# Google File System 2.0: A Modern Design and Implementation

Babatunde Micheal Okutubo
Stanford University
bmokutub@stanford.edu

Gan Tu
Stanford University
tugan@stanford.edu

Xi Cheng
Stanford University
cx1012@stanford.edu

## Abstract

*The Google File System (GFS) presented an elegant design and has shown tremendous capability to scale and to tolerate fault. However, there is no concrete implementation of GFS being open sourced, even after nearly 20 years of its initial development. In this project, we attempt to build a GFS-2.0 in C++ from scratch*[1] *by applying modern software design discipline. A fully functional GFS has been developed to support concurrent file creation, read and parallel writes. The system is capable of surviving chunk servers going down during both read and write operations. Various crucial implementation details have been discussed, and a micro-benchmark has been conducted and presented.*

## 1. Introduction

Google File System (GFS) [1] is a distributed file system known for its capability to handle large file storage at scale, while being fault-tolerant and preserving consistency to a certain level to support typical usages. The original paper on GFS [1] presented a novel and classic design, and had achieved significant impact in both academia and industry. However, because this work was developed at Google Inc., and was proprietary up to date, it is difficult to evaluate GFS from an academic point of view given that there is a lack of open source implementation that reflects its original design and has sufficient documentation and testing.

Very few literature, if not none, have been dedicated to expound many technical details of GFS that are left out in the original paper [1], or to explain implementation ambiguity, that are required to achieve the intended benefits such as crash-resistance and high concurrency. For instance, how to achieve high concurrency on both servers (master and chunk) and client sides? How to handle cluster reconfiguration caused by chunk servers going down? Only high-level sketches exist to answer these questions, without a concrete implementation that one can evaluate.

These questions motivate this project, which aims to build a GFS-2.0 from scratch, with modern software design principles and implementation patterns, in order to survey ways to not only effectively achieve various system requirements, but also with great support for high-concurrency workloads and software extensibility.

A second motivation of this project is that GFS has a single master, which served well back in 2003 when this work was published. However, as data volume, network capacity as well as hardware capability all have been drastically improved from that date, our hypothesis is that a multi-master version of GFS is advantageous as it provides better fault tolerance (especially from the master servers' point of view), availability and throughput. Although the multi-master version has not been implemented due to time constraints, we designed our system so that it can be easily extended to include multiple master servers.

Last but not least, there was no containerization technology back when GFS was first introduced. Our team is motivated to bring the usage of Docker as an intrinsic element of our system so that one can easily scale up servers and manage server restart upon crash.

In this project, we successfully built a fully-functional GFS (single master with a configurable number of chunk servers) that reflects the majority of its original design with a few simplifications such as not supporting concurrent append and snapshot. We applied Singleton and Composite software design patterns to build the core components of the system step by step, and we have leveraged on several state-of-the-art open-sourced libraries such as Abseil, Protobuf, gRPC, LevelDB, Parallel HashMap, and YAML to achieve high software quality and performance. Furthermore, we have conducted rigorous efforts to test our code as a significant portion (nearly 30%) of our codebase is composed of tests, including unit tests, mocked tests, integration tests and end-to-end tests.

This paper is organized as follows. Section 2 presents the architecture of our system with details regarding each component of the system and how to build them. Section 3 discusses the testing effort in our development. Section 4 shows the performance and benchmark results of our system. Section 5 discussed the opportunities for further de-

[1] https://github.com/Michael-Tu/cppGFS2.0

velopment based on this work and our learning.

## 2. Architecture

Figure 1 shows the architecture diagram of GFS-2.0. The File system is divided into three main components: Master server, Chunk servers and Client library.

### 2.1. Master Server

The master server is the central part of the file system. It handles file metadata and chunk servers control operations in the filesystem. The actual data isn't stored on the master, but on the chunk servers. It knows about all the files in the file system and all the chunks for a file, and where each file chunk is stored.

