# MIT 6.824: Lecture 3 - GFS

*02 May 2020* · 10 min read

The Google File System paper is relevant to this course because GFS is an example of *distributed storage*, which is a key abstraction in building distributed systems. Many distributed systems are either distributed storage systems or systems built on top of distributed storage.

Building distributed storage is a hard problem for a couple of reasons:

- These systems are built to get a *high performance* when the volume of data is too large for a single machine, which leads us to *sharding (splitting)* the data over multiple servers.
- Because multiple servers are involved, there will likely be more *faults* in the system.
- To improve fault tolerance, the data is usually *replicated* across multiple machines.
- Replication of data leads to potential *inconsistencies* in the data. A client could read from a stale replica, for example.
- The protocols for better consistency often lead to a *lower performance*, which is the opposite of what we want.

This cycle, which leads back to performance, highlights the challenges in building distributed systems. The GFS paper touches on these topics and discusses the trade-offs that were made to yield good performance in a production-ready distributed storage system.

**Table of Contents**

## Paper Summary

The system was built at a time when Google needed a system to meet its data-processing demands with the goals of achieving good performance while being:

- *Global:* Not tailored for just one application, but available to many Google applications.
- *Fault Tolerant* : Designed to account for component failures by default.
- *Scalable:* It could expand to meet increasing storage needs by adding extra servers.

Also note that it was tailored for a workload that largely consisted of sequential access to huge files (read or append). It was not built to optimize for low-latency requests; rather, it was meant for batch workloads which often read a file sequentially, as in MapReduce jobs.

## Design Overview

A GFS Cluster is made up of a *single master* and multiple *chunkservers*, and is accessed by multiple clients, as shown in the figure below.

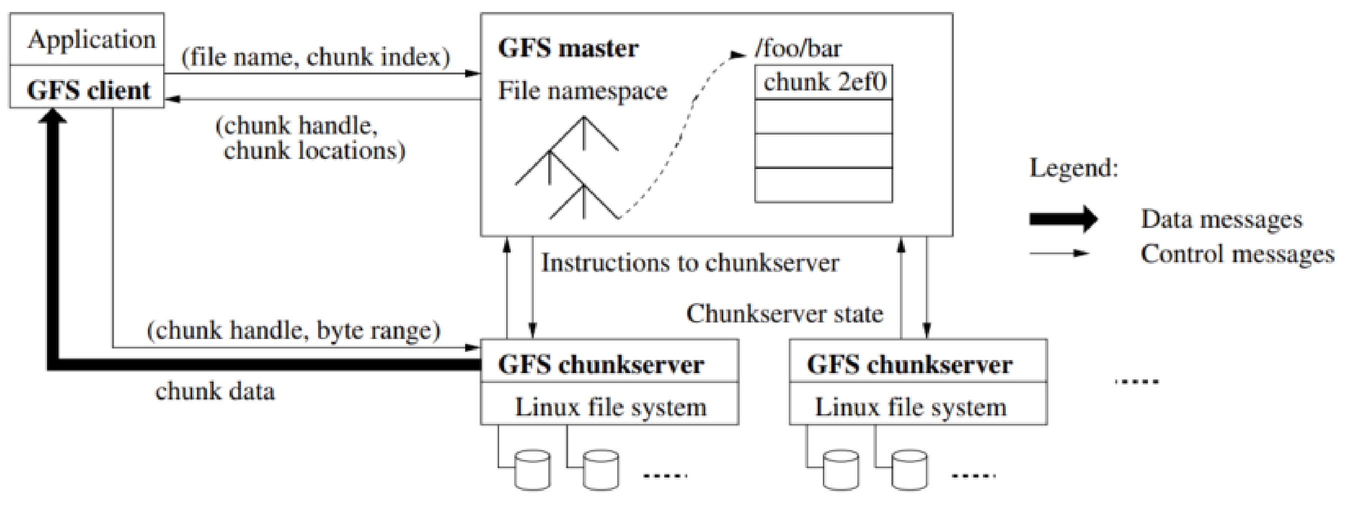


**Figure 1: GFS Architecture**

Breakdown of the architecture:

- A client and a chunkserver may reside on the same machine, provided it has enough resources.
- A file stored in GFS is broken into *chunks*. A chunk is identified by an immutable and globally unique *chunk handle.*
- A chunk can be replicated across multiple chunkservers. It is configured to be replicated across three chunkservers by default.
- The master keeps track of all the filesystem metadata. It knows how a file is split into chunks, and keeps track of what chunkservers hold each chunk.

  It is also responsible for garbage collection of orphaned chunks (when a file is deleted), and the migration of chunks between chunk servers for rebalancing load.
- The master communicates with each chunkserver via *Heartbeat* operations to pass instructions to it and collects its state.
- There is a *GFS Client library* that is linked into each application using GFS. The library handles communication with the master and chunkservers to read and write data on behalf of the application.

- The chunkservers do not perform any form of caching but instead rely on the Linux buffer cache which keeps frequently accessed data in memory.

An interesting design choice made in this system is the decoupling of the data flow from the control flow. The GFS client only communicates with the master for metadata operations, but all data-bearing communications (reads and writes) go directly to the chunkservers. I'll explain how that works next.

