Technical Report on

The Google File System

Parallel & Distributed Computing

Prepared By:
Muhammad Daud Mazhar
18L-0919 - BCS 6C
May 12, 2021

Table of Contents

1. Introduction:

The Google File System, shortened for GFS, was introduced by Google in order to maintain and handle large amounts of data efficiently. The key problem that Google has attempted to tackle was the surge of excessively large amounts of data being stored on distributed servers, and quite interestingly, Google has adopted a rather unconventional approach of utilizing commodity hardware to store thousands of terabytes of data over thousands of disks incorporated in machines that they term as *'chunkservers'*. All of this data is concurrently accessed/manipulated by thousands of remote clients or users. It is the witty design of the GFS that has enabled the functionality of this distributed system successfully. We will discuss the choices made by engineers at Google, compare them with conventional file systems like the Network File System (NFS) and evaluate the trade-offs for Google.

2. Design/Architectural Choices:

The GFS offers similar abstractions to those of the NFS but has specialized in accordance to the needs as identified by user patterns inside the Google system. The primary assumptions on which the entire system is designed are based on the types of files Google clients use and the traditional sizes of those files. There are millions of files that are hundreds of MBs in sizes on average. Also, the users generally tend to access the files mostly for *read* and *append* purposes rather than write. This statistic has led the engineers at Google to tailor the GFS to accommodate these patterns of access in particular. However, it does not mean that there is no room for random writes; it is just that the file system is not optimized for such operations.

2.1 64 MB Chunk Size:

The statistical requirements collected by Google are quite different from those of the NFS, where random reads/writes are very common. Further, the NFS incorporates functionalities that are rather tailored towards dealing with large amounts of files but with smaller sizes as compared to those assumed by the GFS. Perhaps the most intriguing design choice the GFS has made is fixed-sized 64 MB chunk sizes. This is significantly higher than those used by the NFS or other similar file systems.

A repercussion of this strategy is that the *GFS Master Node* needs to maintain a thousand times lesser metadata, if let's say, the chunk size was instead fixed to 64 KBs. Due to the lesser metadata that needs to be stored, it is possible for the master to maintain all of this in the main memory. Also, the number of RPC requests from clients to the master are significantly

decreased as usually, most reads/writes by the clients can be done on the same chunk requested initially. This further reduces the network overhead incurred as the same TCP connection can be maintained for data exchange over that particular chunk.

The trade-off here is that accessing random memory (especially small files corresponding to single chunks) from this relatively large amount of metadata causes high latencies in control operations, which are, in fact, noticeable for the clients. If these files are requested by many clients, the few chunkservers storing these chunks might become *hot spots* and get overwhelmed. Moreover, at a later point in time, the memory might not remain sufficient enough. But this is not a very big issue as the cost of integrating larger memories does not overweigh the benefits yielded by storing the metadata in the main memory - reliability, performance, simplicity and flexibility, to name a few.

2.2 Single, Centralized Master for the Entire Cluster:

Another influential decision Google has made is the centralization of the master node, for the entire cluster, which is typically assisted by hundreds of chunkservers. These two types of servers together enable the GFS to serve a large number of clients accessing the data. The master is responsible for maintaining all the file system metadata, storing and locating chunks information and access control information. The chunkservers, on the other hand, store file data, and are responsible for creating replicas when needed.

The primary advantage of centralizing the master node lies in its ability to maintain an overall view of the entire cluster of chunkservers, which enables it to work out the most optimal chunkserver locations for chunk placement, efficiently. The implementation of a single master server is also simple and time efficient.

This kind of an approach does have a downside to it. This system is prone to *single point failure*, i.e. the failure of the master. If such a situation occurs, the entire cluster might halt, but the recovery mechanism devised to confront this problem will be discussed later on.

