Seminar Report On

Google File System

Submitted by

SARITHA.S

In partial fulfillment of requirements in Degree of
Master of Technology (MTech)
In
Computer & Information Systems

DEPARTMENT OF COMPUTER SCIENCE
COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY
KOCHI 682022
2010

COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

KOCHI 682022

Certificate

This is to certify that the seminar report titled ***'GOOGLE FILE SYSTEM'*** *submitted by* ***Saritha S****, in partial fulfillment of the requirements of the award of MTech Degree in Computer & Information Sytems, Cochin University of Science and Technology, is a bonafide record of the seminar presented by her during the academic year 2010.*

Mr. G.Santhosh Kumar
Lecturer
Dept. of Computer Science

Prof. Dr.K.Poulose Jacob
Director
Dept. of Computer Science

Abstract

The great success of Google Inc. is attributed not only to its efficient search algorithm, but also to the underlying commodity hardware and, thus the file system. As the number of applications run by Google increased massively, Google's goal became to build a vast storage network out of inexpensive commodity hardware. Google created its own file system, named as Google File System. Google File System was innovatively created by Google engineers and ready for production in record time in a span of one year in 2003, which speeded Google's market thereafter. Google File system is the largest file system in operation. Formally, Google File System (GFS) is a scalable distributed file system for large distributed data intensive applications.

In the design phase of GFS, points which were given stress includes component failures are the norm rather than the exception, files are huge in the order of MB & TB and files are mutated by appending data. The entire file system is organized hierarchically in directories and identified by pathnames. The architecture comprises of a single master, multiple chunk servers and multiple clients. Files are divided into chunks, which is the key design parameter. Google File System also uses leases and mutation order in their design to achieve consistency and atomicity. As of fault tolerance, GFS is highly available, replicas of chunk servers and master exists.

Key words: Distributed File System, data, Google File System, master, chunk, chunkservers, garbage collection, fault tolerance.

Index

1. Introduction

Google has designed and implemented a scalable distributed file system for their large distributed data intensive applications. They named it Google File System, GFS. Google File System is designed by *Sanjay Ghemawat, Howard Gobioff and Shun-Tak Leung* of Google in 2002-03. GFS provides fault tolerance, while running on inexpensive commodity hardware and also serving large number of clients with high aggregate performance. Even though the GFS shares many similar goals with previous distributed file systems, the design has been driven by Google's unique workload and environment. Google had to rethink the file system to serve their "very large scale" applications, using inexpensive commodity hardware.

Google give results faster and more accurate than other search engines. Definitely the accuracy is dependent on how the algorithm is designed. Their initial search technology is Page Rank Algorithm designed by Garry Brin and Larry Page in 1998. And currently they are merging the technology of using both software and hardware in smarter way. Now the field of Google is beyond the searching. It supports uploading video in their server, Google Video; it gives email account of few gigabytes to each user, Gmail; it has great map applications like Google Map and Google Earth; Google Product application, Google News application, and the count goes on. Like, search application, all these applications are heavily data intensive and Google provides the service very efficiently.

If we describe a bit more about Google operation in terms of lower level rather than application perspective then we can understand why a unique file system is required to serve their need. Google store the data in more than 15 thousands commodity hardware, which contains the dozens of copies of the entire web and when it comes to serving the query, Google serves thousands of queries per second, one query reads 100's of MBs of data and one query consumes 10's of billions of CPU cycles. These numbers tells us how large-scale Google applications are and how efficient it has to be to serve it all. They should have very smart read, write type operations to do it in this large scale. They are

storing all the data in 15K+ inexpensive commodity hardware. This is a big challenge to their system because these PCs can go down anytime and considering the number of PCs and the lifetime of the PCs, it is obvious that something is failing everyday. So they designed it in a way that it is not an exception but it is natural that something will fail everyday. GFS tries to handle these exceptions and also other Google specific challenges in their distributed file system, GFS.

2. Design Overview

2.1 Assumptions

In designing a file system for Google's needs, they have been guided by assumptions that offer both challenges and opportunities.

• The system is built from many inexpensive commodity components that often fail. It must constantly monitor itself and detect, tolerate, and recover promptly from component failures on a routine basis.

• The system stores a modest number of large files. Usually 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 need not optimize for them.

