, as well as renew leases. Files are divided into fixed-size chunks, typically 64MB in size as specified in the original paper, each identified by a globally unique 64-bit integer known as a chunk ID. These chunks are replicated multiple times [1]. All these features have been implemented in GoLFS. The architecture can be found in (Figure 1) taken from the original GFS paper [1].

Additionally, the original paper introduces two valuable features: master snapshot and atomic record append. Reads are strategically served based on network topology to optimize response times. The master node is responsible for tasks such as re-replication, re-balancing of missing chunks, and garbage collection of outdated chunks. However, due to the project's limited scope, these features have not been implemented in GoLFS.

III. Implementation

While the original paper provides an overview for GFS, it leaves out some minor details. The following sections will outline detailed implementation of GoLFS based on the original paper. Each section specifies a major feature of the system. Many parts of the system will be highlighted and explained through the implementation notes.

A. Create

The GoLFS system works the following way for a typical create workflow. When the client receives an upload file call, it first computes the number of chunks based on the file size, and the fixed chunk size. Then, the client calls master's `Create` remote procedure, with the filename and number of chunks as arguments. The master will save this information in its internal state, by creating a new entry in the `FileMetadata` map with file name as key and chunkHandles as values, and a new entry in the `ChunkMetadata` with chunk handle as key, and chunk's server location as value. The master node then returns the chunk handle and chunk locations to the client. The client uses this data to call chunk server along with the file content. Each chunk server then saves the content at local disk, using chunk handle as file name, and acks client's request.

In GoLFS, the master will replicate the chunks once created, based on a pre-configured number. The master's reply will contain the address for all chunk replicas, including the primary. The client will then be responsible for uploading the chunk content to all replicas in different servers.

For chunk placement, the original paper has described several sophisticated strategies based on minimizing chunk server's chunk count and minimizing network distances. Due to time constraint on this project, GoLFS simply adopts a round robin strategy for chunk placement. This can easily be optimized by using a min-heap of all chunk servers at master, using chunk replica count as weight. We guarantee chunk placement will always be on the least loaded chunk servers.

B. Read

For the read operation, the client calls master's `Read` remote procedure with file name as argument. The master returns a list of chunk metadata which consists of chunk handle and chunk server address. The client uses this data to call each chunk server's `Read` remote procedure, and return the file content of that chunk. The client merges each chunk together to form the final file.

C. Write

The file mutation operation is the most complex of all basic file operations. To start, the client first asks master which chunk server has the lease for the chunk handle, and the location of all its replicas, by calling master's `GetPrimary` RPC. Master will reply with identity of the primary chunk server and all other replicas. If no chunk server has the lease for the chunk, the master will grant a chunk server of its choosing.

Once the client receives reply from master, it pushes data to all replicas in any order by using the `Write` RPC on the chunk servers. Chunk servers which receive the data caches the data in an LRU buffer without updating the chunks. Once all replica servers acknowledge the client about the reception of mutation data and return an update ID, the client will send a `CommitWrite` call to the primary chunk server with the update IDs. The primary chunk server will then commit the write in its chunks, and oversees all replica chunk servers to commit their own writes. Only after all replicas successfully commit their writes, the primary replica server will reply success to the client, and the mutation operation concludes here. If any of the replicas servers fails, the request is retried.

When granting leases, the master will attempt to make an intelligent decision when choosing chunk server. Since the primary chunk server will be responsible for communicating and propagating the update to all replica chunk servers, it makes sense to minimize the distance between it and all other chunk servers. This may be achieved by using a shortest path algorithm.

D. Delete

When a master's `Delete` RPC is called, it marks the file as to be deleted and hidden, by adding a dot prefix to its file name, then acknowledges the client request immediately. The file will be retained for a pre-configured period (three days in the original paper), then the garbage collection worker will ask the chunk server to remove these chunks permanently. More details below in the Garbage Collection section. During the file retention period, the client may still access the file through a special call, thus providing reliability and error-proof property to the client and applications.

