GFS+Raft: Using Raft to Improve Fault Tolerance in GFS

Chavez Cheong
Duke University

Mark Liu
Duke University

Cristal Zhu
Duke University

Nathan Nguyen
Duke University

Abstract

The Google File System (GFS) is a distributed file system with a single master that implements fault tolerance through checkpointing. The Raft algorithm is a well-known protocol for implementing a fault-tolerant state machine. In this paper, we describe GFS+Raft, our implementation of a fault-tolerant GFS master server using the Raft protocol. By closely examining every operation GFS runs, we identify the metadata-altering operations that need to be replicated by Raft, and used a pre-lease mechanism to record the exact lease timer and lease holder before granting a lease to eliminate the non-determinism in GFS. We present a summary of the design and implementation of GFS+Raft, a description of our testing approach, and our quantitative analysis of the fault tolerance of GFS+Raft, the performance penalty it incurs in comparison to the single-master GFS, and the effects of tuning different Raft parameters on the performance of GFS+Raft. On the whole, we find that GFS+Raft does noticeably improve the fault tolerance of GFS, but incurs a significant performance penalty in the process.

1 Introduction

Our project aims to use the Raft protocol to improve the fault tolerance of GFS' master servers, a larger-scale system than the key-value stores that Raft is usually implemented on, and in doing we implemented Raft. Our goals are (1) to improve GFS by creating a fault-tolerant distributed master server using Raft, (2) to evaluate the costs of introducing this fault tolerance in terms of latency and throughput, and (3) to investigate the effects of tuning different Raft parameters (election timeout, heartbeat frequency, and raft cluster size) on the performance of the system (availability to serve requests, average time to elect leader, latency, throughput).

In Section 2, we describe the background of GFS, the current limitations of the system that we would like to improve on, and the background of Raft. In Section 3, we provide a design overview for our fault-tolerant GFS+Raft, including the overall architecture of the existing GFS, required modifications to achieve fault tolerance, and the design decisions we took to handle the non-determinism of GFS. In Section 4, we provide a detailed explanation of the implementation. Section 5 describes our testing and deployment architecture. Section 6 presents results and metrics. In Section 7, we discuss related work in the field. Section 8 explores future work. In Section 9, we give a detailed description of team member contributions. We summarize our work in Section 10.

2 Background

GFS is a distributed file system designed by Google to offer fault tolerance while operating on cheap commodity hardware. The original GFS design involves a single master server and many *chunkservers*, which store all of the data on the file system in fixed-size chunks. Clients communicate with this single master server, which contains all of the metadata necessary for authenticating and directing users to their desired files on the file system. [1]

GFS has key limitations in both scalability and fault tolerance due to the single master server. GFS struggles to scale because the single master must serve all client metadata requests in the cluster. Despite the design purposely limiting the master's involvement in serving client requests, Google still found it difficult to scale the single master to meet exponentially increasing user demand. Additionally, the presence of a single master server presents a single point of failure. The original design does mention a log file recovery system, but this is slow and repeated failures have the possibility of wiping the system.

Raft is a consensus algorithm that manages a replicated log for a cluster of servers in order to ensure replicated state machines. The system's history is subdivided into a series of elections, where each election has one leader. This leader is in charge of replicating the log of each server and of committing changes once a quorum of servers agrees on a change. Raft has been used in many production systems such as CockroachDB, MongoDB, and Splunk. [4]

3 Design

This section describes the existing infrastructure and the motivations behind the changes that we made to the infrastructure. It then moves on to describe the overall modifications required to support fault tolerance in the master and the justifications behind the modifications. We end this section with a discussion on the use of leases while handling the non-determinism of GFS.

3.1 Existing Architecture of GFS

There are three main components of the existing architecture: the Master Server, the Client, and the Chunk Server. The Master Server implements one singleton Locking service and three gRPC services: Metadata Manager (handles client file creation/deletion requests and requests for metadata), Chunk Server Manager (handles file creation/deletion between master and *chunkservers*), and Chunk Server Heartbeat Monitor Task (heartbeats between master and *chunkservers*). The Client provides a client-side cache and file system APIs. The Chunk Server provides one Chunk Server Caching Service and two gRPC services: File Chunk Manager (handles read/write requests from clients) and Write Lease Manager (provides leases to *chunkservers*. The existing architecture of the GFS codebase we are working on is described in Figure 1.