2.3 Separation of Control & Data Flows:

The GFS does implement a centralized master control system, but it is designed in such a way that the data and control operation flows can be distinguished and one process does not bottleneck the other. This results in high-performance file access with minimal involvement of the master. The master is only contacted to retrieve vital information to entertain the client RPC request, i.e. appropriate chunk identifier and location of the replicas (configurable, but 3 replicas by default). Once this information is provided, the client coordinates directly with the chunkservers for data exchange, thus freeing the master to attend other incoming client requests.

The advantage of this technique is that the individual bandwidth of each chunkserver containing the requested chunks and their replicas, is fully utilized, as each *primary chunkserver* transmits data to *secondary chunkservers* for replicating files and gathering their responses, without master node's supervision. This decoupling of control instructions and data flow allows optimal file access performance, engaging multiple clients concurrently.

Despite the advantages of this separation, the problem lies for the client libraries, which get complicated as they must deal with interacting both with the master server and the chunkservers.

2.4 Relaxed Consistency Model:

Google makes use of a rather relaxed consistency model to enhance the system's flexibility. The GFS applies a passive replication mechanism with slight modifications. The GFS maintains 3 replicas by default, which varies from conventional NFS and AFS models which do not provide replication with update. In the GFS, the data and control flow are decoupled as discussed in Section 2.3 above, and helps in creating a high performance system. The data inside any chunk can be altered by either *write* or *record append* operations: known as mutations in the data. For write operations, the offset is predetermined by the client application, whereas the GFS decides the offset in case of record appends.

For record append, this system guarantees an append at least once, and atomically with sequential data writing. However, it does not guarantee that all the replica chunks will have identical data since the GFS may add padding or record duplicates in between.

Similarly, another issue is that concurrent write operations to the same location are not serializable and may result in corrupting regions of the file data, making the file *inconsistent*. Therefore, this approach is a compromise between achieving simplicity for 'append only' mutations and maintaining strictly identical chunk replicas, where Google chose to opt for the former.

In terms of its significance to the API, we can safely argue that this system is *domain-specific*. It weakens the consistency guarantees for general applications, and works reliably with Google's applications and services in particular since their applications are customized to tolerate the resultant semantics of these mutations. Other applications can work equally well with this system if they can design their application functionalities to adopt checkpointing, self-validating and to rely on *append only* modes instead of *overwriting* data (which Google applications are capable of doing).

3. Scalability:

The GFS has been a successful player in the distributed system world because of its capabilities to achieve high performance, reliability, scalability and availability. It has managed to meet Google's storage requirements, which is spread across hundreds or thousands of commodity hardware chunkservers. In this section, we will look at the scalability aspects considering the single master server being used for the entire GFS cluster, discuss the caching mechanisms used and evaluate the advantages of *record append* operation.

3.1 Scalability Achieved by Single Master Server:

This part is partially discussed in section 2.2. We will continue our discussion by understanding how the master delegates the client RPC requests to the chunkservers for further processing, and escapes the picture itself. The primary responsibility of the master is to maintain system metadata which includes file namespaces, mapping files onto their associated chunks, locating these chunkervers, and handling chunk space management.

Communication between the chunkservers and the master is maintained through periodic *HeartBeat* and *Handshake* messages, which enables the master to keep track of live chunkservers and update metadata for the chunks and their replicas stored locally on those chunkservers. For any incoming client requests, the updated metadata is returned to the client, which then establishes connection with the designated primary chunkserver for data retrieval and manipulation, and the master exits the loop.

As 3 replicas (by default) are stored on different chunkservers, in different racks, the master is responsible only for maintaining the locations of those replicas, while the replication process is entirely assigned to the primary chunkserver chosen by the master. The clients share the data directly with all the chunkservers as specified by the master, but the master itself does not participate in the process. Therefore, the resource-intensive and time consuming task of creating replicas, and performing write or append operations is entirely under the management of primary chunkservers, which then communicates the order of mutations to secondary chunkservers to achieve consistency among all replicas. Since the chunk size is huge, i.e. 64 MB, the metadata that needs to be stored in the main memory does not generally exceed approximately 100 MB in size even if there are thousands of chunkservers inside the cluster.

