International Journal of Computer Applications (0975 – 8887)
Volume 103 – No.9, October 2014

From Idea to Reality: Google File System

Aditya Kamath
MIT College of Engineering
Paud Road, Kothrud
Pune-411038.

Archit Jaiswal
MIT College of Engineering
Paud Road, Kothrud
Pune-411038.

Kranti Dive
MIT College of Engineering
Paud Road, Kothrud
Pune-411038.

ABSTRACT

In the recent times, there has been an exposure to distributed systems that exemplify some degree of transparency predominantly through distributed file systems. Any user would like the look and feel of remote files just as the local ones. The Google File System implements such features. Instrumenting a master/slave pattern, the GFS provides appending data, taking snapshots and checkpoints which in various ways help in handling critical situations and failures. In addition, duplication of chunk servers allows faster access and retrieval of requested data.

General Terms

Append, Chunk servers, snapshots, and master/slave.

Keywords

GFS, chunk server, Meta Data.

1. INTRODUCTION

A Distributed File System is a file system that merges the file systems of individual machines in network. Files are stored (distributed) in different machines in a computer network but are accessible from all machines. Another name for Distributed file system is network file system. The Andrew File System [AFS], which has a heavy client cache and uses Kerberos for authentication & the Apple Filing Protocol by Apple Computer are some examples of various file systems.

Developing an effective design is an important skill in distributed systems, requiring an awareness of the different technological choices and a thorough understanding of the requirements of the relevant application domain. The eventual goal is to come up with a consistent distributed system architecture incorporating a consistent and complete set of design choices able to address the overall requirements. This is a demanding task and one that requires considerable experience with distributed systems development. Taking into consideration these essential requirements, the Internet search giant Google implemented their own file system popularly known as the GOOGLE FILE SYSTEM [GFS]. This paper shall cover a deep insight of the GFS.

2. LITERATURE REVIEW

A general question that would strike one's mind is that what motivated and inspired Google to develop and implement its own file system? The reason is simple. Inside the Google's world of database, nothing is small. As a consequence of the services Google provides, Google faces the requirement to manage large amounts of data – including but not being limited to the crawled web content to be processed by the indexing system. So the block size of general file system was needed to be re-examined for their file system [3].

Google's operation of data management chiefly appends new data rather than over-writing existing data. Append allows clients to add information to an existing file without altering the previously written data. The web behemoth Google is continuously pioneering various applications leading to various challenges that they have to face. There are many different components that are required for these applications and therefore it is more flexible for Google if the file system and applications are co-designed. Co-designing helps in handling the components by the file system itself which increases the overall efficiency. Google implemented their file system, Google File System (GFS), handling these problems that serve their environment and application, along with general distributed file system features like scaling, security, performance, reliability, etc. [3].

GFS runs reliably on the physical architecture -that is a very large inexpensive system built from commodity hardware that might fail anytime [2]. It is the duty of GFS to constantly monitor itself and detect, tolerate, and recover promptly from component failures on a routine basis [5]. One of the assumptions made in the design of GFS is to consider disk faults, machine faults as well as network faults as being the norm rather than the exception [8]. Large streaming reads and small random reads are the two types of workloads the GFS manages. Performance-conscious applications often batch and sort their small reads to advance steadily through the file rather than go back and forth [5]. In order to meet the goals of high distribution, tolerance to high failure rates, fast-path appends and huge files the designers provided their own API to the GFS file system. In addition to *read, write, open, and create, close, delete* operations, the API offers two more specialized operations, *snapshot* and *record append*. The former operation provides an efficient mechanism to make a copy of a particular file or directory tree structure. The latter supports the common access pattern whereby multiple clients carry out concurrent appends to a given file [2]. *Pathnames* are used to identify the various files that are organized hierarchically in the directories.

3. INSIDE THE GOOGLE FILE SYSTEM

GFS is designed as a distributed file system to be run on clusters up to thousands of machines. A cluster is simply a network of computers with each cluster containing hundreds or even thousands of machines. Multiple GFS clusters are currently deployed for different purposes. The largest ones have over 1000 storage nodes, over 300 TB of disk storage, and are heavily accessed by hundreds of clients on distinct machines on a continuous basis. A GFS cluster is of a master/slave pattern which consists of a single *master* and multiple *chunk servers* (slave machines where all the data is stored) and is accessed by multiple clients executing concurrently. All data is replicated into a number of independent *chunk servers* on the data nodes for the purpose of data protection from events such as software or hardware failures, the default being three. These replicas are stored in different racks and their location is stored by the master.

