GFS Simple Implementation

Andy L. Khuu
Stanford University
andykhuu@stanford.edu

Vincent J. Heng
Stanford University
vheng@stanford.edu

Abstract

The Google File System is a scalable and distributed file system developed at Google that provides high performance operations, and fault tolerance. In this paper, we implemented GFS Simple Implementation, a light localized version of GFS in Golang, with the goal of recreating key features of GFS to explore how certain design decisions and configurations affect overall performance under various workloads. Our system supports concurrent CRUD requests from multiple clients. As part of our testing, we focused on bench marking the performance of the system under a variety of workloads and provide the results of our findings.

1. Introduction

In this project, we implemented a light version of the Google File System. With our implementation we sought to preserve the power and fault tolerant properties of GFS while providing a clear and simple interface for viewers to see the more nuanced technical details and obstacles that goes in to building a large distributed system. We sought to explore and tackle these unspoken technical problems such as "How do you actually implement leases and manage them between master and chunk server? and "How do you isolate and write structured code for the multitude of tasks that the master has?" By confronting these problems with modern software paradigms and coding patterns, we hoped to gain a better understanding about the concrete work it takes to building such a large system and get better practice at tackling the unsung technical obstacles. Along this journey, we worked to preserve the naive implementations (e.g. synchronous) we initially created as a means of exploring the performance differences that technical optimizations (e.g. async $>$ sync processes) provide in such a large system. Most notably, a key optimization that we frequently utilized was the conversion of repeated actions into separate overlapping asynchronous routines. For example, we saw that making the forwarding of write requests to secondary replicas asynchronous led our system to significant performance improvements.

This paper is organized as follows. In Section 2, we discuss the architecture of GFS Simplified Implementation through describing the main components of the system and their responsibilities. In Section 3, we walk through the testing process and functionality tests that were routinely performed during development to ensure correctness. Section 4 focuses on bench marking, stress tests, and data collected and compared across the modified configurations and architectures. In Section 5, we discuss considerations and challenges that we faced during implementation. Lastly, Section 6 summarizes production GFS features that we were not able to implement, given the time constraint, that would have created more opportunities to experiment with

that is related to our current areas of exploration.

2. Architecture

In our implementation, the system is broken down into three main modules that will be described in the following subsections: Master Server, Chunk Server, Client.

2.1. Master Server

The master server is the core service provider and main point of contact for GFS clients. It is the key entity which tracks the current state of the file system. More specifically, it contains an in-memory list of chunk servers in the system, a map of file names to file handles, a map of file handles to chunk server, a map of chunk handles to lease, and a set of chunk handles that are currently in-use.

The master is responsible for managing all the file chunks (64KB segments) on the system but does not directly store these chunks. Through the in-memory data structures it is comprised of, the master server designates chunk servers as location where a given chunk will be stored on. This is relevant in the case of creating a new file and in the case of writing to a file such that it grows to span across more chunks. When removing files, records of the chunk location and chunk handles will be removed. Essentially, clients that interacts with files in the file system will first communicate with the master to determine the locations of the chunks they want to modify and from there send operations requests to these locations. Throughout this entire process, the chunk server will be in constant communication with the master to ensure that all affected file metadata is properly updated.

2.1.1. Lease Manager Similar to the original GFS implementation, the master server is also in charge of managing and distributing leases to chunk servers. We encapsulate this functionality in an entity we call the Lease Manager. In particular, whenever a client wants to make a mutation to a specific chunk, it first inquires the master about what chunk server currently holds the lease for the desired chunk. If no one holds the lease, one is granted to one of the chunk servers that holds a replica of the chunk. This data is relayed to the client as part of the GetChunkLocationRpc. Since our implementation is built on a single computer, we were able to implement our Lease type by leveraging background daemons and the computer's internal system clock. For the sake of our simulations, we set the timeout for our lease implementations to 30 seconds.

