

# Survey on Google File System

Naushad UzZaman  
naushad@cs.rochester.edu

## Abstract

In this survey paper we examine how Google File System works based on the paper *Google File System* by Sanjay Ghemawat, Howard Gobioff and Shun-Tak Leung. Google has designed and implemented a scalable distributed file system for their large distributed data-intensive applications. They named it Google File System, GFS. 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, their design has been driven by their unique workload and environment. They had to rethink the file system to serve their “very large scale” applications, using inexpensive commodity hardware! The largest cluster to date of their paper provides hundreds of terabytes of storage across thousands of disks on over a hundred of machines, and it is concurrently accessed by hundreds of clients.

In this survey paper, we describe the GFS briefly and also how it works for few operations like read, write, record append, then we describe briefly how they do the fault tolerance and finally conclude by showing the evaluation results of GFS.

## Google Applications

Who doesn't know about Google? If you don't know what Google is, then google the word Google! Because of its success and popularity

Google has reached the height that the word google (name of the search engine google, noun) has become a verb now and people use it frequently to mean to search something. The question is how google works? How it can 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 described in [Brin and Page 1998] [Page et al 98] and the current technology of using both software and hardware in smarter way is described from 50 thousand feet above in [Google-Technology]. For the faster retrieval of results, just good algorithm will not help, the use of hardware and file system has to consider the google's environment of handling something in such a large scale, which claims to have dozens of copies of entire web and handles more than 15K commodity class PCs!

Now 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. The question is how Google does it?

## Google Operations

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 the

multiple clusters are distributed worldwide. 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 can tell you 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.

We will describe the motivation behind the GFS in more detail in later sections; here we just wanted to give some idea about the motivation behind the file system, relating its day-to-day operations. Before going into much detail, we will describe few distributed file system, to make the readers understand why Google needed to come up with their very own distributed file system, GFS.

## Distributed File System

Distributed File System<sup>1</sup> is a file system that joins together 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. Distributed file systems are also called network file systems.

There have been many distributed file systems that are for educational research to commercial use, available under from GPL license to

commercial use. Here are few examples of Distributed File Systems<sup>2</sup>:

- Examples of Distributed file systems
  - *Andrew File System (AFS)* is scalable and location independent, it has a heavy client cache and uses Kerberos for authentication. Implementations include the original from IBM (earlier Transarc), Arla and OpenAFS.
  - *Network File System (NFS)* originally from Sun Microsystems is the standard in UNIX-based networks. NFS may use Kerberos authentication and a client cache.
  - *Apple Filing Protocol (AFP)* from Apple Computer. AFP may use Kerberos authentication.
- Examples of Distributed fault tolerant file systems
  - *Coda* from Carnegie Mellon University focuses on bandwidth-adaptive operation (including disconnected operation) using a client-side cache for mobile computing. It is a descendant of AFS-2. It is available for Linux under the GPL.
  - *Distributed File System (Microsoft) (Dfs)* from Microsoft focuses on location transparency and high availability. Available for Windows under a proprietary software license.

We gave examples of different distributed file systems and also fault tolerant distributed file systems. Now we will describe two types in more details, namely AFS (Andrew File System) and CODA, with their design goals.

---

<sup>1</sup> Definition of Distributed File System is borrowed from <<http://www.eos.ncsu.edu/guide/glossary.html>>

---

<sup>2</sup> Examples borrowed from [wiki-filesystem]

## **Andrew File System (AFS)**

AFS is a distributed file system created in the Carnegie Mellon University Andrew Project, and later, a software product of Transarc Corporation, IBM, and OpenAFS. AFS distributes, stores, and joins files on networked computers. This distributed file system software makes it possible for users to access information located on any computer in a network.

### **Design goals of AFS (Andrew File System)<sup>3</sup>**

- Maximum performance
- Ability to handle large number of users
- Scalability
- To be able to handle inevitable expansions
- Reliability to ensure maximum uptime and availability
- To ensure computers are available to handle queries
- Security

In summary, the Andrew File system has benefit over traditional file systems in terms of security and scalability. AFS can handle very large-scale systems. More on AFS can be found from wikipedia page of AFS [wiki-AFS].

