Simplified GFS

Jason Lin
Stanford University
450 Jane Stanford Way
jason0@stanford.edu

Abstract

Simplified GFS is, as its name implies, a simplified version of the Google File System. It is written in Python and attempts to emulate the overall architecture and design of the original Google File System while also maintaining a level of simplicity for both client use and programmer development. It allows a client to create, read, write, and delete files within a distributed filesystem while also allowing for recovery from both master and chunkserver failures.

1. Introduction

The motivation behind building simplified GFS was for me to gain a deeper understanding of how to build distributed filesystems and how to ensure they are able to recover from component failures. I was interested in building GFS in particular because I had previously interned at Google on a team related to Colossus, which is the successor to GFS. The team I interned on was building a new client reader for Colossus that aimed to make client reads cost under 10 microseconds of CPU time. While I learned a lot about the Colossus client, I regretfully didn't get a chance to learn much about the actual Colossus system while I was there. This is why when I saw we were studying GFS, I jumped at the chance to actually build it and fully understand the entire system that I had been working with previously.

My language of choice was Python, because although my experience on Colossus had been in C++, I was far more comfortable working in Python overall. Additionally, since the aim of the project was more to learn about the overall system rather than to maximize performance, I believed that the use of Python would allow me to iterate faster on the code that I was to be writing at the cost of code performance, as it is a much more programmer friendly language than something like C++.

The emphasis on ease of use and/or development over performance can be found in several other places within Simplified GFS. For example, the standard GFS uses a chunk size of 64MB and 3 replicas as its standard values. While these numbers can be easily configured for simplified GFS, the standard numbers that I used were 64KB for chunk size and 2 replicas. This greatly helped with ease of testing and development, while still retaining all of the essential components needed for functionality and fault tolerance.

I will structure the rest of the paper similarly to how the original GFS paper was structured. This will allow me to easily compare and contrast Simplified GFS as I differ from the standard GFS operating procedure.

2. Design Overview

2.1. Assumptions

1. Like GFS, we assume that hardware can easily fail. We must be able to recover easily from such failures, although we do not assume that there is a monitoring system in place.

2. We do not expect to have as many files or as large of files as GFS does, but Simplified GFS should be able to handle hundreds of files each of around 1MB in size.

3. We **only** allow writes to append to the end of an existing file within Simplified GFS. This is a stark departure from GFS, which optimizes for append writes but still supports random writes within files. This decision was made to simplify the writing process, and I believe it is justified since Google themselves say that they expect a very small percentage of writes to be random writes.

4. Multiple clients may append to the same file simultaneously, but performance will be slow due to the design of the system. This will be elaborated on in the **challenges** section.

2.2. Interface

The interface mostly follows the interface of GFS. There is no explicit file hierarchy, and the file namespace is completely flat. However, files are still identified by pathnames,

and so a user can still build a hierarchical filesystem implicitly. We support creation, reading, writing, and deletion of files. For create, the user specifies the file name to be created. For read, the user specifies the filename to be read, the offset to start reading from, and the amount to read. For writes, the user specifies the filename to write to, the data to write, and the offset within the file to start writing from (which should correspond to the end of the file). For deletes, the user specifies the filename of the file to delete.

2.3. Architecture

The architecture matches that of GFS. We have a single master and multiple chunkservers. The master controls all metadata for files within the system. The client will contact the master whenever it wants to read from or write to files, and the master will provide it with the necessary metadata to do so. Files are divided into chunks of size 64KB, and chunkservers replicate those chunks for fault tolerance. One main difference, as mentioned before, is that the default number of replicas for simplified GFS is 2 instead of 3. Finally, the master and chunkservers all communicate with each other, both at an interval and on startup, in order to exchange state information.