2.2. Chunk Server

Chunk servers are where file chunks are stored. For the sake of simplicity in our implementation, every chunk server is allocated a linux based directory which serves as their store for all chunks. The number of chunk servers in a system is configurable and set in the master server. Accessing of files for any CRUD operations will require direct read or write to the memory of these chunk servers. In addition to these more trivial functions, chunk servers are also tasked more obscured functions in order to drive core Google File System semantics and maintain consistency.

2.3. Handling Mutations

When a *write* operation is requested by a client, the writing of data into physical memory must be replicated across n chunk servers where n is determined by the configured replication factor. To reduce bandwidth consumption, the client only send a write-request to a single chunk server, called the primary chunk server (Ghemawat (2003)). Among the set of replicas, a primary server is determined by the current holder of the chunk lease or if non-existent, a random replica server. When a primary chunk server receives a write request, it first checks if the lease it holds on the chunk is about to expire and if so, it will communicate with the master to renew the lease. Following that, the primary chunk server then forwards this write-request to secondary chunk servers to replicate the file chunk and only returns to the client when it receives an acknowledgement that all replicas have successfully committed

Name	Description
SendHeartBeatMessage	Called by chunk servers for periodic communication with master
GetChunkLocation	Called by clients to find replica locations of chunks
GetSystemChunkSize	Called in set up to determine system chunk size
CreateFile	Called by clients to create empty files
RemoveFile	Called by clients to remove files
RenewChunkLease	Called by chunk servers to renew a lease

Table 1. MasterServer RPC Interface

Name	Description
Read	Called by clients to read a chunk
ReceiveWriteData	Called by clients as part of data flow ingestion for mutation
PrimaryCommitMutate	Called by clients to request a write operation
SecondaryCommitMutate	Called by chunkservers to prompt secondary replicas to commit a mutation
CreateNewChunk	Called by master to create a replica of a chunk
RemoveChunk	Called by master to remove chunk
ReceiveLease	Called by master to pass a lease to a chunkserver

Table 2. Chunkserver RPC Interface

the mutation. In our system, any error on the secondary servers are met with a repeated call until completion. Any arbitrary chunk server can serve as a primary or secondary under different client transactions. The secondary chunk servers will communicate the status of a write operation the primary and the primary responds with a cumulative success status to the client if all chunk servers successfully wrote or fail status in the else case.

Furthermore, the primary chunk server is also responsible for determining a serial order of file mutations. This is extremely important when any given primary receives a flurry of concurrent write requests to a single chunk. In our system, the primary server determines this serial order through a first come first serve basis. Allowing the single primary chunk server to determine an order is critical to maintain synchronization and consistency across the distributed system. This serial order is included in the forwarded write-request to the secondary chunk servers so that all chunk severs storing replicas of a chunk with handle x follow this mutation order. This is critical to ensuring that all replicas observes the same effects from concurrent sets of mutations.

2.4. Handling Reads

On the other hand, the *read* operation observes less overhead since it does not compromise consistency. After retrieving a list of chunk location(s) from the master server, the client may send a read request directly to any one of the replicas. Our current method of load balancing is through random selection. In other words, to reduce hot spots and evenly distribute the load of read-requests sent to chunk servers, clients will send a request to a randomly chosen chunk server that contains the targeted chunk. The chunk server then responds with the data that the client requests to read, fulfilling the read-request.

2.5. Client

We implemented a Client object to represent a source of file system requests. The Client has access to exported master server and chunk server functions, which simulates public API endpoints of production GFS. Its implementation comprises of five main operations: a constructor, Read, Write, Create, and Remove. The latter 4 are intentionally

written to be similar to the commonly known Linux functions. This allows the client invocation of these functions to be familiar and recognizable. The Client object also serves as a medium to carry and execute specific workloads we created in order to benchmark the relation between operation performance and system configuration

3. Functionality Testing

Functionality tests were routinely added and ran to ensure correctness throughout the implementation process. These tests were meant to check for correct behavior and stress the system in an end-to-end fashion. Test cases created, wrote, read, and removed files of a variety of sizes from single-chunk to multi-chunk spanning files. We also created test cases where multiple clients access the same file simultaneously. The number of clients running in parallel were also varied to validate lease management and consistency over time.