• The workloads primarily consist of two kinds of reads: large streaming reads and small random reads. In large streaming reads, individual operations typically read hundreds of KBs, more commonly 1 MB or more. Successive operations from the same client often read through a contiguous region of a file. A small random read typically reads a few KBs at some arbitrary offset. Performance-conscious applications often batch and sort their small reads to advance steadily through the file rather than go back and forth.

• The workloads also have many large, sequential writes that append data to files. Typical operation sizes are similar to those for reads. Once written, files are seldom modified again. Small writes at arbitrary positions in a file are supported but do not have to be efficient.

• The system must efficiently implement well-defined semantics for multiple clients that concurrently append to the same file. The files are often used as producer consumer queues or for many-way merging. Hundreds of producers, running one per machine, will

concurrently append to a file. Atomicity with minimal synchronization overhead is essential. The file may be read later, or a consumer may be reading through the file simultaneously.

• High sustained bandwidth is more important than low latency. Most of the target applications place a premium on processing data in bulk at a high rate, while few have stringent response time requirements for an individual read or write

2.2 Google File System Architecture

A GFS cluster consists of a single *master* and multiple *chunkservers* and is accessed by multiple *clients*. The basic analogy of GFS is *master* maintains the metadata; client contacts the master and retrieves the metadata about *chunks* that are stored in chunkservers; next time, client directly contacts the chunkservers. Figure 1 describes these steps more clearly.

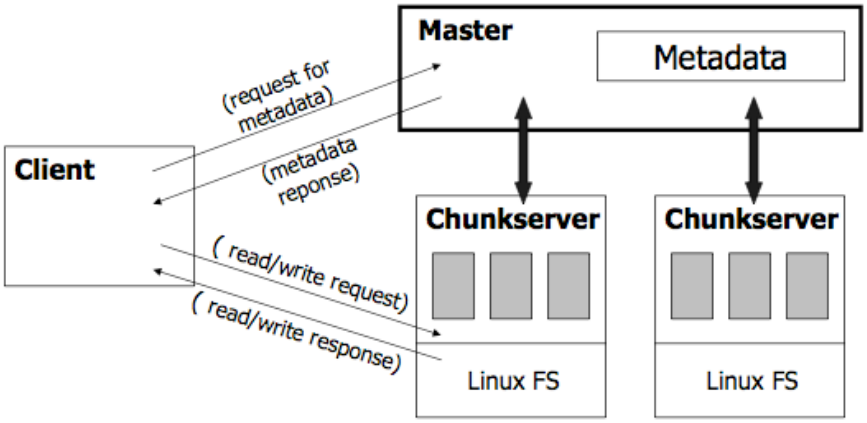

Figure 1

Each of these is typically a commodity Linux machine running a user-level server process. Files are divided into fixed-size *chunks*. Each chunk is identified by an immutable and globally unique 64 bit *chunk handle* assigned by the master at the time of chunk creation. Chunkservers store chunks on local disks as Linux files and read or write

chunk data specified by a chunk handle and byte range. For reliability, each chunk is replicated on multiple chunkservers. By default, three replicas are stored, though users can designate different replication levels for different regions of the file namespace. The master maintains all file system metadata. This includes the namespace, access control information, the mapping from files to chunks, and the current locations of chunks. It also controls system-wide activities such as chunk lease management, garbage collection of orphaned chunks, and chunk migration between chunkservers. The master periodically communicates with each chunkserver in HeartBeat messages to give it instructions and collect its state. GFS client code linked into each application implements the file system API and communicates with the master and chunkservers to read or write data on behalf of the application.

Clients interact with the master for metadata operations, but all data-bearing communication goes directly to the chunkservers. Neither the client nor the chunkserver caches file data. Client caches offer little benefit because most applications stream through huge files or have working sets too large to be cached. Not having them simplifies the client and the overall system by eliminating cache coherence issues. (Clients do cache metadata, however.) Chunkservers need not cache file data because chunks are stored as local files and so Linux's buffer cache already keeps frequently accessed data in memory.

Before going into basic distributed file system operations like read, write, we will discuss the concept of chunks, metadata, master, and will also describe how master and chunkservers communicates.

2.2.1 Chunk

Chunk in GFS is very important design decision. It is similar to the concept of block in file systems, but much larger than the typical block size. Compared to the few KBs of general block size of file systems, the size of chunk is 64 MB. This design was to help in the unique environment of Google. As explained in the introduction, in Google's world, nothing is small. They work with TBs of data and multiple-GB files are very common.