E. File Garbage Collection

In GoLFS, the master will initialize a File Garbage Collection worker that runs on a timed schedule. After the master marks a file as to be deleted, it will also record the timestamp of the deletion request. When the worker wakes and perform its action, it will scan the master state's file metadata section, and if it finds any file with deletion timestamp older than the retention period (three days), it will send `Delete` requests to all the chunk servers holding these chunk replicas. After all chunk servers acknowledge the completion of the delete operation, the worker will remove the entry from the file metadata.

F. Chunk Server Failure Detection

The master will also initialize a Chunk Server Failure Detection worker that runs on a time schedule. This worker will periodically scan master state's chunk server metadata section, and if any chunk server's `LastPingTimeStamp` is more than a preset period, it will attempt to `Ping` the chunk server. If an acknoledgement is received, it will update the timestamp to the latest; otherwise, it will remove the chunk server entry from master state.

G. Failure Recovery

The master flushes its state to persistent storage after every state change in an atomic operation. This guarantees master state will always be consistent with some sacrifice in availability and latency. On master start, it will first check if any state can be recovered, and attempt to recover from an older state.

In GoLFS, an external monitor process acts as a both failure detector and a recovery tool in the case of master failure. It runs on a time schedule and keeps checking if master is running. Once it detects master is down, it will restart the master process.

H. Mutual Exclusion

Almost all system states are manage by the single master process, making synchronizing concurrent operations easy to program. In GoLFS, the master employees read-write locks on each of its metadata sections (File, Chunk, ChunkServer). This ensures the correctness of concurrent operations.

IV. Evaluation

Due to the scope of this paper, the evaluation of GoLFS focuses mainly on the fault-tolerant and throughput of the system. Each experiments is repeated with the same iterations, but different number of clients and master availability.

A. Methodology

Due to resource constraints, the entire cluster runs as a single virtual cluster. That is, all applications, clients, and servers run on the same physical node as processes, each has their own network address. The following measurements are collected on a GoLFS cluster which consists of a single master process and seven chunk server processes. Number of clients varies from two to eight based on the running experiments. Each clients will run twenty iterations before reporting the total time elapsed for its run. All clients start at the same time and run concurrently. The final duration is measured by the longest running client process.

An external process is setup to monitor master's health, and will restart the master if it can no longer detect master's presence. The monitor is a bash job and runs every second.

To simulate master failure, another external process is created. This process has a pre-defined availability parameter (from zero to one-hundred). Every three seconds, it randomly generates a number between zero and one-hundred, using a uniform distribution. If the randomly generated number is less than the availability parameter, the master is assumed to continue to live. However, if the randomly generated number exceeds the availability parameter, then the process will find and kill the master process, allowing the monitor process to come in and resurrect master, thus completing the master's failure-recovery loop.

B. Discussion

Two major experiments are conducted on the same cluster setup specified in the methodology section. First experiment focuses on read only traffic, while the second on write only traffic. Each experiment results is discussed below in details.

For the read only experiment, the number of clients has an impact on the latency of the replies (Figure 2). This demonstrates the lack of scalability of a single master: the entire system can be easily bound by master's compute power. To address this issue, some solutions are discussed in the Future Work section.

Another interesting observation from the read only experiment is that, availability does not have a visible impact on latency for master availability above 40%. However, when master availability drops below 40%, the latency increases drastically.

In the write only experiment, similar things can be said about impact of client count and availability (Figure 3). One interesting observation is that write has much lower read traffic. This is possibly due to the fact that the experiment uses writes of one megabyte, which is tiny compared to real-world, production workloads. However, it can be argued that the relative latency trend should remain similar even for larger write requests, due to the lightweight workload of master server, and an efficient chunk placement strategy among the chunk servers which will be doing the heavier chunk updates.

V. Related Work

Several attempts have been made by students of CS244B to implement the Google File System. Among which, the Google File System 2.0 [3] is the most mature and production ready. The authors leveraged many open source techonologies and solutions such as Docker and LevelDB, and employed strategies like multi-master to address some shortcomings of the original design of GFS. Another attempt was the Simplified GFS made in Python [2]. This project is lightweight and employs many features from the original GFS. GoLFS is more similar in scope as Simplified GFS project.