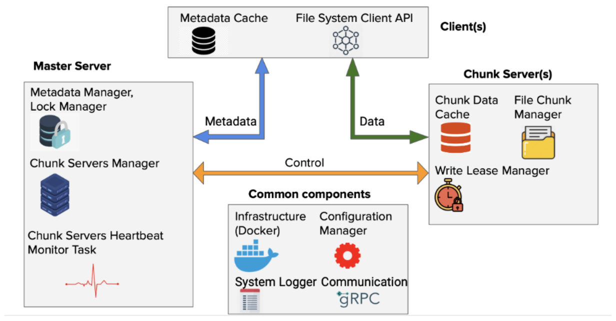

Figure 1: Existing high-level architecture diagram

3.2 Required Modifications

In GFS, the state that is stored by the master server is that of the metadata. Meanwhile, the Raft log needs to record operations that need to persist across primary failures. Thus, we needed to store and replicate operations that change the metadata stored in the master. We distinguish between RPCs that need to be replicated and those that do not in Table 1. For example, reading or writing to chunks for *chunkservers* do not need to be replicated since this operation has no effect on the metadata, but client creating or deleting files requires replication as this affects the chunk handle metadata in the master server.

Figure 2 shows an overall architecture diagram for our updated design for GFS+Raft. All operations that edit the state of the metadata now are routed through the Raft service. For example, when the master wants to update a chunk handle, it goes through the Raft service first before editing the metadata in the metadata manager, finally sending the updated file chunk to the appropriate *chunkserver*. Other messages that do not involve metadata stay the same - for example, the heartbeat monitor on the master can freely send heartbeat messages to *chunkservers*.

All of the appropriate variables such as the Raft log and the current term of each server are persisted using the Raft Service Log Manager, which stores the data for Raft onto a LevelDB disk storage.

RPC Classification for Different Components)		
Component	Replicated	Non-Replicated
Master	Lease Request	Heartbeat
Chunk	Lease Request	Read/Write
Client	Create/Delete	Read/Write

Table 1: Classification of Operations into Replicated and Non-Replicated Operations

3.3 Handling Nondeterminism with Leases

In the original GFS design, leases are granted to *chunkservers* upon request, where the master server stores the timestamp of the lease alongside which *chunkservers* was granted the lease, which means that leases are inherently nondeterministic. In addition, when we apply any client Create/Delete requests to the state machine, the master server grants a lease to a random *chunkservers* in the replica group, which is a nondeterministic metadata operation that must be replicated using Raft.

We cannot have nondeterministic operations when we use Raft due to the possibility of state machines diverging after applying the same log. We can solve this using the same idea as the "Primary-Backup Replication" paper, where we pass along the non-deterministic elements of the state-machine operations to the replicas. In this case, when we need to grant a lease either upon request or for file creation, we first add to the log which server should receive the lease and the lease expiration time. This should ensure that if another leader is elected, the same *chunkservers* are considered to have a lease with consistent expiration times. [3]