This division in responsibilities and the ability to maintain the metadata for the entire cluster in the memory allows the master to exclusively handle hundreds of clients without significant delays in operation, thus enhancing scalability and performance.

3.2 Caching Mechanisms in the GFS:

The caching mechanism used in the GFS differs a lot from the generally used file systems like NFS. No caching is implemented below the file system interface. This is particularly because the majority of Google's target workloads stream through large data sets or read small parts from it randomly. Therefore, there is very little reuse of data within a single application run, consequently, eliminating the need to cache file data onto the clients. The only data that is cached onto the client side consists of chunk identifiers and locations of replicas, so that the client does not bother the master in case the connection to the chunkservers is lost.

This varies from the typical approach adopted by the NFS which implements client-side caching. The file data as well as the metadata is cached onto the clients in order to reduce subsequent access times for the clients (mostly because chunk sizes are smaller and clients need more chunks to access required files). The cache at the client side is also used as a temporary buffer to maintain all the *write* data, which is then translated onto the servers, all at once, when the files are finalized by the clients.

The GFS does not include this scheme since the file mutations are not handled by the master node and each mutation can be processed through chunkservers. Also, the clients usually do not require more chunks from the master after the initial request as the chunk size is rather big, and occasionally contains all the file portions needed by the client for reading/writing the data. As identified by Google initially, most accesses are sequential, e.g. reading through webpages. Such operations do not incur the need to incorporate caches in the system; consequently, avoiding inclusion of any unnecessary cache coherence protocols. The master only relies on the Linux based, default cache system for maintaining frequently accessed metadata inside its memory.

3.3 Benefits of *Record Append* Operation:

As previously discussed, the GFS is optimized for the record append operation when it comes to mutation. This is mainly because the client only needs to provide the data for record append, and the system determines the offset itself. Concurrent *write* operations to the same location might cause fragmentation problems from multiple clients. Clients would need additional synchronization methods if they wish to use the traditional *write* operations to mutate data atomically, like the *record append* in GFS. This would incur an extremely high overhead of using a complicated *lock manager* system.

In addition to this, the checksum computations are also well optimized for *record append* operations in contrast to *write* functions. The checksum is simply updated (incremented) for the last partial checksum block used. In case new blocks are filled by the append method, new checksums are computed. On the other hand, for regular *write* methods, the tedious tasks of verifying the first and last blocks of the range being overwritten, performing the write operation

thereafter, and finally computing and recording new checksums bears a lot of overhead. To keep things simple, Google has gone with optimizing its system with the record append feature and kept the consistency-related issues for APIs to cater for.

4. Fault Tolerance Against Disk or Other Component Failures:

This part of the report focuses on the fault tolerance strategies implemented in the GFS. The three fundamental players in the GFS, namely, client, chunkserver and master, play a key role in providing a platform that is capable of tolerating a number of system disruptions, resulting due to various factors including application and OS bugs, human errors, and failures in power supply, network devices or disks and memory etc. Since the entire GFS infrastructure does not rely on high end machinery, and instead uses cheap hardware modules, it is inevitable for the company to develop and deploy a system that accounts for these shortcomings. The GFS includes a mechanism whereby *fast recovery* is possible in case the system crashes. Both the master and chunkservers possess the capability of restoring their states rapidly within a matter of seconds and resume normal operation. Under this section, we will discuss the recovery mechanisms designed for server and component failures separately, and address the problem of stale data access by the client.

4.1 Master Server Failure:

The GFS uses logging in order to store each activity that occurs within the GFS infrastructure. The key metadata is persistently stored in operation logs, wherein the data is written asynchronously and sequentially. The master node, like chunks, is replicated onto multiple machines to provide reliability. The operation log and checkpoints are maintained in the same manner as maintaining chunk replicas by chunservers. If a master server crashes, the monitoring system outside the GFS environment delegates the master node responsibilities to a new master process (using the operation log entries) initiated elsewhere.