Their average file size is around 100MB, so 64MB works very well for them; in fact it was needed for them. It has few benefits, e.g. it doesn't need to contact master many times, it can gather lots of data in one contact, and hence it reduces client's need to contact with the master, which reduces loads from the master; it reduces size of metadata in master, (bigger the size of *chunks*, less number of *chunks* available. e.g. with 2 MB chunk size for 100 MB data, we have 50 chunks; again with 10 MB chunk size for same 100 MB, we have 10 chunks), so we have less chunks and less metadata for chunks in the master; on large chunks the client can perform many operations; and finally because of lazy space allocation, there are no internal fragmentation, which otherwise could be a big downside against the large chunk size.

A probable problem in this chunk size was that some small file consisting of a small number of chunks can be accessed many times, thus developing a hotspot. But in practice this is not a major issue, as Google applications mostly read large multi-chunk files sequentially. Later they came up with a solution by storing such files with a high replication factor.

The other properties of chunk that need to be mentioned are, chunks are stored in chunkserver as file, chunk handle, i.e., chunk file name, is used as a reference link. For each chunk there will be replicas across multiple chunkservers.

2.2.2 Metadata

The master stores three major types of metadata: the file and chunk namespaces, the mapping from files to chunks, and the location of each chunk's replicas. Among these three, the first two types (namespaces and file-to-chunk mapping) are kept persistent by keeping the log of mutations to an *operation log* stored on the master's local disk. This operation log is also replicated on remote machines. In case the master crashes anytime, it can update the master state simply, reliably, and without risking inconsistency with the help of these operation logs. The master doesn't store chunk location information persistently, instead it asks each chunkservers about its chunks when it starts up or when a chunkserver joins the cluster.

Since metadata is stored in memory, master operations are fast. Furthermore, it is easy and efficient for the master to periodically scan through its entire state in the background. This periodic scanning is used to implement chunk garbage collection, re-replication in the presence of chunkserver failures, and chunk migration to balance load and disk space One potential concern for this memory-only approach is that the number of chunks and hence the capacity of the whole system is limited by how much memory the master has. This is not a serious limitation in practice. The master maintains less than 64 bytes of metadata for each 64 MB chunk. Most chunks are full because most files contain many chunks, only the last of which may be partially filled. Similarly, the file namespace data typically requires less then 64 bytes per file because it stores file names compactly using prefix compression.

2.2.3 Master

Master is a single process running on a separate machine that stores all metadata, e.g. file namespace, file to chunk mappings, chunk location information, access control information, chunk version numbers, etc. Clients contact master to get the metadata to contact the chunkservers.

3. System Interaction

Master and chunkservers communicate regularly to obtain the state, if the chunkserver is down, if there is any disk corruption, if any replicas got corrupted, which chunk replicas store chunkserver, etc. Master also sends instruction to the chunkservers for deleting existing chunks, creating new chunks.

3.1 Read Algorithm

We have explained the concept of chunk, metadata, master, and also briefly explained the communication process between client, master, and chunkservers in different sections. Now we will explain few basic operations of a distributed file systems, like Read, Write and also Record Append that is another basic operation for Google. In this section, we will see how the read operation works.

Following is the algorithm for the Read operation, with Figures explaining the part of the algorithm.

1. Application originates the read request
2. GFS client translates the request form (filename, byte range) -> (filename, chunk index), and sends it to master
3. Master responds with chunk handle and replica locations (i.e. chunkservers where the replicas are stored)

(Figure 2 describes the steps)

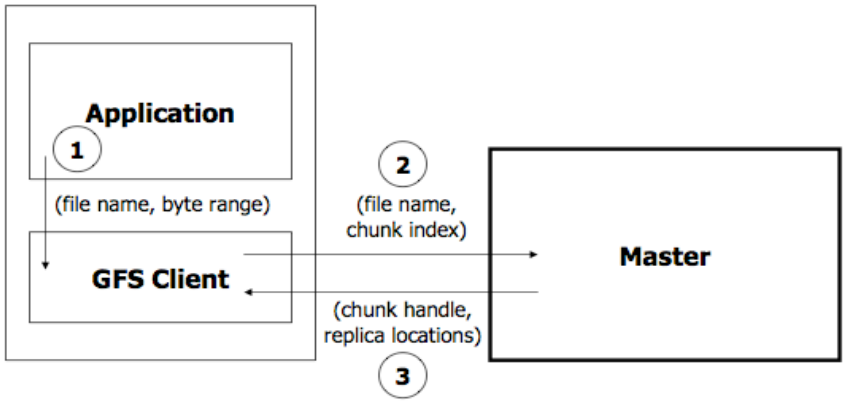

Figure 2

4. Client picks a location and sends the (chunk handle, byte range) request to the location
5. Chunkserver sends requested data to the client
6. Client forwards the data to the application