2.4. Single Master

Again, clients must contact the master server whenever it wants to read from or write to files. For reads, the master will respond with a list of chunkservers that the desired chunk is on, which the client will then contact. The client will cache this information for a pre-determined time period, and this time period is reset every time the same data is read. For writes, the master will respond with the leaseholder (primary) for that chunk and also all replicas of that chunk. The client also caches this lease information and will only ask the master for a new lease whenever the primary responds that it is no longer the primary. All of these help us avoid overloading the single master, as it is definitely a potential bottleneck in our system.

2.5. Metadata

The master stores similar metadata in Simplified GFS as it does in GFS. It keeps track of a filename to chunk mapping as well as the chunk to replica list mapping. It stores the file to chunk mapping persistently, but does not store the chunk to replica list mapping persistently. Instead, it asks chunkservers for their respective chunks whenever they join the cluster.

The master also does a periodic background scan of its metadata, in which it does garbage collection, checks for missed heartbeats from chunkservers, and finally rereplicates data if it notices a chunkserver is down.

Whenever the master stores data persistently, it does so in its log, which is in essence a checkpoint for normal GFS. This design choice vastly simplifies the necessary master logging operations. We use Python's pickle library to directly convert arbitrary Python objects into a serialized format that can be written to and read from disk.

In Simplified GFS, we write the filename to chunk mapping, the list of all currently connected chunkservers, and the current chunk ID counter to disk. When the master restarts from a crash, we can then recover our internal in-memory data structures by reading these data structures from disk. We write the list of currently connected chunkservers persistently so we can poll them for their chunk information whenever the master comes back up. We write the current chunk ID counter to disk because the way the master assigns chunk IDs is using a monotonically increasing integer counter. If we remember what number this counter was at, then we can avoid assigning a duplicate chunk ID to different chunks.

2.6. Consistency Model

Simplified GFS also follows the relaxed consistency model that GFS follows. Explicitly, since 'writes' and 'record appends' in GFS are combined into a single operation in Simplified GFS, we guarantee in Simplified GFS that a non-parallel write will return the offset that it was specified to be written at. However, if another client is writing to the same chunk in parallel, then the returned offset may be different from the specified offset. For example, if we specify 'write('test.txt', 'test data', 0)', then we are guaranteed to return 0 for non-parallel writes. If there is another writer writing to test.txt, then the returned number may not be 0.

In both cases, regardless of whether or not there is another parallel writer or not, we are guaranteed that the data written will be written starting at the offset returned by write. So, in our above example, if the write returns 2, then we are guaranteed to have 'test data' start at offset 2 within the file 'test.txt'.

Simplified GFS also achieves consistent mutation ordering by having the primary choose a single ordering of parallel mutations. Furthermore, it uses chunk version numbers to detect stale replicas and chunk checksums to detect any corrupt data.

3. System Interactions

3.1. Leases and Mutation Order

The lease mechanism that Simplified GFS uses is similar to that of GFS. On write, a client will first request a primary for that chunk from the master. If the master does not have a primary for that chunk within its metadata, then it will randomly assign one of the replicas as the primary. The client then sends data to all of the replicas in any order. Once that's done, then the client tells the primary to apply mutations. The primary will choose an ordering of the mu-

tations that it has received, apply them locally, then tell the secondary replicas to apply their mutations in the primary's order.

Leases are given a set timeout by the master, and when the lease expires, the old primary will tell the client this, and the client will request another primary from the master. Currently, there is no lease extension mechanism implemented.

Whenever writes are large or cross chunk boundaries, like GFS, Simplified GFS also splits writes into separate operations per chunk.

3.2. Data Flow

We also attempted to imitate GFS's data flow ideology. This means that when we send data to replicas, this is done in a line directly through all chunkservers. The client first sends data to chunkserver 1, which then sends that data to chunkserver 2, and so on. On the other hand, when actually applying mutations, the client first tells the primary, which then tells each secondary. Theoretically, this allows for us to fully utilize the network bandwidth for each machine, but in practice it lead to many more problems, which we will describe in more detail in the **challenges** section.