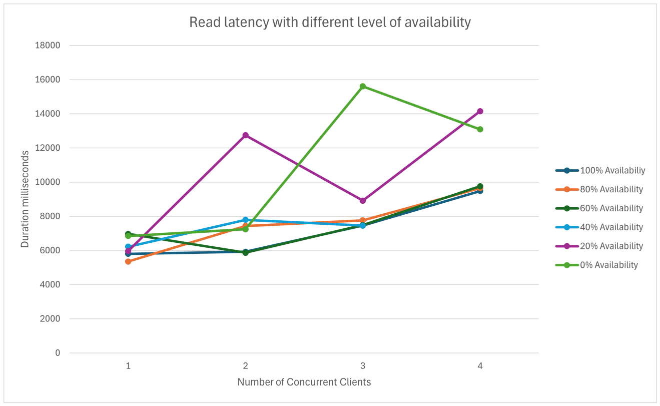

Fig. 2. Enter Caption

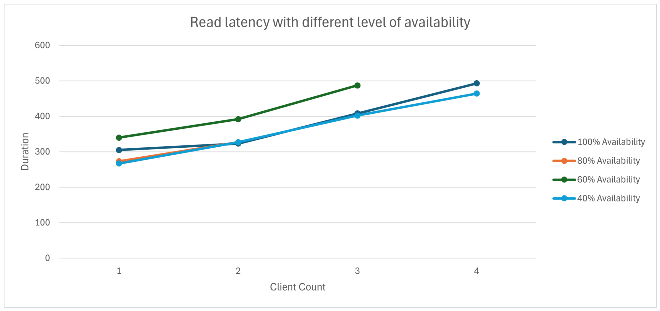

Fig. 3. Enter Caption

VI. Future Work

Due to time constraint of the semester, only limited features are implemented in this initial version of GoLFS. Some missing features include snapshot feature, atomic record append, chunk re-balancing and re-replication, as well as more efficient strategies for chunk placement, and primary chunk selection.

Aside from the missing features based on the original paper, a few potential items could further improve the GoLFS system. One improvement is to incorporate multiple master servers. The master cluster can be set up as either an active-active cluster, or an active-passive cluster. The master servers can synchronize with each other through a consensus algorithm such as Raft [4]. Having a multi-master architecture will undeniably increase scalability and further improve availability of the system. However, the system will have less of a consistency guarantee, due to some master may lag behind from synchronization latency. Further experiments are needed to determine the viability and performance impact of such approach.

An alternative approach is to decouple master server and master state. We may consider making master node a stateless serving layer, and use a distributed cache system like Redis [6], along with a distributed key-value store like Cassandra [5] to persist state. Using this approach, we are able to scale the stateless nodes if computation is a bottleneck, or scale the distributed storage if storage is a bottleneck.

Yet another improvement for GoLFS would be to implement an append-only logging system for master state recovery. Using an append-only log for master state persistence, as opposed to check-pointing, will drastically lower master's commit latency. While using an append-only log may slow down failure recovery speed, since modern hardware is more reliable than ever, hardware failures become more rare, so the drawback becomes negligible.

VII. Conclusion

The GoLFS project implements the GFS system as specified in the original paper from Google. It uses Golang and bash scripts to set up virtual clusters for experiments and testing. It demonstrates that, based on the original GFS design, GoLFS can achieve fault-tolerant properties without much impact on performance. The paper also discusses some potential workaround for known issues such as the bottleneck at master, and using an append-only log to persist master state.

VIII. Source Code

The GoLFS project is completely open-sourced, and the source code repository can be found on Github at `https://github.com/tonymuu/GLFS`.

References

[1] Ghemawat, S.; Gobioff, H.; Leung, S. (2003). "The Google File System".
[2] Lin, J. (2020). "Simplified GFS".
[3] Okutubo, B.; Tu, G.; Cheng, X. (2020). "Google File System 2.0: A Modern Design and Implementation".
[4] Ongaro, D.; Ousterhout, J. (2014). "In Search of an Understandable Consensus Algorithm".
[5] Apache Cassandra. (2024). https://cassandra.apache.org/doc/latest/
[6] Redis. (2024). https://redis.io/