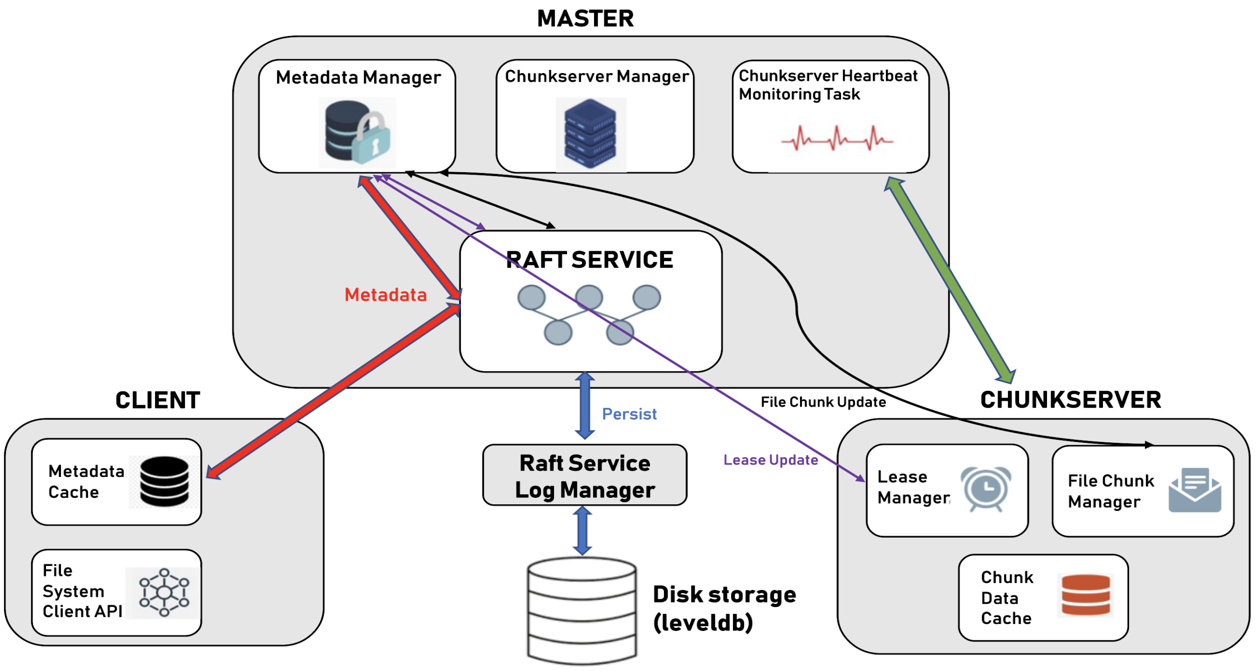

Figure 2: New high-level architecture diagram

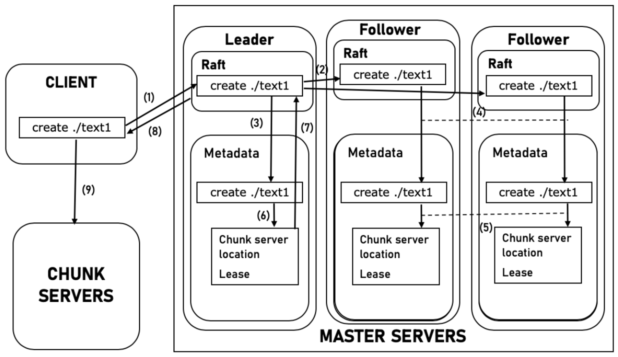

Figure 3: Client-Master paradigm

3.4 Client Workflow

Figure 3 shows an example of how a client will typically communicate with GFS+Raft.

First, if a client wants to create a file with name text1, (1) it will first send their create file request to the Raft master server who is the leader. Then, (2) the Raft leader will add this request to the log and send AppendEntries requests to the follower servers to replicate the new log entry. When the entry is committed, (3) the leader and (4) the follower servers will apply the entry to their state machine and (5) the followers and (6) the leader should end with the same output. However, (7) only the leader should send back the output to the Raft service and (8) send the reply for the operation back to the client. Finally, (9) the client can communicate with the *chunkservers* responsible for text1 independently without communication with the master servers.

4 Implementation

The source code of GFS+Raft can be found here. To add fault-tolerance through Raft, we made five modifications to their architecture: (1) Raft Service, (2) GFS Client, (3) Metadata Manager, (4) Lease Service, (5) Raft Service Log Manager.

4.1 Raft Service

Our implementation focuses on implementing Raft as a gRPC service to run on top of each replicated master server.

Instead of running a single master server, as the current architecture does, we initialize multiple master servers, each running Raft Service. The initial total number of master servers in the Raft cluster is statically set at initialization by configuration and cannot be dynamically modified. Future work could be to implement an old/new configuration in order to allow for dynamic Raft configurations during runtime.