4. Performance Testing

Performance testing was very similar to functionality testing. The key difference lies in the chosen workloads and the usage of benchmark-specific super classes of a clients and chunk servers during boot up. The super classes contains an additional configuration object that enables logging of operational latency and toggling of certain architectural details. This allowed us to measure and record the effects of these design decisions and observe how it reduces system bottlenecks.

During performance testing, the latency of CRUD operations were recorded. We define latency as the amount of time between the client making a request to when the client receives a success response from the system. We focused on tracking latency for Read and Write since they are directly impacted by client workload and system configuration.

There were two points of architectural details we experimented with. The first is in the replication of chunks across multiple chunk servers. In one case, we replicated the chunks sequentially, where the system is blocked from making a subsequent replications until the previous replication responds successfully. The other case conducts replications asynchronously. The second point of experimentation is when the primary chunk server forwards the commit request to the secondary chunk servers. Here, we also tested two cases: the first using sequential execution and the second asynchronously.

In Figure 1, we observe the effects of various size workloads on latency a client-side Read and Write operation where chunksize was configured to 64kB. Latency was measured to be the total time between the invocation of the Read or Write API and the time when the client receives a success response. For Write, this indicates replication is completed across the configured number of chunk servers and that the master server's metadata accurately reflects this. For Read, this success response contains the data the client requested to read. The Async descriptor indicates that the GFS configuration is using asynchronous replication of write data and asynchronous forwarding of write commits from primary to secondary chunk servers. On the contrary, the Seq description indicates a sequential replication and commit forwarding process. We observed lower latency in asynchronous write compared to its sequential counterpart. However, there is no performance difference between asynchronous and sequential read. This makes sense because the asynchronous optimization targets only the GFS workflow for the write operation.

Figure 2 displays the effects of the number of concurrent clients on Read latency with varying number of Chunk Servers. The systems are configured to have 3, 5, 10, and 50 chunk servers. Replication factor is set to 3, and chunk size is 64kB. Furthermore, the current implementation uses a Random Sampler for load balancing. When a Client request to Read, it is randomly assign to contact a one of the three Chunk Server containing the targeted chunk. The file being read is 1kB.

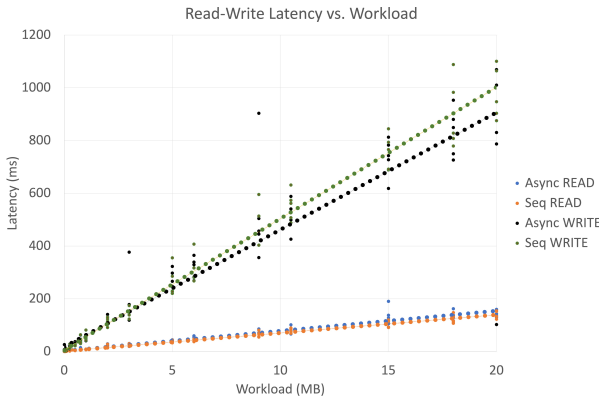

Figure 1. Effect of workload on Async/Seq Read and Write

Overall, we observed a linear correlation between the number concurrent clients and the temporal overhead of Read. This makes sense because chunkServers must handle Client read request in a linear queue manner. However, this is suboptimal performance because this indicates that the current implementation will not scale well. Fortunately, because of the scalability of GFS, we can somewhat address this performance issue by increasing the number of Chunk Servers in the system. We observe that through increasing the number of ChunkServers, more clients can be handled with a lesser increase in latency. Further optimizations are possible if we raise the replication factor to increase accessibility and use a more effective load-balancer that can actively mitigate hotspots and bottlenecks in the cluster.