The master server is made up of four main components: Metadata Service, Lock Manager, Chunk Server Manager, and Chunk Servers Heartbeat Monitoring Task.

#### 2.1.1 Metadata Service

The Metadata Manager is the singleton object that manages concurrent CRUD (creation, read, update and deletion) operations upon metadata, which include mapping from filename to a series of chunk handles it owned (indexed by chunk index), and mapping from chunk handle to chunk version, server locations and primary lease holder.

The concurrent operations are achieved by parallel hash maps, which is composed by a number (default 16) of submaps underneath so that the read and write accesses to different submaps can be safely parallelized. An essential responsibility that the Metadata Manager carries is the management of namespace, i.e. to allow concurrent creation of files even if they share the same parent directory. This is facilitated by the Lock Manager, which manages the creation and fetch of locks for files, as their names are not known a priori. The details of how Lock Manager works is shown in the next sub-section.

The Metadata Service handles the client's request and operates on the metadata via the Metadata Manager. It provides the full CRUD support per client's call, and orchestrates a series of operations based on the protocol. For instance, a file open request with creation mode led the service to call Metadata Manager to create the file metadata and to call the chunk server manager to allocate the first chunk (see details about chunk server manager below). A write request involves even more interactions, as the Metadata Service needs to send requests to chunk servers for granting lease and for advancing chunk versions. The Metadata Service, along with other services are implemented as gRPC service calls in C++.

#### 2.1.2 Lock Manager

The Lock Manager is responsible for managing lock resources for files. This singleton object is necessary as the server does not create a lock until a file is created. It provides concurrent creation and fetch services via parallel hash map. Furthermore, it adopts the absl::Mutex object, which is provided by the Abseil library and supports both reader and writer lock modes. The use of parallel hashmap, reader/writer locks, Lock Manager and Metadata Manager provides the master server with full capability to provide concurrent file CRUD operations.

#### 2.1.3 Chunk Server Manager

This manages all the chunk servers in the cluster. It is aware of all the available chunk servers in the cluster. It is in charge of allocating chunk servers for storing a chunk and therefore knows where each chunk is stored. When a new chunk server is added to the cluster, during startup it reports itself to the chunk server manager running on the master and the chunk server manager registers it and the chunk server can now be selected for chunk allocation. Subsequently, the chunk server periodically reports itself to the manager. Information reported includes the available disk space, the list of chunk handles stored and the manager replies with the list of chunk handles that are now stale and can be deleted by the chunk server.

When a new chunk is being created, the metadata manager asks the chunk server manager to allocate some number of chunk servers for storing the chunk. The chunk server manager maintains a sorted list of chunk servers, ordered by their available disk, having chunk servers with the maximum available disk at the top of the list, so it selects the N chunk servers with the maximum available disk. This helps to achieve both load balancing and better disk usage across chunk servers. It keeps track of the N chunk servers that were selected for the chunk and the metadata manager will ask for the location of a chunk, during a read/write request.

Chunk servers can be unregistered from the manager. Which means the manager will no longer be aware of the chunk server and wouldn't use the chunk server for chunk allocation. This is done when a chunk server is unavailable/unresponsive and explained below in the heartbeat monitoring task.

Since the chunk servers periodically report themselves to the manager, and they report the stored chunk handles as part of it, the manager is able to update the information it has about this chunk server, such as available disk and stored chunks. A chunk server may have been previously allocated to store a chunk, but the chunk server doesn't have that chunk anymore, maybe it crashed during write or data was corrupted, through this report, that manager is now aware that the chunk is not on that chunk server and stops