This service calls RequestVote RPCs to elect a leader of the Raft cluster on election timeout. After a leader is elected, the leader sends AppendEntries RPCs to all followers both to update their logs and to serve as a heartbeat to prevent further elections.

We design the request flow so that all metadata change requests from clients (create/delete file) and from chunks (lease expiry), as well as from the master when there are *chunkserver* updates or *chunkserver* failures are redirected as RPC requests to the Raft service instead of their original destination, where they will be replicated across the Raft cluster of master servers.

Then, the operation only begins when the Raft Service commits the log. To facilitate fault tolerance, when applying operations in the log, we iterate through the log operations sequentially and only apply the operations in the Raft Service when the metadata update is successful. This may possibly result in duplication of metadata update requests if a master server shuts down before application to state machine can complete, but this should be fine because metadata changes are idempotent.

In our current design, applying to state machines is synchronous for leader master servers. This is to ensure that the client only gets the result from a master that has already applied its metadata updates.

In the case of bringing a long-crashed master server up to date, there may be a large number of log requests that need to be applied to state machines, which may result in Raft significantly slowing down because the Raft cluster is blocked while the newly-restarted master servers are applying their responses and not able to respond to RPCs. One way of tackling this would be to apply committed log entries to the state machine asynchronously, so as to not block any further sending of AppendEntries. However, this could potentially cause high latency when replying to clients.

4.2 GFS Client

Previously, the GFS client interacted directly with the Metadata Manager service on the master server to create/delete files and request file chunk metadata. However, with the Raft Service implemented as a wrapper service around all metadata requests sent to master servers, we modified the GFS client to contact the Raft service instead.

We initialize the GFS client with access to all the addresses of the master servers in a parallel hash map. Before sending any request to the master server, the GFS client checks the identity of the leader with the Raft service and updates the latest leader in its cache. It does this in two ways, it sends a request to determine the current leader using the current leader it holds in its cache and updates the latest leader in its cache using the reply from the master server. If the current leader identity request times out, it will sequentially iterate through the master servers in its hash map and send requests for the identity of the current leader to update its cache. It then sends the requests to the Raft service of the leader, which forwards the requests to its replicas. Should a request fail or timeout, the GFS client will invalidate the current leader in its cache and continue looking for the current leader using the mechanism described above.

As the request to get a leader does not modify any metadata or state, we made it non-replicated so the master servers can reply to the client directly without replicating this request.

4.3 Metadata Manager

Previously, metadata requests and file creation/deletion requests from clients to the master server were managed by a combination of the singleton Metadata Manager class and the Master Metadata gRPC service. However, with our Raft wrapper layer, metadata operations are now local operations to be conducted when their log entries are applied. Therefore, we move away from the functionality of the Master Metadata gRPC service and refactor Metadata Manager to support calls from Raft Service when applying log entries.

In addition, the implementation of the Metadata Manager performed multiple metadata operations in a single function call, including nondeterministic behavior like selecting *chunkservers* and expiration timestamps for leases. Thus, we needed to refactor Metadata Manager to only modify certain portions of metadata at a time to preserve the state machine behavior of the system. We described how GFS+Raft handles this in the next section.

4.4 Lease Service and Pre-Lease Mechanism

The Lease Service is a gRPC service running on the *chunkservers* which the master uses to grant a lease to the primary *chunkserver* for a chunk. Previously, the master could respond to the lease requests directly or grant leases directly to *chunkservers* at will. As this is a metadata-modifying operation, GFS+Raft reroutes these requests to the Raft Service, but as leases are non-deterministic operations, they will need to be handled differently to avoid state machine divergence.

When a request for a lease comes into the Raft Service, the GFS+Raft Pre-Lease Mechanism first checks if an existing lease already exists for the chunk, and returns this directly without any log replication to avoid unnecessary operations. If an existing lease does not exist or the previous lease is expired, we first prefetch a list of *chunkservers* using the Metadata Manager and set the lease expiration time before handling the file creation request replication at all. Then, we can add the *chunkservers* replica group to the log alongside the file creation request, resulting in a deterministic behavior when applying the log.