4. Master Operation

4.1. Chunk Re-replication

Every time the master notices that a chunkserver is down, it removes that chunkserver from all replica lists it is in. During its regular background scan, the master checks each chunk to see how many replicas it has. If it is below the specified number of replicas, then it beings the re-replication process. It can also begin re-replication due to a chunkserver reporting that one of its chunks has been corrupted.

Re-replication is performed by the master, which tells a chunkserver (without the chunk to be replicated) to ask another chunkserver (with the chunk to be replicated) to send its data. The chunkserver with the chunk to be replicated then sends it data to the chunkserver without the chunk to be replicated.

4.2. Garbage Collection

Simplified GFS will also do lazy deletion whenever the client deletes a file. This means that the file isn't actually deleted, but just renamed to a deleted name, meaning it can still be read and recovered by renaming it back to the original name. After a certain time period has elapsed, then the master's background scan will notice that its deletion period has elapsed, then delete all metadata pertaining to that chunk from its memory.

During heartbeats exchanged with each chunkserver, the master will let each chunkserver know the chunk IDs of all chunks that the chunkserver has but the master does not, which then in turn allows the chunkservers to delete those chunks from memory and disk.

4.3. Stale Replica Detection

For each chunk, the master keeps track of a version number. This version number is updated every time the master assigns a new lease for that chunk. This version number is used when a replica rejoins the cluster to see if that replica missed any mutations to chunks while it was down. If so, then its version number will be stale, and it will begin the re-replication process stated above. The version number is also used during the re-replication process to ensure that chunkservers are only copying up-to-date chunks.

5. Fault Tolerance

This was one of my main learning goals for this project, as it was a completely new area of systems for me, so it was fascinating to build and learn about fault tolerance.

5.1. High Availability

5.1.1 Chunk Replication

Each chunk is replicated on 2 servers for availability, which can easily be configured to 3. As mentioned before, any time a chunkserver goes down, the master will notice in its background scan (or whenever the client notices that the chunkserver is down and notifies the master). Upon noticing, it will immediately re-replicate this chunk to another server, thus preserving availability for chunks.

When chunkservers restart, it reads its chunk files on disk in order to recreate its in memory metadata. If it missed any mutations, then it will notice this using its chunk version numbers, which were also written to disk. If it didn't miss any mutations, then that chunkserver is just appended to the list of replicas for a chunk.

5.1.2 Master Recovery

Currently, we do not replicate the master state using shadow masters, as is done in standard GFS. We assume that the master does not permanently fail, since we do not have any monitoring infrastructure, and we assume that we are able to easily recover from temporary master failures. It can recover by simply reading its needed data structures from the log that has been persistently written to disk.

5.2. Data Integrity

For every 64 bytes of a chunk, each chunkserver will create a checksum of those bytes. It then uses this checksum to verify all data in a given range whenever either a client or another chunkserver attempts to read that data. When a client writes data, we generate the checksum for each block written, and append the checksum onto existing checksums

whenever the beginning of a checksum block has already been written.

For simplicity, we use an ASCII-based checksum: each character in a string is turned into its ASCII digit format, then each digit is added up to form the checksum number. The upside of this approach is that it is extremely easy to calculate and does not require any external cryptography library to be imported. In fact, one of the reasons I settled on this strategy was because I had originally started by importing a hashing library in Python, but these hashes weren't able to be pickled by the pickle library when attempting to write the checksums to disk.

Unfortunately, there are several downsides to this simplified checksumming as well. First are the several security concerns. Since an attacker can easily create an incorrect message that adds up to the same amount as the checksum, this type of checksumming does not fare well against malicious attacks, as it can only reliably check for accidental errors. Along the same line of reasoning, if there is an accidental error that happens to end up with the same checksum amount, then this will also not be detected. Secondly, since we use a character's ASCII value for checksumming, this necessarily means that we can only handle ASCII characters being written.

6. Measurements

Here, I will showcase two benchmarks regarding read and write throughput for Simplified GFS. Again, since the focus of this was learning rather than performance, there do exist significant bottlenecks within the system.

