A Survey of Google File System
CS 347 Mini Project Report

Karan Taneja, 15D070022
Sachin Goyal, 150020069
Sarthak Nijhawan, 15070037

November 8, 2017

Abstract

Google needs a whole together different file system to serve its data intensive application which have to tackle TB's of data consisting of billions of KB sized files. We present a concise overview of Google File System (GFS) and try to understand nature of its requirements and how it is designed differently than standard file systems in use. We consider the main assumptions, additional features, architecture specific details and additional operations added because of needs particular to a distributed-system as large as Google. Our goal is also to highlight how file systems can be developed according to needs of a system for a performance boost.

1 Introduction

Apart from other goals of distributed file systems such as performance, scalability, reliability, and availability, GFS is targeted for large distributed data-intensive applications, fault tolerance while running on inexpensive hardware, high performance to a large number of clients. GFS had to be robust in terms of file access time, recovery from failures and concurrently writing content.

"The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients."

2 Needs of GFS

The following points highlight the main needs that GFS has to meet:

- **Component failures are the norm rather than the exception.** GFS is running in on thousands of storage machines built from inexpensive commodity parts and is accessed by a huge number of clients. It is guaranteed that some components are not working at any given time. Problems can be caused by application bugs, OS bugs, human errors, and the failures of disks, memory, connectors, networking, and power supplies. GFS needs monitoring, error detection, fault tolerance, and automatic recovery systems to keep functioning properly.

- **Files are huge.** There are multi-GB files, growing data of many TBs made of billions of objects, billions of approximately KB-sized files that need to be handled by GFS. Keeping track of such a large collection of data, it retrieval and storage is done by GFS.

- **Appending new data is more common.** Overwriting existing data is less common for GFS and therefore, it is optimized for read-only data and sequential accesses. Because of the patterns of usage, appending is the focus of performance optimization of GFS. Atomicity is main focus because is optimizes sequential access for large files .

- **Flexibility in application interface and design is needed.** There is no burden on the applications in relation to storage and retrieval as processing time is of utmost importance There is an atomic append operation for multiple clients to append concurrently to a file.

3 Assumptions in Design of GFS

The following assumptions were made while designing GFS:

- Components fail very often and GFS has to monitor itself and detect, tolerate, and recover from component failures.

- The system stores a modest number of large files. We expect a few million files, each typically 100 MB or larger in size. Multi-GB files are the common case and should be managed efficiently. Small files must be supported, but we need not optimize for them.

- Optimization is largely done for large streaming reads reading hundreds of KBs, 1 MB or more and successive operations from a client reading a contiguous region. Small random reads, on the other hand, must be optimized by application themselves, for example by batching and sorting their reads to avoid going back and forth in a file.

- Sequential writes, appending data to files have to be highly optimized. Writes at arbitrary positions are generally small and do not have to be efficient.

- Semantics for multiple clients concurrently appending to the same file have to defined. Files in GFS are used as producer-consumer queues. Minimal synchronization overhead is essential and consumer might be reading through the file simultaneously as writers are writing.

- Very few hard response time requirements are there but because of the high processing performance promises, high bandwidth becomes important. Latency is of little issue in most applications.

4 Interface of GFS

Just like most distributed file systems, files are organized hierarchically in directories and identified by pathnames. Also operations to create, delete, open, close, read, and write files exist.Other than these standard features, GFS has snapshot and record-append operations. Snapshot creates a copy of a file/directory tree in an optimized way.

GFS has a large number of files shared by multiple clients. The various applications of Google involve feature of multiple clients working on the same file. Here if we use a locking mechanism similar to the critical section logic we studied, the speed will decrease drastically. So the GFS uses a modern way called Record-append which allows multiple clients to append data to the same file. This operation uses producer-consumer queues to help many clients simultaneously append to without additional locking. These operation are discussed later in detail..

5 Architecture

5.1 Main Componenets of GFS

The GFS architecture on a very basic level consists of 3 parts -

1. **Chunkservers**: There are multiple chunkservers which store the file locally. Files are usually divided into fixed size ‘chunks’. Chunks have a unique ID assigned to them by the master at the time of their creation.