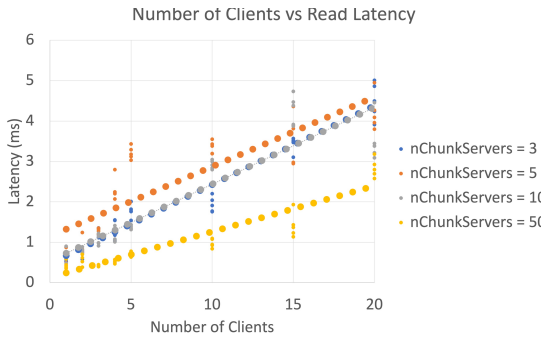

Figure 2. Effects of increasing clients on read latency

5. Challenges

In this section, we will discuss the variety of challenges we faced while implementing Simplified GFS: ranging from design problems to implementation bugs. While daunting at first, the free range we were provided in making our own design decisions on how to implement GFS ended up being an enjoyable experience which has helped us grow immensely.

5.1. Simulating GFS

The first immense design decision that we had to tackle dealt with how we wanted to mimic GFS. We of course did not have access to a multitude of commodity machines that we can separately set up to work with but we also didn't want to build a single heavily coupled binary with fixed behavior.

To this end, we juggled around three potential set ups, 1.) Implementing all the components (master, chunk server, client) in a single binary, 2.) Implementing all components as separate binaries and starting them up individually (e.g. different terminals) 3.) Implementing a master and chunk server binary and opening their ports for access by client binaries. We ultimately decided to follow option 3 as we felt it gave us the best middle ground in our options and had the fastest run way for us to start testing.

Despite the set up of a single binary for the master and chunk server creations, we still wanted to preserve complexity that comes with managing asynchronous communication between a multitude of nodes. To keep our project within the scope of our abilities, we decided to utilize a combination of goroutines and local listening ports to mimic the existence of individual nodes. To communicate between these nodes, we decided to use gRpc as we felt it best simulated reality and gave a sense of latency with network messaging.

5.2. Handling Concurrent Mutations

Another problem that we struggled with throughout this project dealt with the

implementation of concurrent mutations. Most notably, we struggled to develop fine grained synchronization patterns that could handle high traffic on the chunk servers and master while maintaining high performance. A key challenge for us was figuring out a way to elegantly protect a multitude of shared resources which were constantly being contended for. As a sanity test, we initially started off with extremely coarse locking with a single lock on the entire chunkserver before progressing to more fine grained locks per internal data structure.

6. Future Work

While we are satisfied with our progress, there are still a multitude of extensions and features which we hope to flush out in our Simplified GFS implementation. Most notably, we would want to implement functionality to support Record Append and Snapshot operations into our system so that we can support the full suite of functionality in the original GFS. By doing so, we hope to explore more nuanced design decisions and learn more about the plethora of problems we have to deal with in supporting more complex operations. Furthermore, there is still a lot of work to be done in flushing out our testing suite so that we can detect and fix edges cases which cause periodic failures in our system. A key failure that we periodically see in our implementation comes from when we bring our implementation to scale. Most notably, we see a huge decrease in performance as our locking is quite coarse and significantly reduces the performance of our system. Lastly, we hope to embed failure recovery mechanisms into our system such that it can sustain common failures at scale as supported by the original paper. We feel that this would be an extremely interesting path to explore and can also pave the way for our system to migrate to using containerization technology such as Docker which allows us to easily spin up and manage isolated components.

As a stretch goal, we hope to isolate some components of our system and leverage services such as Redis which would allow our emulation of GFS to scale for actual live use. This would involve building out a more robust and interactive client interface that supports the creation and modification of actual files. We feel that that this poses an extremely challenging but exciting opportunity for us to see what it takes to build out a distributed system from scratch.

7. Acknowledgements

We would like to thank Prof. David Mazieres and T.A Geoff Ramseyer for all their support and guidance throughout this quarter.

References

Ghemawat, S. (2003). *The google file system*. https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf (accessed: 04.22.2022)