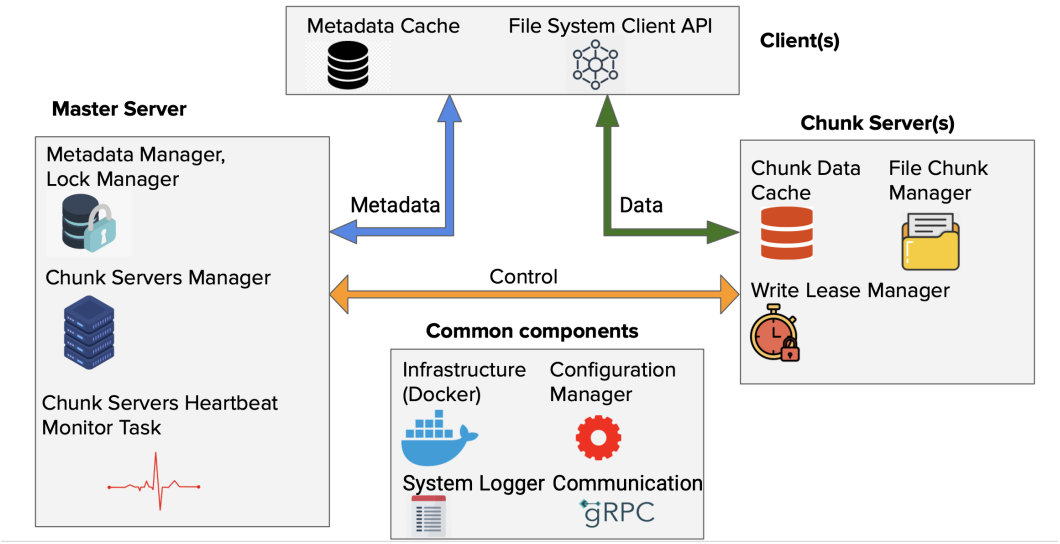


Figure 1. Our high-level architecture diagram.

including this chunk server as part of the location where the chunk is stored.

#### 2.1.4 Chunk Servers Heartbeat Monitoring Task

This is a background thread that monitors the heartbeat of all the registered chunk servers in the chunk server manager. It periodically sends heartbeat messages to the chunk servers and declares the server as available if it gets a response within 3 (configurable) attempts. If it doesn't get a response after those attempts, it declares the chunk server as unavailable and asks the chunk server manager to unregister the chunk server, so that the manager stops allocating that chunk server for chunks. If the chunk server becomes available or if it was just a network partition that prevented the heartbeat task from receiving a response from the server, because the chunk servers report themselves to the manager on startup and periodically, the chunk server manager will learn about this chunk server again and re-register it.

### 2.2. Chunk Servers

The chunk server handles data operations. A file is divided into file chunks of a configurable fixed size (default is 64MB) and each file chunk is stored on a subset of chunk servers depending on the replication need of the client. By default, chunks are replicated across 3 chunk servers. The chunks to be stored on a chunk server is determined by the master chunk server manager, described above. Chunk servers handle persisting this chunk data to disk. Clients only interact with chunk servers for actual data read or write.

The chunk servers are made up of 5 main components: File Chunk Manager, Chunk Server File Service, Chunk data cache, Chunk Server Lease Service, and Chunk Server Report Task.

#### 2.2.1 File Chunk Manager

The file chunk manager is in charge of persisting chunks to disk. We implemented this using LevelDB [2], which is a fast persistent key value store built by Google. The keys and values are treated as arbitrary bytes for storage. The file chunk manager uses levelDB to store chunks, using the chunk handle as the key and the chunk data as the value. This allows us to enjoy the benefits of this high performance storage engine in our system. We offload concurrency safety to LevelDB without explicit synchronization within the file chunk manager during data read and write.

We also make use of the compression feature of LevelDB, which helps to compress the chunk data and reduces disk consumption and makes our IO faster for a cheaper CPU cost. LevelDB compression is very fast and automatically disabled for incompressible data. As part of the data stored on the disk, we also store the chunk version, which is used during read/write. We don't need to cache the chunk data, since LevelDB writes the data as files in the filesystem, we therefore take advantage of the buffer cache.

The file chunk manager provides Create, Read, Write, Delete, GetVersion, UpdateVersion operation for Chunks. The Read and Write operations require the version to be specified and returns an error code if the requested version doesn't match the stored version of the chunk. For create, we store just the chunk handle as key and the create version (1) in the value with empty data.