The chunkservers basically store these chunks on local machines. For reliability, each of the chunk is stored on multiple chunkservers. You can access the chunks/files only through the chunk handlers via chunkserver.

2. **Master**: The master basically keeps a track of all the chunkservers and the mapping between files and the chunks.All this data is termed 'metadata'. The master regularly communicates with all the servers to give instructions to them and keep a regular update of their state.

3. **Clients**: Multiple Clients which are the local linux machines.

We will have a detailed look on each of the components further.

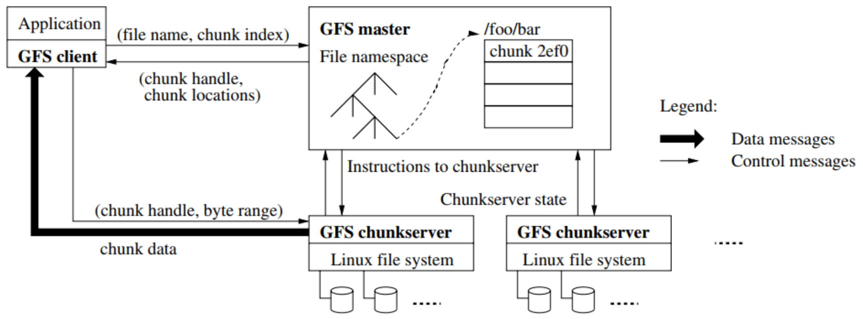


Figure 1: Main Components of GFS

Though we have specified here that clients talk to chunks via master, the case does not remain the same always. Generally clients repeatedly need to access a particular file and thus they make a cache of the location of the chunk where that is stored. The master comes in the way only for the first time a file is accessed or later when the cache location of that file has been removed. This methodology thus prevents the master from becoming a bottle neck for the speed of GFS.

5.2 Chunk Size

As we mentioned earlier, files are usually divides into fixed size packets called 'chunks'. So the natural question which arises is what should be the chunk size. A large chunk size offers advantage of reduction in overload due to accessing multiple different chunks to read a data. Also large chunk size will help in reduction of the metadata stored on the master.

This said, large chunk size has a drawback of causing some chunkservers to become hotspots. When miltiple clients try to access a same big chunk which stoers large number of small files, it becomes a hotspot. So an optimal combiantion of both factors is taken while deciding upon the same.

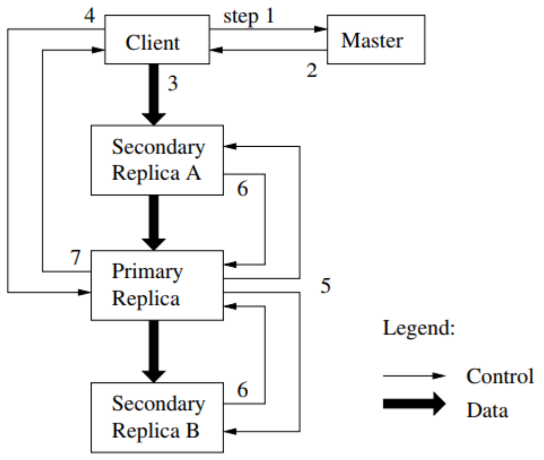
GFS used 64 MB chunk size and uses ‘lazy space allocation’ in which the physical allocation of space is delayed as long as possible. As soon as 64 MB chunk is collected, it is written down in chunk server and this saves it from problem of internal fragmentation.

6 Data Flow and Interactions

Updation of the chunks occurs in a highly sequential order. All the replicas of a chunk need to be updated sequentially to avoid inconsistency. Whenever a client requests a mutation of the chunk, the master replies the client with the address of one of the replicaas called as a ”primary”. This chunkserver will then decide a sequence in which all the replicas will be updated and then transfer the data along with the sequence to all the replicas.