## **CODA<sup>4</sup>**

Coda is another network file system developed as a research project at Carnegie Mellon University since 1987 under the direction of Mahadev Satyanarayanan. It descended directly from an older version of AFS (AFS-2) and offers many similar features. The InterMezzo file system was inspired by Coda. Coda is still under development, though the focus has shifted

---

3

[http://www.osweekly.com/index.php?option=com\\_content&Itemid&task=view&id=224](http://www.osweekly.com/index.php?option=com_content&Itemid&task=view&id=224)

<sup>4</sup> Description of Coda is mostly borrowed from [Coda] and [wiki-Coda]

from research to creating a robust product for commercial use.

### **Features of CODA**

- Disconnected operation for mobile computing
- Freely available under a liberal license
- High performance through client side persistent caching
- Server replication
- Security model for authentication, encryption and access control
- Continued operation during partial network failures in server network
- Network bandwidth adaptation
- Good scalability
- Well defined semantics of sharing, even in the presence of network failures

In summary, Coda is descendant from AFS, so it has the pros of AFS with more focus on improving reliability and performance, ports to important platforms and also extensions in functionality. This is also a fault tolerant distributed file system, so file systems like these could have been an option for Google. But why does Google need their own-implemented file system? That still remains the question! In the next chapter we will try to give the answer to that. Interested readers can refer to the RELATED WORK section of [Ghemawat, Gobioff and Leung 2003] paper to get in-depth understanding of the similarity and dissimilarities between GFS and other distributed file systems.

## **Motivation behind Google File System, GFS**

In last few sections, we have been asking the question “Why does Google need its own distributed file system”, in this section we will try to answer that question. We will at first present the challenges Google faces in its day-

to-day operations to show its motivations behind a new file system.

In earlier section “Google Operations”, we described some reasoning behind why GFS, Google File System, needs to be fault tolerant. Here is a recap again. For Google’s day-to-day work they have 15K+ commodity hardware (multiple clusters distributed worldwide) that keeps the dozens of copies of entire web. It has to serve to queries very fast and there are thousands of queries sent every second. So it is understandable that the problems can be very often caused in many ways, e.g. application bugs, OS bugs, human errors, and the failure of disks, memory, connectors, networking, and power supplies. To handle this situation in large scale, the monitoring, error detection, fault tolerance, and fast recovery, everything has to be built-in very efficiently. Previous file systems might have the built-in fault-tolerance system, e.g. Coda, but Google’s fault-tolerance system is more intensive and it is their one of their first priorities, unlike other file systems.

In Google’s world, nothing is small. They operate on TBs of data and multi-GB files are very common. So the block size of general file system was needed to be re-examined for their file system.

Google’s operations mostly append new data rather than over-writing existing data. So google has to consider the append operation in their file system along with standard read and write operations.

It is more flexible for Google, if the file system and the applications are co-designed. Google pioneered the advance web searching technology and also continuously pioneering many applications, so they face many challenges that others might not even face before. So, there could be many components that they need for their application, and it is better in terms of efficiency, if that is handled in the file system rather than application. Co-designing the file system and applications helps a lot in this regard and also enables to design the file system

keeping in mind their current and anticipated applications.

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. In the next sections, we will describe different components of GFS.

## GFS Architecture

### Analogy

A GFS cluster consists of a single *master* and multiple *chunkserver*s 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 *chunkserver*s; next time, *client* directly contacts the *chunkserver*s. Figure 1 describes these steps more clearly.

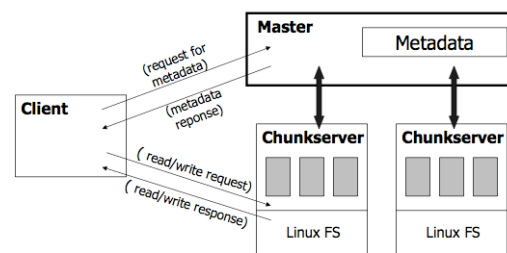


Figure: GFS architecture<sup>5</sup>

Before going into basic distributed file system operations like read, write, we will describe the

<sup>5</sup> Most of the figures in this paper are borrowed from the presentation by the authors in the conference. All rights reserves to the authors. The presentation is available online at: [http://blog.namics.com/2004/The\\_Google\\_File\\_System%20Final.pdf](http://blog.namics.com/2004/The_Google_File_System%20Final.pdf)

concept of *chunks*, *metadata*, *master*, and will also describe how *master* and *chunkservers* communicate.