(Figure 3 describes the steps)

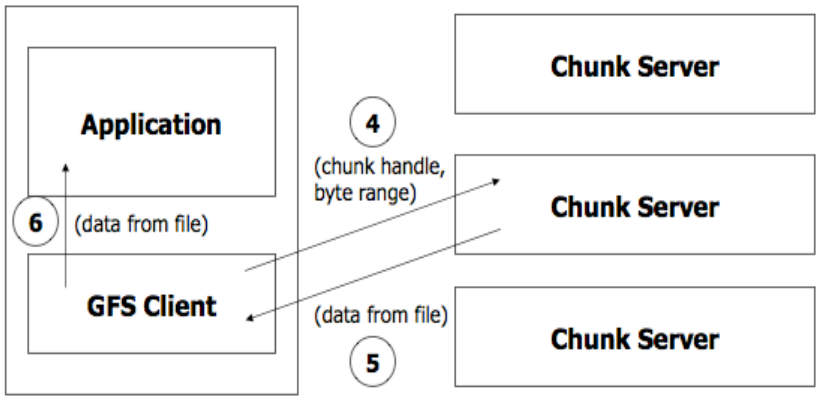

Figure 3

3.2 Write Algorithm

3.2.1 Leases and Mutation

A mutation is an operation that changes the contents or metadata of a chunk such as a write or an append operation. Each mutation is performed at all the chunk's replicas. Leases are used to maintain a consistent mutation order across replicas. The master grants a chunk lease to one of the replicas, which we call the *primary*. The primary picks a serial

order for all mutations to the chunk. All replicas follow this order when applying mutations. Thus, the global mutation order is defined first by the lease grant order chosen by the master, and within a lease by the serial numbers assigned by the primary.

The lease mechanism is designed to minimize management overhead at the master. A lease has an initial timeout of 60 seconds. However, as long as the chunk is being mutated, the primary can request and typically receive extensions from the master indefinitely. These extension requests and grants are piggybacked on the HeartBeat messages regularly exchanged between the master and all chunkservers.

The master may sometimes try to revoke a lease before it expires (e.g., when the master wants to disable mutations on a file that is being renamed). Even if the master loses communication with a primary, it can safely grant a new lease to another replica after the old lease expires.

Write algorithm is similar to Read algorithm, in terms of contacts between client, master, and chunkservers. Google keeps at least three replicas of each chunks, so in Read, we just read from one of the chunkservers, but in case of Write, it has to write in all three chunkservers, this is the main difference between read and write.

Following is the algorithm with related figures for the Write operation.

1. Application originates the request
2. GFS client translates request from (filename, data) -> (filename, chunk index), and sends it to master
3. Master responds with chunk handle and (primary + secondary) replica locations (Figure 4 describes the steps)

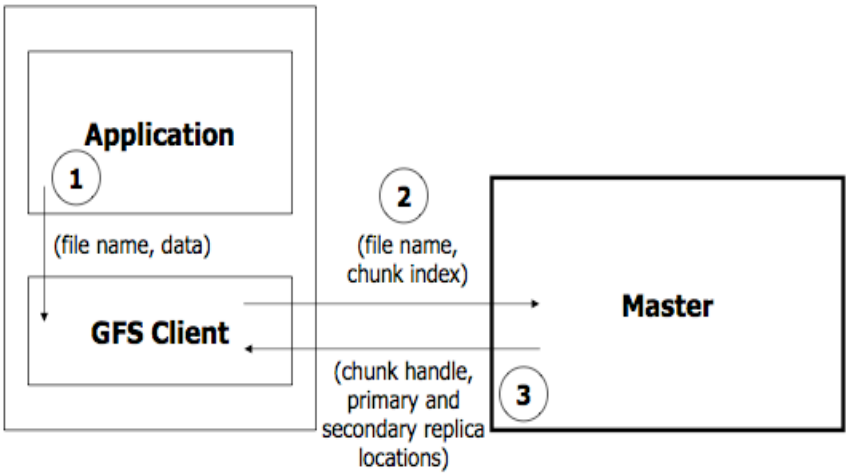

Figure 4

4. Client pushes write data to all locations. Data is stored in chunkservers' internal buffers

(Figure 5 describes the step)