6.1 Data Flow

So now we have to address the issue of how the data will flow in the server to update all the replicas of the chunk. It can be done in a tree methodology where the data is pushed into multiple branches at each node, but this will be an inefficient method since each machine’s outbound bandwidth is now divided among multiple recipients.



levels.png

Figure 2: Levels of Replicas in GFS

So in order to realize maximum throughput and minimum latency, data is linearly transferred in a highly optimized way utilizing the maximum transfer speed at each node. The sequence is decided based on nearest node in terms of distance which is easily calculated using IP address and network topology. Also pipelined implementation is being used. The chunkservers transfer the data immediately as they start receiving it. Summarizing the above discussion, the ideal elapsed time for transferring B bytes to R replicas is B/T + RL where T is the network throughput and L is latency to transfer bytes between two machines.

6.2 Snapshot

As we had mentioned in the introduction section, the snapshot operation makes a copy of a file or a directory almost instantaneously, while minimizing any interruptions of ongoing processes or data updating related mutations. Whenever the master receives any request for snapshot of a chunk, it revokes any outstanding leases and then processes the requests for snapshot. The revoking of leases ensures that any further mutations requests go throught the master.

7 Working of Master

7.1 Namespace Management and Locking

GFS Master implements lock mechanisms for managing multiple access. It has two types of locks viz. read lock and write lock. It has to avoid multiple hazards such as creating file inside a directory while it is being snapshotted. Multiple file creations in same directory can be concurrent because each can acquire read lock on directory name and write lock on the file name.

7.2 Replica Placement

Distributed system of Google has hundreds of chunkservers spread across many machine racks. Communication between two racks on different machines takes more time that machine on same racks.

Replicas of chunk have to be made to ensure availability of data and recovery from component failures which are very common in large distrbuted systems. There are three levels of replicas are maintained by master in GFS.

GFS chunk replica placement policy makes two promises. First promise, data relaibility and availability is served by spread replicas across machines in same rack as well as across racks. Second promise is the full utilization of network bandwidth.

Master is also responsible for creations, re-replications and rebalancing of chunks as and when required. Also, when chunks are deleted, GFS lazily reclaims new available memory during regular garbage collection. Stale chunks, i.e. the chunks that are out-dated, are also managed by master using version number.

8 Fault Tolerance and Diagnosis

This is most important feature that distributed systems must provide for meeting continuous service requirement.

Normal and abnormal termination are handled in same way by GFS. In fact, servers are routinely shut down just by killing the process. But because the master and the chunkservers restore their state and start in seconds no matter how they terminated, clients and other servers only experience a minor hiccup.

Master is very important for GFS to function. To recover from masters failure, the master state, logs, and checkpoints are replicated on other machines. Processes, if failed, can recover in matter of seconds to their initial state. If there is a hardware failure such as disk failure, another machine immediately takes up role of master by using a replica of its state. Since canonical names are used to address master, DNS alias is changed to relocate master.

Apart from component failure, data corruption is also detected and corrected by GFS. For example, GFS chunkservers use checksums to ensure data integrity on disks. Correction is done by using data replicas available on other machines.

Apart from these intrinsic featues of GFS, it is also supported by extensive diagnostic logging for problem isolation, debugging features, performance analysis to ensure quick recovery from fatal losses and improvements over time. Online continuous monitoring tools with minimum overhead are also deployed by the system.

9 Conclusion

GFS is made to support large-scale data processing workloads on hardware where failures are common. Few design features are used in any general file system but others are made specifically to cater unique needs of Google applications. Traditional file system assumptions and Google File System are different in terms of the workload they handle and in terms of constraints they possess. The recovery from failure and data corruption is handled differently.

High throughput is promised to concurrent readers and writers. Failures are treated as the norm, huge files are common and mostly appended rathers that randomly modified. GFS provides fault tolerance by monitoring, extensive replications and ‘fast and automatic’ recovery. Chunk replication plays a major role in recovery of servers. Repairs of the damage and compensation for lost replicas is quick and optimized for system. Checksumming is put in place to detect data corruption which becomes too common for a system with huge number of disks in the system.

References

The Google File System by Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung (Google)

Image Credits

Google Applications: ClassThink.com

Component of GFS: The Google File System by Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung (Google)

Levels of Replicas in GFS: The Google File System by Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung (Google)