## 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 motivation section, 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, some small file consisting of a small number of chunks can be accessed many times! 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 *chunks* there will be replicas across multiple *chunkservers*.

## 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 *chunkserver* about its *chunks* when it starts up or when a *chunkserver* joins the cluster.

## 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*.

## Communication between Master and Chunkservers

*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*.

## 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.

### Algorithm for Read:

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)

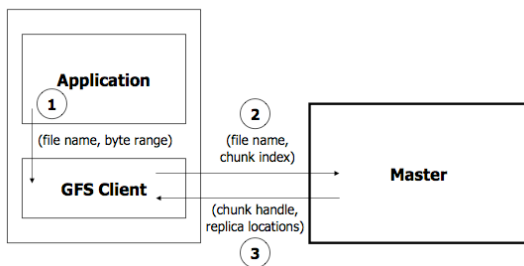


Figure 2: Read Algorithm (explains step 1, 2, 3)

4. Client picks a location (chunkserver #7 in this case) and sends the (chunk handle, byte range) request to the location
5. Chunkserver sends requested data (2048 bytes of data) to the client
6. Client forwards the data to the application

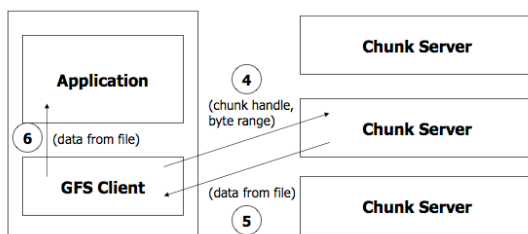


Figure 3: Read Algorithm (explains step 4, 5, 6)

From the algorithm and the figures, it should be very clear to readers on how the read operation in GFS works. We will try to give another concrete example to give a clearer view.

### Example of Read:

1. Indexer originates the read request, (crawl\_99, 2048 bytes)
2. GFS client translates the request form (filename, byte range) -> (filename, chunk index), and sends it to master, e.g., (crawl\_99, 2048 bytes) -> (crawl\_99, index: 3)
3. Master responds with chunk handle (ch\_1003) and replica locations (i.e. chunkservers where the replicas are stored), {chunkservers: 4, 7, 9}

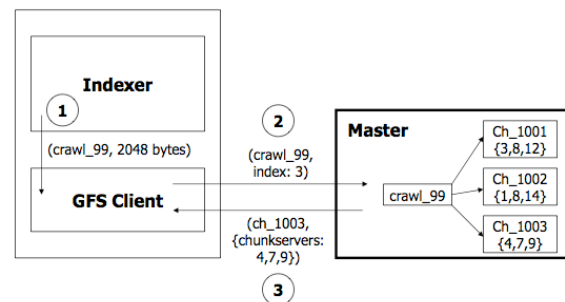


Figure 4: Read Algorithm example (explains step 1, 2, 3)

4. Client picks a location (chunkserver #7 in this case) and sends the (chunk handle, byte range) request to the location
5. Chunkserver sends requested data (2048 bytes of data) to the client
6. Client forwards the data to the application (2048 bytes of data)

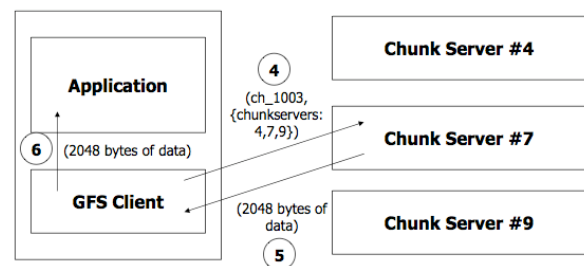


Figure 5: Read Algorithm example (explains step 4, 5, 6)

## Write Algorithm

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.