#### 2.2.2 Chunk Server File Service

This is a gRPC service running on the chunk server that the clients sends chunk data requests to. This handles the Read, Send data, Write requests from clients, Create request from the master server during chunk creation, and the apply mutation request from the primary chunk server during write. This service interacts with the file chunk manager to process the clients request.

For chunk create requests, it asks the file chunk manager to create a new chunk with version equal to 1. The file chunk manager returns an error code if the chunk already exists, otherwise, it succeeds and the service responds successfully to the client.

For read requests, it asks the file chunk manager to read the specified length of data from the specified version of the chunk, from the specified offset. The file chunk manager returns error code if the chunk doesn't exist or the specified version doesn't match the stored version or the offset is greater than the chunk data length. Otherwise the service returns the read data to the client.

The write workflow is quite different. The client first sends the data to the chunk servers file service, and the service calculates the checksum of the sent data to make sure it matches the sent checksum, and then stores the data in the chunk data cache (described below) using the data checksum as the key. After this, the client then sends the write request to the primary server, which first checks if it has a valid lease on the chunk, if it does it then asks the file chunk manager to write the specified length of data to the specified version of the chunk, from the specified offset. The file chunk manager returns error code if the chunk doesn't exist or the specified version doesn't match the stored version or the offset is greater than the chunk data length. Otherwise the file chunk manager returns the number of bytes written.

The primary server sends this write request to all the other replicas in parallel to apply to their copy of the chunk. The replicas check the checksum of the write request, and reads the data for that checksum from the chunk data cache and does the same write using the file chunk manager. And returns the result to the primary chunk server. The primary sends its write result with the result of the other replicas to the client. For deletion, the master server returns the list of chunks to delete when the chunkserver does its periodic report to master, and the file chunk manager is asked to delete them.

#### 2.2.3 Chunk Data Cache

This is a cache on the chunk server for temporarily storing file chunk data sent by clients. As part of the write workflow, clients first push data to the chunk servers, before issuing the write request to the primary chunk server, which then instructs the other replica servers to apply the mutation using that data when it's done, as described above in the Chunk Server File Service. For our use case, the cache maps the data checksum to the data. And data is gotten from the cache by providing the checksum of the data as key and it is returned if it exists. Though we built the cache as a key value structure, which can be used to store anything. In order to avoid having a very large cache, we simplified our cache implementation by just removing the data once it is no longer needed. This is usually after the client write request has been successfully completed.

#### 2.2.4 Chunk Server Lease Service

This is a gRPC service running on the chunk servers, which the master uses to grant lease to the primary chunk server for a chunk. The master sends the lease information to the chunk server and the chunk server keeps track of the active, unexpired lease it has been granted. This is used during write, where the primary chunk server checks if it has a valid lease for the chunk before proceeding.

#### 2.2.5 Chunk Server Report Task

This is a background thread running on the chunk server that periodically reports the chunk server information to the master. The chunk server initially reports itself to the master on startup before then proceeding to periodically report. This keeps the master server updated about this chunk server. Information such as the available disk space, and stored chunks are reported to the master and the master replies with the list of stale chunks that the chunk server can delete.

This also helps in a situation where there is a temporary network partition between the master and the chunk server such that the master has been unable to reach the chunk server for heartbeat. In such a situation, after several attempts from the master, it assumes the chunk server is down and unregisters the chunk server, which prevents clients and the master from communicating with the chunk server. Because the chunk server periodically reports itself, even if the reports were unsuccessfully sent during the network partition, after the network partition is fixed, the report will be successfully submitted to the master, which then rediscovers this chunk server and re-registers it. Now the master starts using the chunk server again for chunk allocation.