This function could be troublesome for Raft, because it is time-based, and if there is a long queue of Raft requests that need to be committed, the lease could be expired by the time the request is sent out. We mitigate this by setting a longer lease expiration when we create the log entry. This could result in less availability if *chunkservers* crash frequently, but this case is more unlikely and we are willing to make this tradeoff as it results in higher availability in the likely case.

4.5 Raft Service Log Manager

To ensure fault tolerance in master servers, some memory has to remain persistent on disk. In particular, the Raft protocol requires that the latest **votedFor**, latest **currentTerm** and latest **log[]** is stored on disk.

We achieve this by using LevelDB as persistent storage to store the memory as keys on disk. As LevelDB only accepts bytes or strings as input keys, we implemented a series of serialization and deserialization methods to convert custom Protobuf datatypes and other classes into strings to be stored on LevelDB.

The Raft Service Log Manager is a singleton class that acts as an interface for the Raft Service to write its own Raft-related metadata like the log or latest **currentTerm** to disk, and also for the Metadata Manager to write metadata changes to disk.

5 Testing and Deployment Architecture

To facilitate a rapid prototyping and testing environment, we ran our master servers and *chunkservers* on localhost. We then used batch scripts to simulate different workloads on the server and checked the success of each operation to verify correctness of Raft. We also implemented unit tests to test the functionality of integration with other libraries and services, such as LevelDB for our Raft Service Log Manager.

6 Results and Metrics

As a coarse estimation for benchmarking, we ran multiple different workloads for a variety of different configurations of GFS+Raft. We primarily utilized shell scripts that push a workload of file creates, writes, and reads of varying weight. The scripts benchmark the latency and throughput performance on a Raft master cluster and three *chunkservers*, with three replicas per file. We also experiment with tuning the Raft variables to see the effect on performance, including the number of master servers in the cluster, the Raft election timeout, and the leader's heartbeat frequency. To eliminate differences in machine processing, we ran all the workloads on the same machine running Windows Subsystem for Linux (WSL2) with 16GB RAM.

6.1 Latency and Throughput Comparisons

We ran five different workloads to compare the original implementation to GFS + Raft: (1) A create only workload with 300 file create requests, (2) A create-write workload with 300 sequential file creation requests with each followed with a write request, (3) a write-heavy workload, with one file create followed by 300 file writes to that file, (4) a read-heavy workload with one file create, one file write, followed by 300 read requests to that file, and (5) a create-write-read workload, where we had a file create request followed by a write and a read to that file 300 times.

Our results can be seen in Figure 4. The largest performance penalty was for write requests. This can be seen in the write-heavy workloads (2), (3), and (5). We reason that this was due to the lease mechanism adding a lot of overhead for writes, since we needed replicate log entries whenever we granted a lease due to a file write. In addition, since the lease request is likely to expire due to the time difference between setting the time and log application, this possibly resulted in more new lease creation and thus more log replication.

File create requests incurred a moderate performance penalty compared to the original system. This was again due to log replication as file create requests modify metadata in the master. However, it did not have the additional overhead of granting and checking leases so the performance penalty incurred was lower than the write requests, as seen by (1).

Read requests only suffered a minor performance penalty (as seen on workload (4)) compared to creates and writes, which was due to read requests not needing to be replicated on the Raft log. However, there was some overhead in general for the GFS + Raft due to locking and rerouting RPCs, so there was still some read performance penalty.

6.2 Parameter Tuning

6.2.1 Election Timeout

We experimented with using different election timeout range for Raft to see if election timeout can affect latency of GFS + Raft. As we did not crash any master in this test, we expect that the election timeout will not greatly affect the latency and throughput of client requests.

We performed experiments with three ranges of election timeout $(500, 1000), (1000, 2000)$, and $(2000, 3000)$, where election timeout of each master is randomly chosen between the range. The result was analyzed in figure 5. As expected, we did not observe any relation between election timeout and our client request latency.