### Algorithm for Write:

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

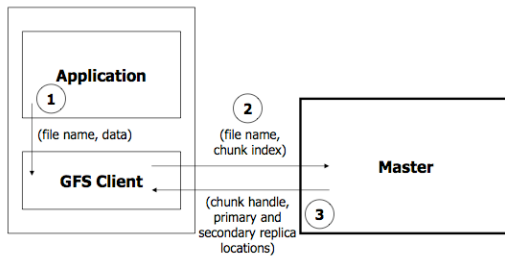


Figure 6: Write Algorithm (explains step 1, 2, 3)

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

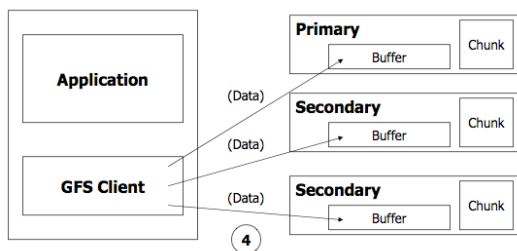


Figure 6: Write Algorithm (explains step 4)

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

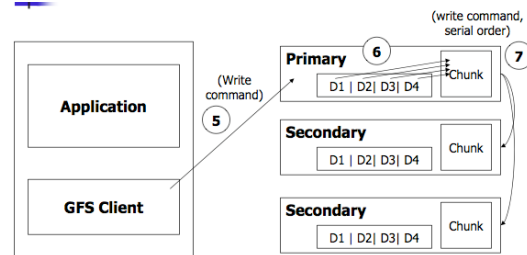


Figure 7: Write Algorithm (explains step 5, 6, 7)

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

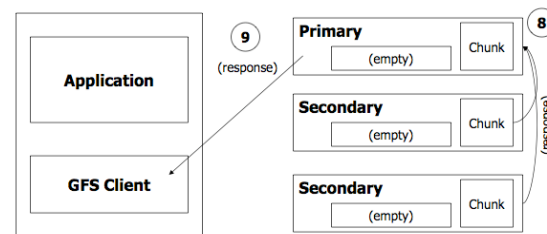


Figure 8: Write Algorithm (explains step 8, 9)

## Record Append:

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 very similar to Write algorithm, following is the algorithm for it.

**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 locations
5. Primary checks if record fits in specified chunk
6. If record doesn't fit, then the primary:
  - Pads the chunk
  - Tells 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 do the same
  - Receives responses from secondaries
  - And sends final response to the client

## Fault Tolerance

We have been claiming about better Fault Tolerance system in GFS from the beginning; in this section, we will show few examples of what GFS does for fault tolerance.

**Fast Recovery:** *master* and *chunkservers* are designed to be able to restart and restore state in few seconds. Recovery is also done based on the priority of the situation. We will see an example in the Performance section that how it restores when one replica is down against when two replicas are down, which is very urgent situation, as we have only one copy left!

**Chunk Replication:** It has *chunk* replicas across multiple machines, across multiple racks. So it recovers the *chunk* very easily when one replica goes down!

**Master Mechanisms:** Master does the following to restore without losing any data and without interrupting any operations:

- Keep log of all changes made to metadata
- Periodic checkpoints of the log
- Log and checkpoints replicated on multiple machines
- Master state is replicated on multiple machines
- Shadow master for reading data if real master is down

In this section we saw what measures GFS takes for fault tolerance system in their file system. In the next section we will see the performance of GFS.

## Performance

In this section of GFS evaluation, we will present some performance results to illustrate the bottlenecks inherent in the GFS architecture and implementation, and also will present some numbers from real clusters in use at Google.

### Performance (Test Cluster)

They set up a test cluster with 1 *master*, 16 *chunkservers* and 16 *clients*. This configuration was set-up for ease in testing. Typically there are hundreds of *chunkservers* and hundreds of *clients* in a cluster.

All the machines are configured with dual 1.4 GHz PIII processors, 2 GB of memory, two 80GB 5400 rpm disks, and a 100 Mbps full-duplex Ethernet connection to an HP 2524 switch. All 16 GFS server machines are connected to one switch, and all 16 clients are connected to the other. The two switches are connected with a 1 Gbps link.