### 2.3. Client Library

The client library provides thread-safe calls: open, read, write and remove. Every client call is translated via a thread-local ClientImpl object, which manages relevant resources such as the cached file chunk metadata, configuration manager as well as the gRPC end points that the client uses to contact the server. The cached metadata is stored in a

per-thread data structure named gfs::client::CacheManager, which provides mapping from (filename, chunk index) to chunk handle, and from chunk handle to chunk metadata (version, primary location) as well as chunk server locations. The CacheManager stores a timestamp when a chunk metadata entry is created, and would return an error upon access request if it expires.

The read implementation is relatively straightforward. The client iterates through all chunks and issues requests to chunk servers to fetch the content (it would contact the master for metadata if it found that it does have one or has an expired one). It concatenates the returned chunk bytes and returns the results to users. Because chunks are replicated, the client call returns successfully as long as one of the read requests is successful, i.e. the client's read can tolerate chunk servers down. Our implementation supports the handling of read upon EOF, i.e. if a requested read is beyond current EOF, we consider this as a valid case and return the bytes read upon EOF.

Our write implementation differs slightly from the GFS paper in that the chunk data is pushed directly from client to all chunk servers in parallel. This significantly simplifies the engineering efforts as the client only needs to handle errors between itself and the chunk servers. Furthermore, pushing data in parallel may incur less latency than the alternative way described in the original GFS paper, although the latter may have a better utilization of each machine's bandwidth. We implemented a retry policy in the write library call that is transparent to users to handle any transient failures such as lease expiration and gRPC call timeout. This improves the stability of this function call.

### 2.4. System Common Components

We use Docker to deploy the file system servers in a containerized manner.

Each component has its Configuration Manager, which allows us to dynamically change configurations of the different components of the system. The configuration manager uses a YAML file for storing and parsing configurations. The YAML file is preloaded with some initial configuration such as master server address/port, lease timeout, max chunk size etc.

We also have a System Logger, which wraps the Google glog library for logging. This is used within all the components of the file system for logging.

## 3. Testing

Testing is a crucial part of this project, as it provides verifications to each component's expected behavior as well as to the end-to-end behavior. Furthermore, it prevents regressions which can easily happen in a large code base development. In this project, we developed comprehensive test suites ( 4K lines of code out of 15K code base) to cover tests at all levels, including unit tests, mock tests, stress tests, integration tests and E2E tests.

Unit tests ensure the correctness of each individual component, and serves as the foundation for the engineering efforts in this work. The server-side components (Lock Manager, Metadata Manager, Chunk Server Manager etc.) have been heavily tested in multi-threaded environments using gtest framework. For example, the unit test for Metadata Manager tests a scenario where concurrent operations are taking place to create files that share the same parent path and to create file chunks underneath. All such unit tests have been stressed tests by repeatedly running for more than 1000 times before merging up.

The mocked and integration tests serve to test a slice of the end-to-end behavior in a single process. For example, a client's read call can be tested by spinning up a metadata service and a chunk server file service in separate threads with some mocking of the metadata and chunk data. This helps us assess the correctness of interactions between different components in a relatively inexpensive way.

We also developed end-to-end tests to ensure the correctness from a user stand point of view. These tests are intrinsically distributed, and we have developed our own python module to automatically generate configuration files and to launch and manage server processes. Some typical test cases include multiple clients concurrently creating files, writing to files in parallel (each client writes to separate files as we do not fully support concurrent write) and reading concurrently.

## 4. Performance and Benchmarks

For performance and benchmarks, we leveraged the open sourced Google Benchmarks library. We did a basic benchmarking of read, create, and create+write performance by testing GFS client APIs against a running GFS cluster of single master and three chunk servers, with three replicas per file.

Due to time constraint, we have not gotten the chance to run masters and chunk servers separately using different machines yet, so the benchmarks are done on the same machine, using inter-process communications via gRPC that's built in Docker. However, it is in our plan to add these stats. once available, to our github README. Thus, the benchmarks here are biased because the network communication delays will be noticeably faster than what it will be like in real-life. However, the numbers we gathered should still shine some lights on the initial performance of our GFS implementation.