6.2.2 Heartbeat Frequency

We experimented with increasing the heartbeat frequency for the leader of the Raft cluster to determine if increased frequency would somehow incur a system performance penalty when coupled with client requests. The benefit of increasing the heartbeat frequency would be a lower likelihood of false election timeouts occurring, but the increased amount of RPCs and thus locking of the system could be a possible downside.

From our results in Figure 6, it seemed that increasing or decreasing the heartbeat timer frequency had little to no effect on throughput or latency. This may be due to the frequency of our workload being far greater than the highest-frequency heartbeat timer that we tested.

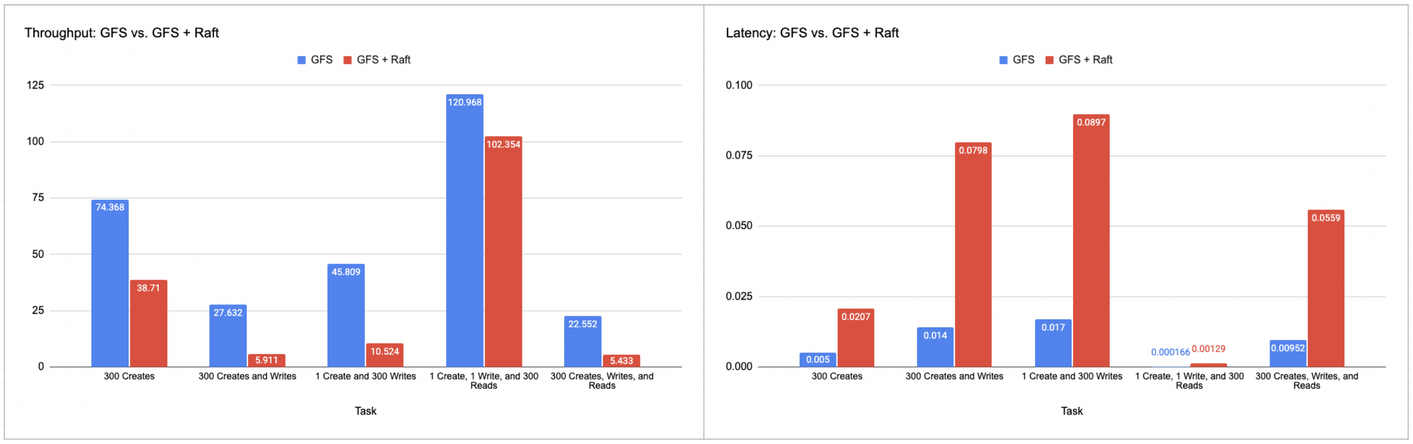

Figure 4: Throughput and Latency Comparison

Election Timeout /s	Throughput	Latency /s
{1000, 2000}	55.463	0.0115
{2000, 3000}	50.31	0.0123
{500, 1000}	55.076	0.0113

Figure 5: Election Timeout

Heartbeat Timer /s	Throughput	Latency /s
100	27.985	0.0357
200	24.53	0.041
500	27.523	0.0363

Figure 6: Heartbeat Frequency

6.2.3 Raft Cluster Size

We also experimented with changing the size of the Raft cluster by increasing the number of master servers. Increasing the number of master servers would result in higher fault tolerance and higher consistency but we predicted there might be a significant performance penalty.

The results can be seen in Figure 7. Increasing the number of Raft masters from 3 to 5 does incur around a 10% decrease in throughput, and increasing from 5 to 7 incurred around a 4% decrease in throughput. This suggests that increasing the size of the cluster does have some performance penalty as expected, since more master servers means more RPCs must be sent and more servers need to replicate a log entry for that entry to be committed. In addition, since we ran this workload locally, there could be a performance penalty from running multiple master server instances on the same machine.