Now let's learn the basic mechanism of this superlative file system. To start with, files are split into number of fixed size [64 MB] chunks. A file is formed by at least one *chunk* and there is always a scope of allocating new chunks in case of file expansion. The role of the master is to manage metadata about the file system defining the namespace for files, access control information, chunk version numbers and the mapping of each particular file to the associated set of chunks [2]. The metadata of a chunk is as small as 64 bytes and therefore all the metadata is stored in the memory by the *master*. This helps to ease the process of various data structures and algorithms and ensures great performance. Handling the system-wide activities such as chunk lease management, garbage collection of orphaned chunks, and chunk migration between chunk servers is also the job of the *master*. Clients never read and write file data through the master. Instead, a client asks the master which chunk servers it should contact. It caches this information for a limited time and interacts with the chunk servers directly for many subsequent operations. This minimization of the involvement of the master helps to avoid bottleneck condition caused by the *master*. The master does not store the persistent information of which chunk servers contain the replica of a given chunk. Instead all information is queried from the chunk servers during the startup which ensures frequent updating of location keeping it in sync. When clients need to access data starting from a particular byte offset within a file, the GFS client library will first translate this to a file name and chunk index pair. Then, it sends the master a request containing the file name and chunk index. The master replies with the appropriate chunk identifier and location of the replicas, and this information is cached in the client and used subsequently to access the data to one of the replicated chunk servers [2]. A request is then sent to the nearest replica that specifies the chunk handle and a byte range within that chunk. Further interaction of the client-master is avoided of the same chunk until the cached information is expired or the file is reopened. The next section shall discuss the dynamics and the working of the GFS.

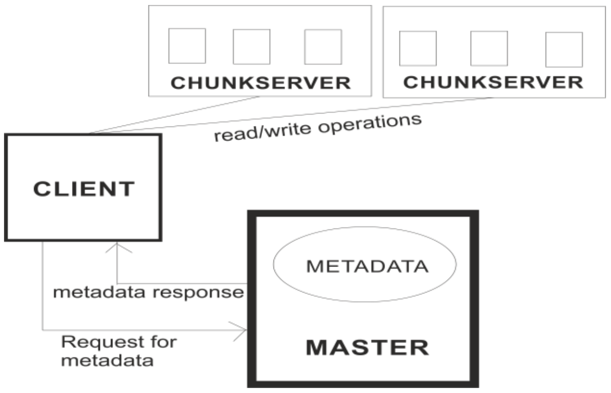

Figure 1: The GFS architecture

4. PIVOTAL OPERATIONS OF GFS

Given below is a quick overview of the working of key operations in GFS- *read, write* and *append.*

4.1 Reading a Chunk

The GFS client library is used by the clients in order to read a chunk from a file. This library then translates the request from (filename, byte range) to (filename, chunk index) and sends it to the master. The master, provided the file name and the offset within the file, looks up the identifiers of affected chunks in the metadata database and determines all chunk servers [primary-where original data is stored, as well as secondary-where the replica is stored][8] suitable to read from. The client preferably chooses the nearest chunk server and then this chunk server performs checksums (in order to detect and prevent data corruption) on the data and then transmits the requested data to the client. If a failure occurs during checksum validation, then the chunk servers reports to the master for a possible data corruption and returns an appropriate failure to the client [8].

4.2 Writing to a Chunk [Modifying]

The important difference between *read* and *write* operation is that while reading, the client reads only one chunk out of the other replicas, while in writing the client has to write in both the original as well as the replicas. In order to write to a particular file, the application has to generate a request to the client for permission to write. Once again utilizing the GFS library, it translates request from (filename, data) to (filename, chunk index), and sends it to master. If there are any other applications trying to alter the same chunk/s, the master then has to synchronize concurrent execution. The master then responds with the chunk handle and the replica (primary and secondary) locations to the client [3]. The *client* then sends a *write* command to the primary which first determines serial order for data instances stored in its buffer and writes the instances in that order to the chunk [3]. The signal order is forwarded to the other replicas commanding them to perform the *write* operation. This is carried out in a pipelined fashion. Whenever a chunk has been modified, its version number is incremented. By versioning each chunk, replicas that have missed certain modifications, [also known as stale replicas] and thus provide outdated data, can be detected by the master and appropriate action (e.g. re-replication) can be taken. If all steps of the modification procedure have been completed successfully, the requests are acknowledged [6] and success is reported to the client [7]. In order to provide updating of data, whenever a new client requests a read operation on a chunk that is currently being modified, the master pipes the request to the primary copy.

4.3 Record Append Operation

A unique feature that differentiates GFS from other distributed file systems is the *record append* operation. For better performance, this operation is optimized in a special manner by the GFS. While trying to minimize locking, GFS ensures that no race condition occurs between two concurrent *append* operations, which is commonly encountered when two applications try to append a write-shared file opened with O_APPEND on a POSIX-compliant file system [8].