In any event of system failure, these log files are read and replayed by the master to restore its state. It is important to recognize that a separate checkpointing mechanism exists to update the log files. Activities are recorded in a compact B-tree like form to keep the operation log small, through which log activities can be directly mapped into memory and be looked up using namespaces without conceding extra parsing overhead. By using appropriate thresholds, the older logs are deleted, as to maintain only the recent operations that would be vital in restoring the master’s pre-system crash state.

‘Shadow masters’ are systems that work as an assistant to master servers. They are not as capable and authorized as the master servers but can enhance performance of the cluster by

dealing with *read-only* requests even if the master is down. Therefore, even in the short period of merely seconds while the master is restoring its state, shadow masters take on the responsibility for managing read requests.

4.2 Chunkserver Failure:

A chunkserver failure means that the actual file contents or chunks of data have become inaccessible to the clients. This is why multiple replicas of the same chunks are kept on different chunkservers, and those too, on different racks, to avoid data loss in case of entire rack failure. When a chunkserver fails, some chunks will get under-replicated. The master node will detect this (through persistent, periodic HeartBeat messages) and initiate steps to recover the chunkserver data. Each lost chunk that needs to be replicated is given a priority by the master based on several factors which include:

1. Preference is given to chunks that have lost more replicas than others, to ensure that more copies are created before the single replica available for a chunk is also lost (in which case, the entire file data associated with that chunk is lost).

2. Priority is given to files that are active or live as compared to dormant files (or recently deleted files) so that clients interacting with those files do not encounter any problems.

3. Lastly, any chunk that hinders the progress of client operations, is given boost priority in order to complete client tasks at the earliest.

4. The distance of the chunk in subject from its goal chunkserver is also evaluated.

Considering the priority rules above, the master picks up a chunk, and delegates an available chunkserver to clone the data from an existing, valid chunk replica location. The master keeps into consideration the goals of managing disk space across chunkservers (called load balancing), restricting cloning overhead on any single chunkserver and spreading chunks across multiple racks.

4.3 Client Request Failure:

The clients cache the chunk identification and replica location details when they send an initial RPC request to the master. The clients then pass on the data to each chunkserver. Each chunkserver also maintains this file data in an internal LRU buffer temporarily, so data need not be transferred over the network again in case the client proceeds activity with the chunkserver. In cases where the replication process encounters errors in any of the chunkservers, the primary informs the client about the errors. The client retries the cloning steps beginning from data sharing to all chunkservers, through to receiving validation from the primary chunkserver that cloning has been successful over all machines.

It is of great significance to remember that the client maintains *persistent TCP connection* with the chunkservers even if the cloning fails so that the master is not disturbed again, and reattempts could be made over the same connection. In case multiple reattempts also fail, then a *write* process is initiated all over again from the beginning.

4.4 Hard Disk Failures:

Hard disk failures is a very common problem faced by commodity hardware in particular. The data stored inside these disks may get corrupted, and the system would want to counter it before any client tries to access this corrupt data. For this purpose, the GFS incorporates checksumming mechanism to validate corruption of data before passing on replica information to clients when requested. Each chunk can be divided into blocks of size 64 KB, and each block has its associated 32 bit checksum. Checksums are stored persistently with logging apart from user data.

Since, as previously discussed in Section 2.4, the GFS does not ensure identical replicas across all chunkservers, each chunkserver needs to perform checksumming individually to detect data corruption and report it to the master via regular handshakes. The details of checksum methods for *write* and *record append* methods are briefly mentioned under Section 3.3. During idle times, the master also keeps running background checks on chunks that are not very frequently accessed. This is to maintain data integrity in case of hard disk malfunctioning for rarely accessed datafiles. When corrupt files are detected, the master replicates valid copies to replace them.