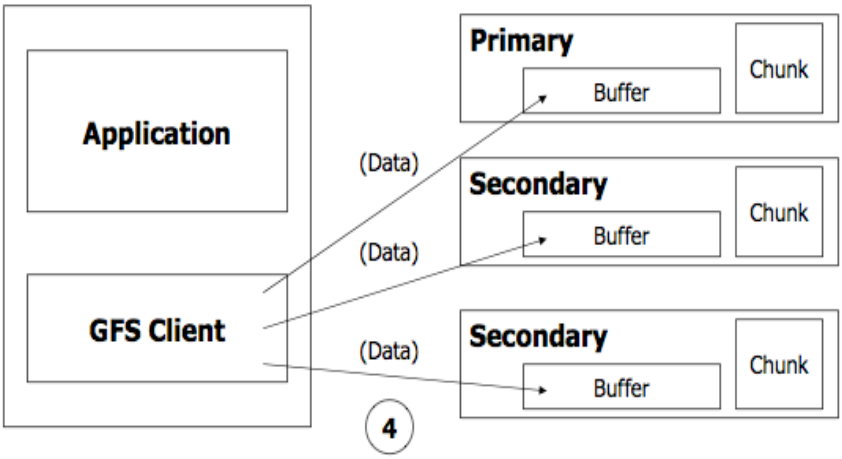

Figure 5

5. Client sends write command to primary

6. Primary determines serial order for data instances stored in its buffer and writes the instances in that order to the chunk

7. Primary sends the serial order to the secondaries and tells them to perform the write

(Figure 6 describes the steps)

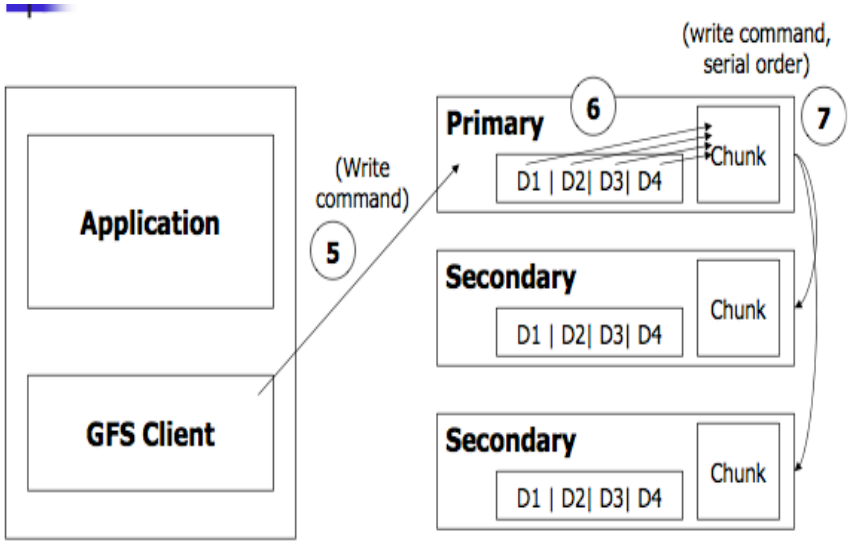

Figure 6

8. Secondaries respond to the primary

9. Primary responds back to the client

Note: If write fails at one of chunkservers, client is informed and retries the write

(Figure 7 describes the steps)