In order to avoid network latency and focus solely on where the compute bottlenecks were in the system, the benchmarks were performed locally on separate processes within the same server. The server has a 2.5 GHz quad-core processor, 16GB of memory, and 512GB of SSD storage. Initially, the Simplified GFS cluster being tested on consisted of 1 master, 4 chunkservers, and 10 clients. In subsequent tests, the number of chunkservers was increased to 10 in order to examine the effect of number of servers on system throughput.

6.1. Reads

We were able to get the best performance from reads. For read tests, we used N=1 through 10 clients reading a randomly selected 4KB region from a 10MB file 250 times. This means that each client read a total of 1MB over the course of the benchmark. **Figure 1** shows our results on a Simplified GFS cluster containing 4 chunkservers.

Here, we can see that Simplified GFS peaks at a read rate of around 5MB/s. This seems to be a much earlier peak than that of standard GFS, as in their graph, one can see that even at 15 clients reading, the read rate trend is still going upward. This could be due to the fact that we have

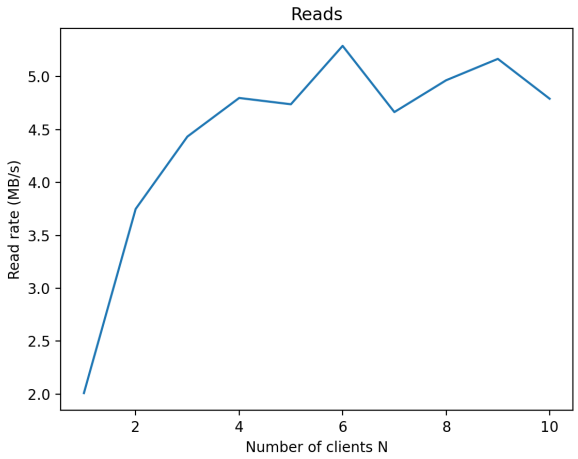

Figure 1. Reads on 4 chunkservers

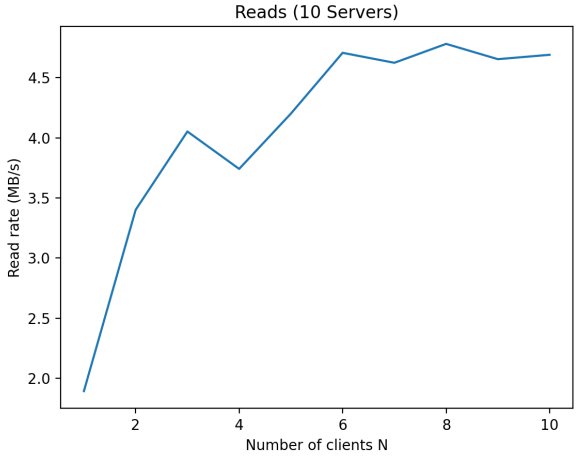

Figure 2. Reads on 10 chunkservers

an unbalanced number of chunkservers and clients in our testing scenario, as we see that the read rate increases all the way up until around 4 clients, which is exactly the number of chunkservers in the cluster. Thus, my next test was to test read throughput on a Simplified GFS cluster of 10 chunkservers and 10 clients. The results of this experiment are shown in **Figure 2**.

Unfortunately, even in the case of 10 chunkservers in the cluster, it looks like read performance is still capped at around 5 MB/s. This makes sense because reads are computationally very cheap to perform as the number of clients increases, as each client only has to talk to the master once per file that it is reading.