### Read

*N* clients read simultaneously from the file system. Each *client* read a randomly selected 4 MB region from a 320 GB file set. This is

repeated 256 times, so that each *client* reads 1 GB of data. Total memories of *chunkservers* are 32 GB (2 GB each for 16 *chunkservers*), so they expect at most 10% hit rate in the Linux buffer cache. Their result should be close to cold cache results.

Figure 9 shows the aggregate read rate for  $N$  clients and its theoretical limit. The limit peaks at an aggregate of 125MB/s when the 1Gbps link between two switches get saturated, or 12.5 MB/s per client when its 100 Mbps network interface get saturate, whichever applies.

The observed read rate for one client is 10 MB/s, which is 80% of the network limit. For 16 clients the rate is 6 MB/s per client, which is 75% of the limit. It is understandable that with the increase of clients, the load increases, so the read rate decreases.

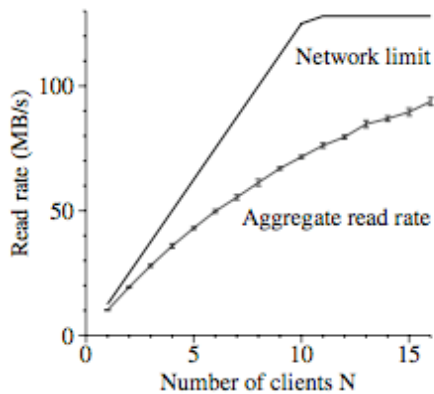


Figure 9: Read Performance

## Write

$N$  clients write simultaneously to  $N$  distinct files. Each client writes 1 GB of data to a new file in the series of 1 MB writes. The aggregate write rate and its limit are shown in the Figure 10. For each bit, it has to write in at least 3 replicas. So the write rate is lower than the read rate.

The observed write rate for one client is 6.3 MB/s, where the maximum limit is 12.5 MB/s. This rate decreases to 2.2 MB/s per client for 16 clients, making the aggregate write rate 35 MB/s.

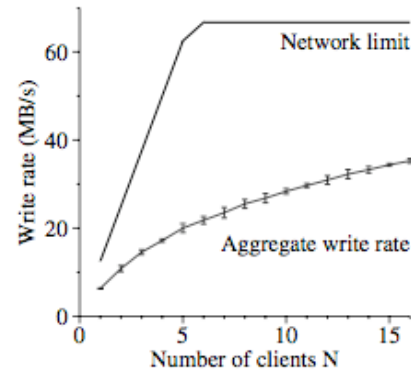


Figure 10: Write Performance

## Record Append

Figure 11 shows the record append performance.  $N$  clients append simultaneously to a single file. Performance is limited by the network bandwidth of the *chunkservers* that store the last chunk of the file, independent of the number of clients. For one client it record appends at 6 MB/s and, mostly due to congestion and variances in network transfer rate seen by different clients, it drops to 4.8 MB/s per client for 16 clients.

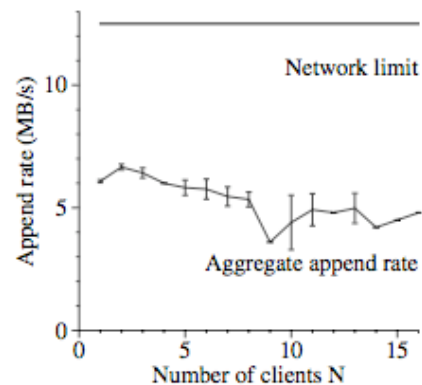


Figure 11: Record Append Performance

## Performance (Real-world cluster)

After the evaluation of one set-up cluster (explained in previous section), in this section, we will show the results of another evaluation in the real world cluster. In this evaluation they used two clusters. Following is the characteristics of both clusters and the Table 1 has the properties of these clusters.