4.5 Detecting & Restricting Access of Stale Data:

Stale data refers to any chunk that has missed mutations due to failures in cloning or if that particular chunkserver was down. It is a vital problem to address so that only up-to-date copies of chunks are provided to all accessing clients. To achieve this, the master assigns chunk version numbers to each chunk. During the process of mutation, the master grants a new lease on the chunk and informs all replica chunkservers about the updated chunk version number. This updated chunk version number is maintained by the master as well as all the chunkservers, so that during any process of file access by the client, these numbers can be verified and only the up-to-date version of chunks are provided to the client.

If a replica is temporarily unavailable, or if the chunkserver contains outdated chunk version number, its location will not be advanced to the client for access. Once that chunkserver is back, the master will identify the outdated chunk versions included in the metadata sent to it via HeartBeat messages, and then manages to remove all stale replicas in its routine garbage collection process.

But there is a slight room for error here. Since the clients cache the chunkserver locations, they might be susceptible to stale data before the master has sent instructions to refresh them. This problem is addressed by defining a window which is restricted by cache's timeout expiry. Also, during the event of the next file opening, the cache purges all the chunk information for that file. Even within this short window, the clients may get stale data in terms of file metadata: directories or access control list, but not the stale file content itself.

[Note: the consistency model is already discussed in detail in section 2.4 under Design Choices]

5. Colossus - A Successor to the GFS:

Colossus is the successor to the Google File System which provides a storage infrastructure to enable enormous scalability for handling data for Google clients. The GFS originally was designed to meet specific requirements as inferred by Google engineers who claimed that high bandwidth is a priority over latency. But over the decade or so, the requirements have changed, and a need arose to develop a more highly scalable file system.

5.1 The Need for a New System:

1. The GFS Was tailored marginally towards batch-oriented applications like web-crawling, indexing etc, but applications like YouTube which require low latency are not very well suited for the GFS design. The problem that lies here is that there was only one master, and latency-sensitive applications require faster control and data flows, which was beyond the scope of a single master.

2. The time required to restore the master server in case of failure (in spite being in seconds) was too high for low latency applications. The design of the GFS was good enough to meet the needs of batch-oriented apps, but not streaming applications.

5.2 The Improvements in the Colossus:

1. The Colossus uses multiple masters instead of one. Google ended up designing a multi-cell approach which helped them build several masters (even hundreds) on top of an underlying chunkserver pool.

2. The GFS made use of *MapReduce*, (a platform used by Google for data-crunching) while the Colossus performs in coordination with the *BigTable*, which is a nearly real-time distributed database infrastructure.

3. The metadata was locally stored on the disks in the GFS. On the contrary, the Colossus has introduced a next-gen, highly scalable distributed metadata model for enhanced availability. This metadata is stored on the BigTable, which resulted in about 100x scalability over the largest clusters in GFS.

4. Background store managers called *Custodians* are used to track disk management tasks etc. which were previously all designated to a single master node.

5. The Colossus has a range of hardware devices, including spinning disks and flash storage devices. Each application gets allocated to different tier services in terms of using Google's hardware resources as specified by the application's latency, durability, and availability etc. requirements.

6. The Colossus incorporates an intelligent system to maximize storage performance, unlike the GFS. It uses a mix of flash and storage devices. The *hottest* (most frequently/recently accessed) data in flashes to derive low latency, while *colder* (aged/obsolete) data is gradually pushed to disks.

6. Conclusion:

Google's GFS and Colossus are both extremely large-scale distributed file systems. The GFS worked extremely well under its specific considerations of optimizing performance for batch-oriented applications requiring sequential data access, and append operations. This problem was addressed by the introduction of the Colossus which provided a massively scalable storage platform by eliminating the problems experienced by the GFS. Since Colossus is still relatively new, its limits to cope up with managing extremely high rates of data influx, keeping bandwidths and latency to optimize user experience, are yet to be tested.