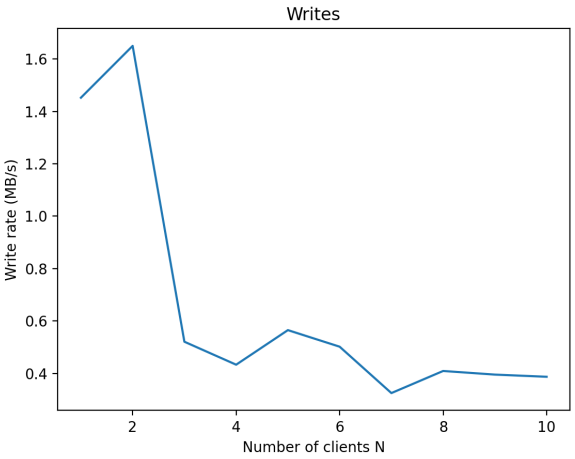

Figure 3. Writes on 4 chunkservers

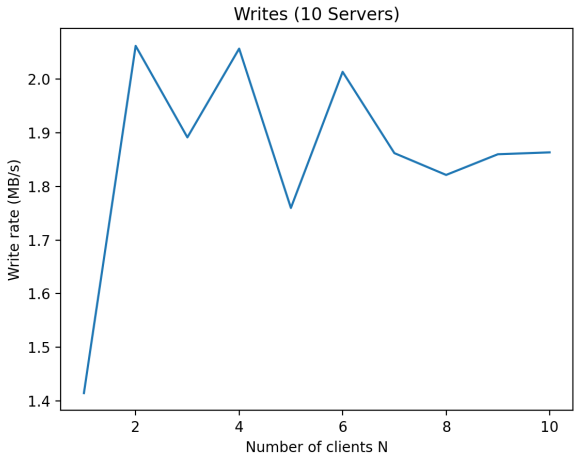

Figure 4. Writes on 10 chunkservers

6.2. Writes

We see definite signs of bottlenecking in our write performance. For write tests, we used N=1 through 10 clients each writing 1MB to a distinct file. **Figure 3** shows our results on a Simplified GFS cluster containing 4 chunkservers.

For writes, we see that Simplified GFS peaks at around 1.5MB/s write rate, and then quickly drops to around 0.5MB/s. This could be due to the fact that writes require much more computation and coordination amongst chunkservers, since chunkservers must send data to each other and then apply it in separate rounds, and given our lower number of chunkservers in our testing instance, this effect is then amplified. Thus, it made even more sense in this case to test write throughput on a cluster of 10 chunkservers as well. The results of this experimentation are shown in **Figure 4**.

From the results of this benchmark, we can see that increasing the number of chunkservers from 4 to 10 increases the write throughput from 0.5MB/s to 2MB/s, which is great to see! We also no longer see a significant drop in write throughput after hitting 3 clients writing. This result also makes sense, since as mentioned before, writes are much more computationally expensive for chunkservers to perform, so adding more chunkservers helps greatly in evenly distributing the computational load.

7. Challenges

While creating Simplified GFS, there were many significant challenges in design and development that I ran into. Although fixing bugs resulting from these challenges was extremely annoying, I found that I grew a lot in my understanding of distributed systems as a result of working through them. Here, I will discuss two of the main challenges I ran into and what I learned from them.

7.1. Balancing GFS Design with Python

One large error that I made was following the design of GFS too closely, given the fact that my choice of Python made so many things inherently different between GFS and Simplified GFS. A notable example of this is the Python XMLRPC library, which is a standard RPC library in Python that uses XML to encode data. I discovered that this library was synchronous, meaning that for a given chunkserver, only one RPC function could be in use at a time. Any subsequent calls to any RPC function on that chunkserver that occurred while a certain RPC function F was running was forced to block until the function F terminated.

As a result of this inherent synchrony, I decided it might be beneficial to the simplicity of my design to make my implementation lock-free and simply rely on the synchronous nature of the XMLRPC library in order to prevent data races. My error was that I should have changed the design of my implementation more to match this synchrony assumption, rather than continue to blindly follow the original GFS design. Many things would have been much easier to implement and also would have had much better performance, had I changed the design accordingly.

The best example of this is with the write functionality. In standard GFS, they chose to first have the client send the data being written to a chunkserver, which then forwards the the data through all replicas linearly. In a second phase, the primary then tells each replica to apply its mutations in an order specified by the primary. I chose to implement this functionality exactly. However, in Simplified GFS, the following scenario regarding parallel writers then fails.