Cluster A:

- Used for research and development
- Used by more than 100 engineers
- Typical task initiated by user and runs for a few hours
- Task reads MBs - TBs of data, transforms / analyses the data, and writes results back

Cluster B:

- Used for production data processing
- Typical task runs much longer than a Cluster A task
- Continuously generates and processes multi-TB data sets
- Human users rarely involved

Cluster	A	B
Chunkservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	155 TB
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	48 MB	60 MB

Table 1: Properties of Cluster A and B<sup>6</sup>

Few points are worth noting, before moving into the performance on these clusters. There were many *chunkservers* at each clusters (342, 227);

<sup>6</sup> Dead files: files that were deleted or replaced by a new version but whose storage have not been reclaimed.

on average, cluster B file size is triple of cluster A file size; Metadata at master is small (48, 60 MB), which helps the master to recover from crash within seconds.

In Table 2, we will see the performance of these two clusters on different operations.

Cluster	A	B
Read rate (last minute)	583 MB/s	380 MB/s
Read rate (last hour)	562 MB/s	384 MB/s
Read rate (since restart)	589 MB/s	49 MB/s
Write rate (last minute)	1 MB/s	101 MB/s
Write rate (last hour)	2 MB/s	117 MB/s
Write rate (since restart)	25 MB/s	13 MB/s
Master ops (last minute)	325 Ops/s	533 Ops/s
Master ops (last hour)	381 Ops/s	518 Ops/s
Master ops (since restart)	202 Ops/s	347 Ops/s

Table 2: Performance (Real-world cluster)

We will summarize the results from the performance table as:

- It had many more reads than writes
- Both clusters were in the middle of heavy read activity
- Cluster B was in the middle of a burst of write activity
- In both clusters, master was receiving 200-500 operations per second, and master can handle it easily, which means master is not a bottleneck

## Experiment in Recovery Time

In this final section of Performance, we will see how the system reacts when any chunkserver goes down!

In the first experiment, one chunkserver in cluster B was killed. That chunkserver had 15K chunks containing 600 GB of data. All chunks were successfully restored in 23.2 minutes, at an effective replication rate of 440 MB/s.

In the next experiment, two *chunkservers* in cluster B were killed. Each was with roughly 16K chunks and 600 GB of data, so in this situation they had only one replica, which means the situation is very urgent! The system successfully restored in 2 minutes! This says how fast the system reacts when it needs to do so.

## Summary and Conclusion

In this survey paper we surveyed about Google File System, GFS. We started by explaining Google applications and operations to understand the motivation behind a new distributed file system. After that discussed about different concepts of GFS architecture, e.g. *chunk*, *metadata*, *master*, communication between *master* and *chunkservers*, read, write and record append operations. We then showed how fault tolerant GFS is and also showed some evaluation results.

GFS clearly demonstrates the qualities essential for the large-scale processing with commodity class PCs. It serves the Google applications a solid backbone that helps these applications to stand ahead of its competitors and serving millions of people every single second! They designed GFS not only for their current applications, but also for anticipated applications too. This proved to be successful and helping Google in implementing new large-scale data intensive efficient applications everyday and adding these successful applications in the Google's application list<sup>7</sup>!

## References:

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

[Brin and Page 1998] Sergey Brin and Lawrence Page The Anatomy of a Large-Scale Hypertextual Web Search Engine, Computer Networks and ISDN Systems, 1998. Available online at  
<<http://infolab.stanford.edu/~backrub/google.html>>

[Page et al 98] Lawrence Page, Sergey Brin, Rajeev Motwani, Terry Winograd. The

PageRank Citation Ranking: Bringing Order to the Web.

[Google-Technology] Google Technology,  
<<http://www.google.com/technology/>>

[wiki-filessystem] Wikipedia page of List of file systems  
[http://en.wikipedia.org/wiki/List\\_of\\_file\\_system\\_s#Distributed\\_file\\_systems](http://en.wikipedia.org/wiki/List_of_file_system_s#Distributed_file_systems)

[wiki-AFS] Wikipedia page of Andrew File System,  
<[http://en.wikipedia.org/wiki/Andrew\\_File\\_System](http://en.wikipedia.org/wiki/Andrew_File_System)>

[Coda] Coda official Homepage  
<<http://coda.cs.cmu.edu/>>

[wiki-Coda] Wikipedia page of Coda File System,  
<[http://en.wikipedia.org/wiki/Coda\\_%28file\\_system%29](http://en.wikipedia.org/wiki/Coda_%28file_system%29)>

---

<sup>7</sup> <http://labs.google.com/>