Record Append is a kind of mutation that changes the contents of metadata of a chunk. When an application tries to append data on a chunk by sending a request to the client, the client pushes the data to all replicas of the last chunk of the file [5]. When the client forwards the request to the master, the primary checks whether the appending the record to the existing chunk would increase the size of the chunk more than its limit [maximum size of a chunk is 64 MB]. If this happens, then it pads the chunk to the maximum limit, commands the secondary to do the same and requests the clients to try to append to the next chunk. If the record fits within the maximum size, the primary appends the data to its replica, tells the secondary to write the data at the exact offset where it has, and finally replies success to the client [5]. If failure occurs while appending into any of the replicas, then the client retries the operation. This is the reason why replicas of the same chunk might contain duplicates of the same record or even a part of it. Thus it can be said that the append operation has *At least One* semantics.

4.4 Checkpoints & Snapshot

The structure of GFS is centralized and hence the master can prove to be a single point of failure. In order to ready restore and resume functioning during such a failure, the *operation log* is replicated over several remote machines. The master recovers its file system state by replaying the *operation log,* also known as the transaction log. An *operation log* contains a persistent record of metadata and also serves as a logical time line that defines the order of concurrent operations. It is an important aspect since every file as well as chunk and their versions are uniquely and externally identified by the logical time in which they were created. Checkpoints are created by the master when the log grows beyond a specific size. These checkpoints help in faster recovery as the master loads only the latest checkpoint from the local disk and replays only a limited number of log records after that. Hence older checkpoints are usually deleted but still some of them are stored in case of catastrophic conditions.

There is difference between *checkpoints* and another unusual operation that the GFS offers, i.e. *snapshot*. This feature provides an efficient mechanism which enables to make a copy of a file or a directory tree structure instantaneously without interrupting any currently executing mutations. This helps to branch two versions of the same data. When the master receives a snapshot request, it first revokes any outstanding leases on the chunks in the files it is about to snapshot. This ensures that any subsequent writes to these chunks will require an interaction with the master to find the lease holder [5]. The master logs the operations to the disk as soon as the leases are revoked or expired. It then stores the new log record in the memory and the newly created snapshot files point to the same chunk as the source file [5].

4.5 Handling Failure [Fault Tolerance]

What does one first do to learn the meaning of a new word, an answer to a question that has a variety of opinions? GOOGLE it. How does Google manage to handle such a large database and provide with the best answers to all the queries? There are thousands or maybe lakhs of people all around the world who '*Google'* every second. The GFS has to handle their queries and this has to be very quick. Google distributes dozens of copies of web into multiple clusters all over the world. It is a known fact that the Google File System is built and completely relies on its inexpensive commodity hardware. Hence, dealing with frequent failures such as application bugs, OS bugs, human errors, and the failure of disks, memory, connectors, networking, and power supplies is one of the greatest challenges for GFS. Monitoring, error detection and quick recovery become the key factors of fault tolerance. The master's role in restoring the data are as follows:

1. Replicate its own state in different machines across different clusters just in case the master goes down.
2. Keeping a transaction log of the metadata.
3. Periodic checkpoints of the log and its replication on different racks of machines.

Chunk replicas are also implemented in case one of the replica goes down [3].

5. CONCLUSION

In the early 21st century, there were a number of search engines like Yahoo which were being used all over the world. This paper gives an answer as to how the Google search engine was able to stand tall against its rivals and later on become the most widely used search engine throughout the globe. The ever growing use of technology in people's daily lives has bought the need to manage big data efficiently. Google has put efforts to provide a distributed system also allowing them to scale up their system when more data is to be stored. Large scale data processing along with the ability to cover up during critical situations and hardware failure, the GFS is the reason for the success of Google and its other services.

6. REFERENCES

[1] Distributed File System Design By Paul Krzyzanowski, Rutgers University – CS 417: Distributed Systems.

[2] Distributed Systems.Concepts & Design by George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair (ch. 21).

[3] Survey on Google File System by Naushad UzZaman, Survey Paper for CSC 456 (Operating Systems), University of Rochester, Fall 2007 Instructed By: Sandhya Dwarkadas.

[4] Paper Trial: Computer Science, Distributed Algorithms & Databases [The Google File System]. <the-paper-trial.org/blog/the-google-file-system/>

[5] [Ghemawat, Gobioff and Leung 2003] Sanjay Ghemawat, Howard Gobioff and Shun-Tak Leung, Google File System, ACM SIGOPS Operating Systems Review, 2003.

[6] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, D. Noveck, Network File System (NFS) version 4 Protocol, RFC 3530.

[7] J.H. Morris, M. Satyanarayanan, M.H. Conner, J.H. Howard, D.S.H. Rosenthal, F.D. Smith, Andrew, a Distributed Computing Environment, Communications of the ACM 29, 03/1986.

[8] The Google File System and its Applications in MapReduce-Johannes Passing, Seminar *Software Design*, Winter term 2007/08, Hasso Plattner Institute for Software Systems Engineering.