Imagine we have chunkservers A, B, and C, and 2 clients writing to them. Client 1 sends data to all chunkservers first, and that data is stored in their respective buffer caches. Client 1 then tells the primary to apply mutations to its secondaries. Lets say that the primary is A, and the order in which it contacts the secondaries to apply their mutations is B then C. At this instance, if Client 2 begins to send data, say starting from B to A to C, then we will deadlock. This is because B will attempt to contact A while A is attempting to contact B, but both are waiting for their respective RPC functions to end.

I attempted to rectify this situation by forcing the sending of data and application of mutations to always follow the same order of chunkservers (ex. data must always be sent from A to B to C and the primary must apply mutations for B then C), but this also failed to prevent deadlock, because the primary could be any chunkserver along a given chain for sending data, and if it is in the middle of a chain, it will lead to the same scenario mentioned above.

I felt that the easiest, albeit very hacky, solution to this was to simply add an interrupt after a short duration of deadlock. Whenever no progress is being made and this duration has elapsed, the interrupt will fire and the sending of data will restart. This successfully prevents deadlock at the cost of extremely slow parallel writes due to constant interrupts. This was the simplest solution that came to mind that avoided having to re-architect the entire write system. If I were to do this again, I would design the write process with this in mind, and probably combine the sending of data and application of data into one function. This would be both significantly easier to implement and avoid the problem of deadlock, which would lead to significantly better parallel write performance.

7.2. Learning Fault Tolerance

Another overarching concept that I struggled with was fault tolerance. Although I found it fascinating to learn about, it was difficult to design systems with fault tolerance in mind, since I had so little experience working with it before. In all of my other computer science classes, I had always assummed that the underlying hardware would never fail. Thus, as I began developing Simplified GFS, I made the mistake of still holding on to that belief, and simply forgot to take into account that fault tolerance was a key learning goal.

This came back to bite me later as I began building out features for chunkserver replication and master recovery. In many cases, I had to add ugly try/catch methods and while loops in order to add bandages to places where chunkservers could fail, but I had not thought about earlier. This lead to several areas of ugly looking code and poor design. It also lead to many bugs during development, as there were errors that I had to slowly debug through, only to find that the errors were due to some assumption of component longevity I had made previously.

8. Future Work

There remains a great deal of work to be done for this to become a fully functional distributed filesystem. There are several notable features that I have not yet implemented. These include parallel writes that cross chunk boundaries (which can be implemented in a straightforward manner by doing what GFS does, which is padding the chunk to the end and doing the write on the next chunk), snapshot functionality for copying directories (since we currently don't have explicit directories, this should be straightforward as well, and can be done by copying metadata for a file and making it point to the same existing chunk), extension requests for leases (which should be piggybacked onto heartbeat messages from chunkservers), and explicit hierarchical directories. While the base functionality of Simplified GFS has been implemented, the addition of these features would be key in achieving full usability.

9. Conclusion

In summary, I believe this has been an excellent learning experience for me, as I've gained significant experience building a distributed, fault tolerant file system. I now feel better equipped to tackle distributed systems problems in my professional life, as I'm armed with a deeper understanding of how exactly a prominent distributed filesystem like GFS works under the hood.

Not only have I learned about the inner workings of GFS, but I've also learned that when designing my own systems, I need to think critically about what aspects of prior designs I should include and what I shouldn't, based on my knowledge of the current situation. It's not necessary to blindly copy every feature, and in fact in many cases can be detrimental to my own systems.

Finally, I now have a lot better of an idea of how to build systems with fault tolerance in mind. Knowing that components can and will fail required a significant shift in mindset for me, and now that I have experience tolerating and recovering from these failures, I'll be able to more easily apply these concepts to new situations.

10. Source Code

The source code for this project can be found at `https://github.com/jason2249/gfs`.

11. References

1. The Google File System: `https://static.googleusercontent.com/media/`

research.google.com/en//archive/
gfs-sosp2003.pdf