Number of Raft Masters	Throughput	Latency /s
3	27.985	0.0357
5	25	0.04
7	23.962	0.0417

Figure 7: Raft Cluster Size

6.3 Fault Tolerance during Leader Crashes

To test the fault tolerance of GFS+Raft, we ran the create only workload with 300 file creation requests while crashing the leader in the middle of the workload, and monitored the operation completion status. Over 10 trials, all 300 files were successfully created, with an average total time of 9.204s real to complete the workload for a throughput of 32.595 requests per second compared to the throughput of 38.71 requests per second. The decrease in throughput is due to the time it takes to elect a leader after a leader crashes, but the consistent successes indicate that GFS+Raft was able to successfully handle a workload with leader crashes.

6.4 Maximal Workload

There was a sharp availability dropoff for requests after the maximum number of chunks allocated has been reached, which our configuration has at 320 chunks with 64 MB allocated per chunk, or a total of 20480 MB.

7 Related Work

Through our research, there was no other implementation of GFS that uses a Raft / Paxos-like consensus algorithm to achieve fault tolerance for the GFS master server. However, there were other general storage systems with fault tolerance that we have reviewed in the space.

7.1 Google Colossus

Google's solution to GFS' scaling limitations was to implement a new storage system called Colossus. Colossus specifically targeted metadata scaling by replacing a single master server with a distributed service hosted on a distributed storage solution called BigTable. .

7.2 TiDB

TiDB is a Raft-based Hybrid Transactional and Analytical Processing (HTAP) database that consists of a row store (TiKV) and a column store (TiFlash) [**?**]. TiKV stores data in an ordered key-value map and uses Raft to maintain consistency among replicas. Unlike TiDB which replicates the actual data, our work focuses on improving the fault tolerance of GFS by replicating metadata instead. The master server replicas only need to handle the metadata while the *chunkservers* handle the actual data, allowing for better performances. [2]

8 Future Work

In our base implementation, even though we have an on-memory log, every log append or modification operation is persisted to disk. While LevelDB is very fast, it is still significantly slower than reading from memory. We hope to modify our implementation to batch the disk updates to reduce the number of disks reads or writes and improve the performance of the master server Raft cluster.

While implementing Raft on the master does indeed increase fault tolerance, it also introduces additional latency. One of the other limitations of the single master GFS was that the single source of metadata could serve as a bottleneck to limit scaling. Thus, to overcome this limitation, we can introduce multi-master sharding of metadata such that different client requests will go to different masters. This can further make up for the latency increases introduced by Raft by decreasing the workload on each primary server, especially at high workloads.

Additionally, the scale of our project only focuses on running tests and starting servers on local hosts, but we didn't run it on a real network. We could extend our benchmarks by testing how GFS+Raft performs under unstable network conditions with varying round-trip times (RTTs) and network partitions.

9 Team Member Contributions

In this section, we detail the progress each team member made. The team made 156 commits, adding almost 3000 lines of C++, Protobuf, and Bash code for implementation, unit tests, and shell script workload tests and benchmarks.

9.1 Chavez

Chavez implemented a singleton class for the Raft Service Log Manager to ensure that Raft metadata and GFS metadata on each master could be persisted on disk even after crashes using LevelDB. The singleton class provided thread-safe interfaces for the Raft gRPC service: current term (get, update), voted for (get, update), log (get, append, delete). He also wrote unit tests using GTest to ensure the correctness of the singleton instance. He then integrated the Raft Service Log Manager with the Raft Service, deciding which points to persist the Raft variables and log to disk,

He worked alongside Mark and Nathan to implement and debug the core Raft Service implementation. In particular, he implemented the logic to update leader commit index and also the logic for when followers need to apply the log to their state machine.

He also integrated the existing Master Metadata Service code with the underlying Raft logic to ensure correct behavior. In particular, he wrote the code that handled the logic when the leader received a client request. He modified the existing CreateFile and Grant Lease implementation in the Metadata Service to differentiate between the leader applying a state machine (sending messages to *chunkservers*) and followers applying log entries (updating metadata without sending messages).

Chavez wrote the pre-lease mechanism that checked if a file chunk had an existing lease before deciding on a suitable *chunkserver* and expiration time that will be applied.

Lastly, Chavez wrote and tested the five shell scripts containing different workloads that were used to test and benchmark GFS+Raft.