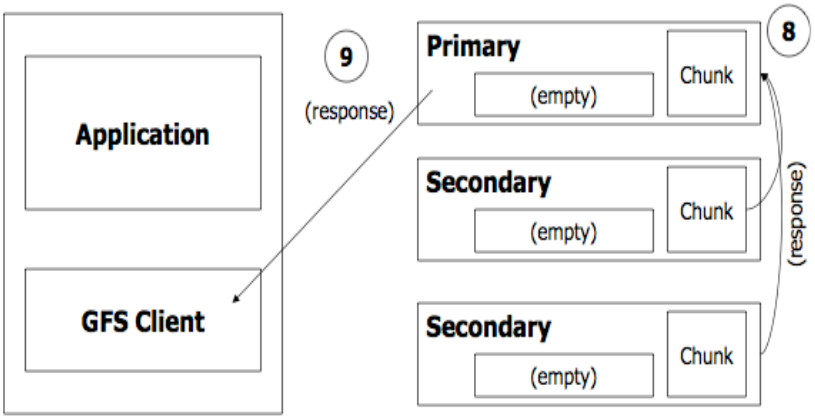

Figure 7

3.3 Record Append Algorithm

Record Append is an important operation for Google, which is not very common in other distributed file system. The reason they had added this operation is, in their environment, it is less likely to overwrite in the existing data, rather appending new data is more likely. So in the record append they merge results from multiple machines in one file and they do this by using file as producer - consumer queue.

Record append is a kind of mutation and follows the control flow as in the write algorithm with only a little extra logic at the primary. The client pushes the data to all replicas of the last chunk of the file. Then, it sends its request to the primary. The primary checks to see if appending the record to the current chunk would cause the chunk to exceed the maximum size (64 MB). If so, it pads the chunk to the maximum size, tells secondaries to do the same, and replies to the client indicating that the operation should be retried on the next chunk. (Record append is restricted to be at most one-fourth of the maximum chunk size to keep worst case fragmentation at an acceptable level.) If the record fits within the maximum size, which is the common case, the primary appends the data to its replica, tells the secondaries to write the data at the exact offset where it has, and finally replies success to the client.

If a record appends fails at any replica, the client retries the operation. As a result, replicas of the same chunk m y contain different data possibly including duplicates of the same record in whole or in part. GFS does not guarantee that all replicas are bytewise identical. It only guarantees that the data is written at least once as an atomic unit. This property follows readily from the simple observation that for the operation to report success, the data must have been written at the same offset on all replicas of some chunk. Furthermore, after this, all replicas are at least as long as the end of record and therefore any future record will be assigned a higher offset or a different chunk even if a different replica later becomes the primary.

Algorithm for Record Append:

1. Application originates record append request.
2. GFS client translates requests and sends it to master.
3. Master responds with chunk handle and (primary + secondary) replica locations.
4. Client pushes write data to all replicas of the last chunk of the file.
5. Primary checks if record fits in specified chunk
6. If record doesn't fit, then the primary:
   - Pads the chunk
   - Tell secondaries to do the same
   - And informs the client
   - Client then retries the append with the next chunk
7. If record fits, then the primary:
   - Appends the record
   - Tells secondaries to write data at exact offset
   - Receives responses from secondaries
   - And sends final response to the client

3.4 Snap Shot

The snapshot operation makes a copy of a file or a directory tree (the "source") almost instantaneously, while minimizing any interruptions of ongoing mutations. The users use it to quickly create branch copies of huge data sets (and often copies of those copies, recursively), or to checkpoint the current state before experimenting with changes that can later be committed or rolled back easily .

Standard copy-on-write techniques are used to implement snapshots. 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. This will give the master an opportunity to create a new copy of the chunk first. After the leases have been revoked or have expired, the master logs the operation to disk. It then applies this log record to its in-memory state by duplicating the metadata for the source file or directory tree. The newly created snapshot files point to the same chunks as the source files.

4. Master Operation

The master executes all namespace operations. In addition, it manages chunk replicas throughout the system: it makes placement decisions, creates new chunks and hence replicas, and coordinates various system-wide activities to keep chunks fully replicated, to balance load across all the chunkservers, and to reclaim unused storage.

4.1 Name space management and locking

Multiple operations are to be active and use locks over regions of the namespace to ensure proper serialization. Unlike many traditional file systems, GFS does not have a per-directory data structure that lists all the files in that directory. Nor does it support aliases for the same file or directory (i.e, hard or symbolic links in Unix terms). GFS logically represents its namespace as a lookup table mapping full pathnames to metadata. With prefix compression, this table can be efficiently represented in memory. Each node in the namespace tree (either an absolute file name or an absolute directory name) has an associated read-write lock.

Each master operation acquires a set of locks before it runs. One nice property of this locking scheme is that it allows concurrent mutations in the same directory. For example, multiple file creations can be executed concurrently in the same directory: each acquires a read lock on the directory name and a write lock on the file name. The read lock on the directory name suffices to prevent the directory from being deleted, renamed, or snap shotted. The write locks on file names serialize attempts to create a file with the same name twice.