For each benchmark we share below, we run each API in each scenario and each parameter configuration for about 500-1000 runs, and report the average latency amongst all the runs. We also explicitly make sure that each run uses a

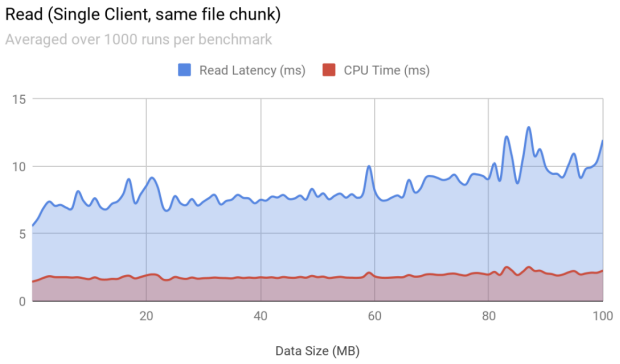


Figure 2. Single Client Read Performance over Data Size

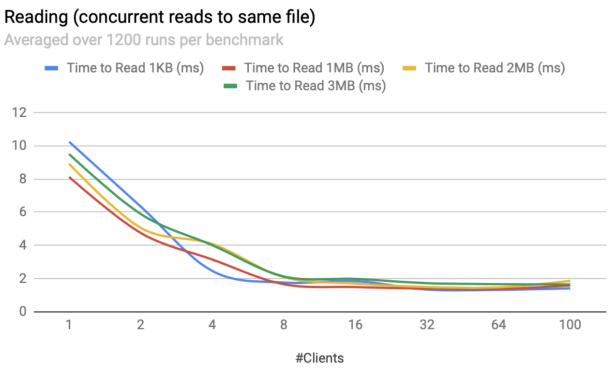


Figure 3. Concurrent Client Read Performance over Client Counts

freshly initialized GFS client, so it will perform the entire end-to-end GFS metadata control and the data control flow, and not reusing the cached metadata locally to bias the results.

For reads, we benchmark our reads using a single client operating, reading the same file chunk with read length ranging from 1KB to 100MB, while making sure each request does not reuse metadata cache but instead re-fetch it from master to get a clear picture of non-cached read performance. As we can see in the Figure 2, the CPU time used on the client is relatively stable, which is expected; but the network time increases as read size increases, causing longer network byte transfer delays. In our cluster, we used 64Mb per GFS block size, and we tested our read performance up to 100MB, and we can clearly tell a bump in read latency around 64MB. This is because in our existing implementations, we read the blocks one at a time sequentially. After we implement the reads for each block index in parallel, the latency should noticeably improve.

We also benchmarked how file reads latency change as the number of concurrent read clients increase. As shown in Figure 3, it turns out the more concurrent clients reading the same file, the better the latency, and the latency seems to stay the same if we are reading the same block. It may seem counterintuitive at first, but it is expected as we are benchmarking concurrent reads to the same file. Since we handle file reads concurrently as well on the server side, while sharing a single file manager, the data cache from one client's read request already paid the cost of fetching the data from disk, so other concurrent client reads requests can leverage the data cache in servers memory and get a better read latency. However, it's also a good idea in future for us to analyze where is the bottleneck of the read performance, and how concurrent random reads or sequential reads latency will be, because those access behaviors will change the caching behavior on the chunk servers.

We also benchmarked the file creation time, file write time to existing files, and file operations that need to create the file before write. Generally, it's expected that creation + write takes longer than a simple write, because of the file creation protocols such as Initialize File Chunks. In our implementation, create + Write operations cannot reuse the metadata returned by the create to write on the first index, because we always enforce to re-issue the open with write call to master, to coordinate the file version advance protocols, and granting lease services, due to data correctness concerns when a chunk server fails. However, it's possible that we incorporate an open with write+create mode to the master, so the file creation not only creates the file, but also initializes the first chunk, assigns a leaseholder proactively, so the writes to first file block doesn't require two metadata calls to master, but for simplicity, we didn't but can consider it in future.