9.2 Mark

Mark first worked on implementing the logic for RequestVote for the sender and receiver and AppendEntries for the sender, as well as added RPC Protobuf definitions for all Raft-related messages. This included adding some infrastructure to use futures to communicate with many Raft servers asynchronously. He also added client side code in the RaftServiceClient class to facilitate client interaction with the Raft service. He then initially added locking for the Raft service after having some inconsistent results for some test runs of the code.

He then worked on log replication with the AppendEntries sender and made sure that leaders could replicate their log entries to followers and apply committed log entries to the state machine. He also added code for timer in the Raft service to facilitate Raft heartbeat messages and election timeouts. He created RPC channels for different master servers to communicate with clients and *chunkservers*.

He was involved in the debugging process for the Raft service as well as debugging the original GFS implementation and helped consolidate the results with the shell scripts.

9.3 Cristal

Cristal worked on refactoring the configuration system to support the initialization of multiple masters. She also worked on routing requests from the client server to the Raft cluster. Now with multiple master servers that are organized into a parallel hash map, the client does not necessarily know the address of the leader master server. Hence, Cristal decided to send client requests to one layer above. Clients query the Raft service of the latest leader for the updated identity of the leader and send requests to the Raft service of the current leader. The Raft service then communicates with other services, such as the master server metadata service, to handle requests such as creating, deleting, and editing files.

Cristal also helped with the delete file request handling for the Raft service. She did so mainly by adding a handler that logs the delete file requests, sends AppendEntries requests to the follower servers to replicate the new log entry, and returns gRPC call status in the implementation of the Raft service.

Other than involving in the debugging processes throughout the project, Cristal also ran the shell scripts for benchmarking with Nathan. She then calculated the metrics, including throughput and latency, and created tables and graphs that were used in the final report. Cristal also structured and made the presentation slides, as well as reorganizing the final report.

9.4 Nathan

Nathan worked with Mark on Raft service. In particular, he partly wrote the AppendEntries RPC handler and some parts of leader/candidate conversion of masters, along with Mark. He also wrote the election and heartbeat timeout function for candidates upon election and for leader. He also used Chavez's Raft Service Log Manager to persist. He added locking to some parts of Raft Service to ensure that it would achieve consistency.

Nathan also worked partly on disconnecting direct interaction from clients to Master Metadata Service and redirected the communication to Raft instead. Particularly, he implemented the handler in Raft for open file requests and delete file requests, and direct that to master metadata to modify and retrieve the *chunkserver* information.

Nathan was also involved in various debugging processes for Raft Service and protocol clients. He also wrote a Python notebook to analyze testing and benchmarking results. Nathan also ran the script for testing workload, analyzing tests for heartbeat frequency and raft cluster size.

10 Conclusion

We have demonstrated how to increase fault tolerance on a single-master GFS, with certain trade-offs in latency and throughput. GFS+Raft provides the benefit of handling clients' requests robustly, even when the master leader crashes or becomes unavailable, compared to the original single-master GFS, especially when the master fails for an extended period.

We learned that the most significant trade-off between GFS + Raft and single-master GFS is that single-master GFS achieves high throughput by allowing metadata creates and edits to run concurrently, whereas for GFS + Raft, we need to linearize all metadata requests, such as create file, delete file, or chunk server updates, resulting in significantly slower latency.

Despite not having good latency as compared to the single-master GFS, the added fault-tolerance of GFS+Raft allows us to run the master servers on cheaper commodity hardware instead of the more specialized, durable hardware that a regular GFS master would need to run on. Therefore, it depends on the purpose of future systems and hardware architectures whether our system can bring benefits over the original GFS.

References

[1] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 20–43, Bolton Landing, NY, 2003.

[2] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. Tidb: A raft-based htap database. *Proc. VLDB Endow.*, 13(12):3072–3084, aug 2020.

[3] Rui Oliveira, José Pereira, and André Schiper. Primary-backup replication: From a time-free protocol to a time-based implementation. pages 14–23, 02 2001.

[4] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 305–320, USA, 2014. USENIX Association.