4.2 Replica Placement

A GFS cluster is highly distributed at more levels than one. It typically has hundreds of chunkservers spread across many machine racks. These chunkservers in turn may be accessed from hundreds of clients from the same or different racks. Communication between two machines on different racks may cross one or more network switches.

Additionally, bandwidth into or out of a rack may be less than the aggregate bandwidth of all the machines within the rack.

The chunk replica placement policy serves two purposes: maximize data reliability and availability, and maximize network bandwidth utilization. For both, it is not enough to spread replicas across machines, which only guards against disk or machine failures and fully utilizes each machine's network bandwidth. Chunk replicas are also spread across racks. This ensures that some replicas of a chunk will survive and remain available even if an entire rack is damaged or offline

4.3 Creation, Re-replication and Balancing Chunks

When the master *creates* a chunk, it chooses where to place the initially empty replicas. It considers several factors.

(1)We want to place new replicas on chunkservers with below-average disksp ace utilization. Over time this will equalize disk utilization across chunkservers. (2) We want to limit the number of "recent" creations on each chunkserver. (3) As discussed above, we want to spread replicas of a chunk across racks.

The master *re-replicates* a chunk as soon as the number of available replicas falls below a user-specified goal. This could happen for various reasons: a chunkserver becomes unavailable, it reports that its replica may be corrupted, one of its disks is disabled because of errors, or the replication goal is increased. Each chunk that needs to be re-replicated is prioritized based on several factors. One is how far it is from its replication goal. For example, we give higher priority to a chunk that has lost two replicas than to a chunk that has lost only one.

Finally, the master *rebalances* replicas periodically: it examines the current replica distribution and moves replicas for better disks pace and load balancing. Also through

this process, the master gradually fills up a new chunkserver rather than instantly swamps it with new chunks and the heavy write traffic that comes with them.

5. Garbage Collection

After a file is deleted, GFS does not immediately reclaim the available physical storage. It does so only lazily during regular garbage collection at both the file and chunk levels. When a file is deleted by the application, the master logs the deletion immediately just like other changes. However instead of reclaiming resources immediately, the file is just renamed to a hidden name that includes the deletion timestamp. During the master's regular scan of the file system namespace, it removes any such hidden files if they have existed for more than three days (the interval is configurable). Until then, the file can still be read under the new, special name and can be undeleted by renaming it back to normal. When the hidden file is removed from the namespace, its in memory metadata is erased. This effectively severs its links to all its chunks.

In a similar regular scan of the chunk namespace, the master identifies orphaned chunks (i.e., those not reachable from any file) and erases the metadata for those chunks. In a HeartBeat message regularly exchanged with the master, each chunkserver reports a subset of the chunks it has, and the master replies with the identity of all chunks that are no longer present in the master's metadata. The chunkserver is free to delete its replicas of such chunks.

6. Fault Tolerance

6.1 High Availability

6.1.1 Fast Recovery

Both the master and the chunkserver are designed to restore their state and start in seconds no matter how they terminated. In fact, there is not distinction between normal and abnormal termination; servers are routinely shut down just by killing the process. Clients and other servers experience a minor hiccup as they time out on their outstanding requests, reconnect to the restarted server, and retry.

6.1.2 Chunk Replication

Each chunk is replicated on multiple chunkservers on different racks. Users can specify different replication levels for different parts of the file namespace. The default is three. The master clones existing replicas as needed to keep each chunk fully replicated as chunkservers go offline or detect corrupted replicas through checksum verification

6.1.3 Master replication

The master state is replicated for reliability. Its operation log and checkpoints are replicated on multiple machines. A mutation to the state is considered committed only after its log record has been flushed to disk locally and on all master replicas. For simplicity, one master process remains in charge of all mutations as well as background activities such as garbage collection that change the system internally.

When it fails, it can restart almost instantly. If its machine or disk fails, monitoring infrastructure outside GFS starts a new master process elsewhere with the replicated operation log.

Moreover, "shadow" masters provide read-only access to the file system even when the primary master is down. They are shadows, not mirrors, in that they may lag the primary slightly, typically fractions of a second. They enhance read availability for files that are not being actively mutated or applications that do not mind getting slightly stale results.

6.2 Data Integrity

Each chunkserver uses checksumming to detect corruption of stored data. A chunk is broken up into 64 KB blocks. Each has a corresponding 32 bit checksum. Like other metadata, checksums are kept in memory and stored persistently with logging, separate from user data.