As we can see from Figure 4, the creation latency is the gap between the create+write latency curve and the write latency curve, which is relatively stable regardless of the data size, which is expected as file creation time is not affected by the write operations that will come later. In future, we think it's worth adding additional analysis to see how file creation latency changes as we increase the number of replicas for the system.

## 5. Discussion

Several extensions can be made based on the current development. First, operation logs and checkpoints of the metadata states can be easily added. As we have utilized Protobuf to define the metadata states, and fully integrated them with the Metadata Manager, the operation logs can be implemented by serializing these states as bytes using the Protobuf and persisting them to states by leveldb [2], which is already used in the chunk server implementation. A simple implementation of the checkpoint service would be spinning out a background thread that periodically wakes up and takes a snapshot (by making the metadata read-only) and serializing the states to bytes and then to disk. The op-

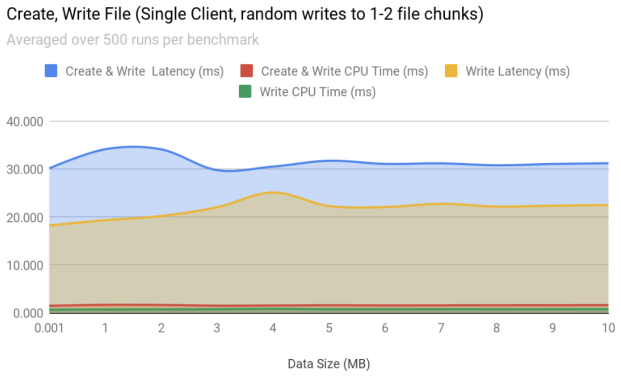


Figure 4. Single Client Create, and Create+Write vs Data Size

eration logs can be easily implemented by inserting leveldb write calls when metadata are mutated. A caveat is that some care needs to be given to the recovery of file locks (the locks under LockManager) during log reply, as we do not directly persist lock states.

A second extension is to develop a multi-master GFS based on the current architecture. Several steps are necessary in order to achieve this goal. First, we need to implement a consensus algorithm (e.g. Paxos, Raft) to synchronize states stored at the master servers. Second, we need a DNS service to resolve names so that the server address can be translated to the client in a transparent way (e.g. failover should happen transparently as client switches to a different master server). Last but not least, a load balancing service is needed for the master's DNS service. These developments require substantial engineering effort, but are valuable to explore in the future.

As we implement a fully functional GFS that supports concurrent reads and parallel writes, and is able to survive multiple failure scenarios, there are tremendous lessons learned along the way. First and foremost, applying discipline software design discipline and rigorous testing is a key to carry out an elaborate design such as GFS. While the paper may strike someone as straightforward, the devils are in the details. With substantial effort given to testing, we managed to find design and implementation issues in individual component levels and fixed them early on, which is key to the success of this project.

Second, it is important to choose the right tool and library so that one can avoid reinventing the wheel. We thought carefully about what libraries to use to build our system and evaluate the decision critically. For example, when applying the parallel hash map to manage lock and metadata, we realized that the default option is not sufficient, as we would like to have atomic operations on "check if this value exists, if so return the value otherwise return an error". The default function interface has to break this into two calls which are subject to time-of-check-time-of-use issues. This led to the investigation of building our own solution based on the default parallel hash map, where we develop our own interface such as TryInsert, TryGetValue with explicit lock management in these function implementations.

Third, adding system logging turns out to be an important decision as it provides critical information to debugging (and turns out to be a big plus for demo). In small projects, the role of logging is often downplayed, but in a large distributed system, logs are the key to understand the behavior of a system.

## 6. Acknowledgement

We would like to thank Prof. David Mazieres and our TA Jim Posen for their guidance and support of this project.

## References

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

[2] S. G. Jeff Dean. *LevelDB*, 2011 (accessed May 9, 2020).