For reads, the chunkserver verifies the checksum of data blocks that overlap the read range before returning any data to the requester, whether a client or another chunkserver. Therefore chunkservers will not propagate corruptions to other machines. If a block does not match the recorded checksum, the chunkserver returns an error to the requestor and reports the mismatch to the master. In response, the requestor will read from other replicas, while the master will clone the chunk from another replica. After a valid new replica is in place, the master instructs the chunkserver that reported the mismatch to delete its replica.

7. Challenges in GFS

Most of Google's mind-boggling store of data and its ever-growing array of applications continue to rely upon GFS these days. Many adjustments have been made to the file system along the way, and together with a fair number of accommodations implemented within the applications that use GFS; Google has made the journey possible.

Problems started to occur once the size of the underlying storage increased. Going from a few hundred terabytes up to petabytes, and then up to tens of petabytes that really required a proportionate increase in the amount of metadata the master had to maintain. Also, operations such as scanning the metadata to look for recoveries all scaled linearly with the volume of data. So the amount of work required of the master grew substantially. The amount of storage needed to retain all that information grew as well.

In addition, this proved to be a bottleneck for the clients, even though the clients issue few metadata operations themselves. When there are thousands of clients all talking to the master at the same time, given that the master is capable of doing only a few thousand operations a second, the average client isn't able to command all that many operations per second. There are applications such as MapReduce, where there are a thousand tasks, each wanting to open a number of files. Obviously, it would take a long time to handle all those requests, and the master would be under a fair amount of duress.

Under the current schema for GFS, there is one master per cell. As a consequence, people generally ended up with more than one cell per data center. Google also ended up doing "multi-cell" approach, which basically made it possible to put multiple GFS masters on top of a pool of chunkservers. That way, the chunkservers could be configured to have, say, eight GFS masters assigned to them, and that would give us at least one pool of underlying storage—with multiple master heads on it. Then the application was responsible for partitioning data across those different cells.

Applications would tend to use either one master or a small set of the masters. A namespace file describes how the log data is partitioned across those different cells and basically serves to hide the exact partitioning from the application. But this is all fairly static.

While adjustments were continually made in GFS to make it more accommodating to all the new use cases, the applications themselves were also developed with the various strengths and weaknesses of GFS in mind.

8. Conclusion

The Google File System demonstrates the qualities essential for supporting large-scale data processing workloads on commodity hardware. While some design decisions are specific to the unique setting, many may apply to data processing tasks of a similar magnitude and cost consciousness. Google started work on GFS by reexamining traditional file system assumptions in light of current and anticipated application workloads and technological environment.

We treat component failures as the norm rather than the exception, optimize for huge files that are mostly appended to (perhaps concurrently) and then read (usually sequentially), and both extend and relax the standard file system interface to improve the overall system. The system provides fault tolerance by constant monitoring, replicating crucial data, and fast and automatic recovery. Chunk replication allows us to tolerate chunkserver failures. The frequency of these failures motivated a novel online repair mechanism that regularly and transparently repairs the damage and compensates for lost replicas as soon as possible. Additionally, check summing is used to detect data corruption at the disk or IDE subsystem level, which becomes all too common given the number of disks in the system.

The design delivers high aggregate throughput to many concurrent readers and writers performing a variety of tasks. This is achieved by separating file system control, which passes through the master, from data transfer, which passes directly between chunkservers and clients. Master involvement in common operations is minimized by a large chunk size and by chunk leases, which delegates authority to primary replicas in data mutations. This makes possible a simple, centralized master that does not become a bottleneck. GFS has successfully met the storage needs and is widely used within Google as the storage platform for research and development as well as production data processing. It is an important tool that enables Google to continue to innovate and attack problems on the scale of the entire web.

References

[1] Sanjay Ghemawat, Howard Gobioff and Shun-Tak Leung, The Google File System, ACM SIGOPS Operating Systems Review, Volume 37, Issue 5, December 2003.

[2] Sean Quinlan, Kirk McKusick "GFS-Evolution and Fast-Forward" Communications of the ACM, Vol 53, March 2010.

[3] Naushad Uzzman, Survey on Google File System, Conference on SIGOPS at University of Rochester, December 2007.

[4] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In Proceedings of the 16th ACM Symposium on Operating System Principles, pages 224–237, October 1997.