**Single Master**

As noted earlier, there is a single master in a GFS cluster which clients only interact with to retrieve metadata. This section highlights the role of the master in decoupling the data flow from the control flow.

To read the data for a file:

- The client first communicates with the master, sending it a request containing the file name and the *chunk index*. The client derives the chunk index from a combination of the file name and the byte offset the application wants to read from.

- The master replies with the corresponding chunk handle and the location of its replicas.

- The client caches this information using the file name and chunk index as the key.

- The client then sends a request to one of the replicas specified by the client, usually the one closest to it, specifying the chunk handle and the byte range for the requested data.

- By caching the information from the master, further reads to the same chunk do not require any more client-master interactions until the cached information expires or the file is reopened.

**Chunk Size**

In typical Linux filesystems, a file is split into blocks and those blocks usually range from 0.5-65 kilobytes in size, with the default on most file systems being 4 kilobytes.

A block size is the unit of work for the file system, which means reading or writing any files is done in multiples of that block size.

In GFS, chunks are analogous to blocks, except that chunks are of a much larger size (64 MB). Having a large chunk size offers several advantages in this system:

- The client will not need to interact with the master as much, since reads and writes on the same chunk will require only one initial request to the master to get the chunk location information, and more data will fit on a single chunk.

- With large chunk sizes, a client is more likely to perform many operations on a given chunk, and so we can reduce network overhead by keeping a persistent TCP connection to the chunkserver over an extended period of time

- Third, it can reduce the size of the metadata stored on the master, since it's keeping track of fewer chunks.

Google uses lazy space allocation to avoid wasting space due to internal fragmentation. Internal fragmentation means having unused portions of the 64 MB chunk. For example, if we allocate a 64 MB chunk and only fill up 10 MB, that's a lot of unused space.

According to this Stack Overflow answer,

> Lazy space allocation means that the physical allocation of space is delayed as long as possible, until data at the size of the chunk size is accumulated.

From the rest of that answer, I think what this means is that the decision to allocate a new chunk is based *solely* on the data available, as opposed to using another partitioning scheme to allocate data to chunks.

This does not mean the chunks will always be filled up. A chunk which contains the file region for the end of a file will typically only be partially filled up.

**Metadata**

The master stores three types of metadata in memory:

- File and chunk namespaces (i.e. directory hierarchy.)
- The mapping from files to chunks.
- The location of each chunk's replica.

The first two types listed are also persisted on the master's local disk. The third is not persisted; instead, the master asks each chunkserver about its chunks at master startup and when a chunkserver joins the cluster.

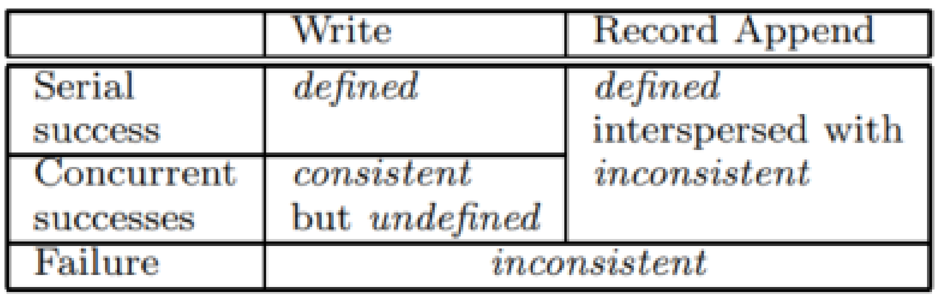
By having the chunkserver as the ultimate source of truth of each chunk's location, GFS eliminates some of the challenges of keeping the master and chunkservers in sync regularly.

The master keeps an operation log, where it stores the namespace and file-to-chunk mappings on local disk. It replicates this operation log on several machines, and GFS does not make changes to the metadata visible to clients until they have been persisted on all replicas.

After startup, the master can restore its file system state by replaying the operation log. It keeps this log small to minimize the startup time by periodically checkpointing it.

**Consistency Model**

The consistency guarantee for GFS is relaxed. It does not guarantee that all the replicas of a chunk are *byte-wise identical*. What it does guarantee is that every piece of data stored will be written *at least once* on each replica. This means that a replica may contain duplicates, and it is up to the application to deal with such anomalies.



| | Write | Record Append |
|---|---|---|
| Serial success | *defined* | *defined* interspersed with *inconsistent* |
| Concurrent successes | *consistent* but *undefined* | |
| Failure | *inconsistent* | |

**Table 1: File Region State After Mutation**

From Table 1 above:

- A file region is *consistent* when all the clients will see the same data for it, regardless of which replica they read from.
- After a file data mutation, a region is *defined* if it is consistent and all the clients will see the effect of the mutation in its entirety.

The data mutations here may be *writes* or *record appends.* A write occurs when data is written at a file offset specified by the application.

#### Record Appends

Record appends cause data to be written *atomically at least once* even in the presence of concurrent mutations, but at an offset chosen by GFS.

If a record append succeeds on some replicas and fails on others, those successful appends are not rolled back. This means that if the client retries the operation, the successful replicas may have duplicates for that record.

Retrying the record append at a new file offset could mean that the offset chosen for the initial failed append operation is now blank in the file regions of the failed replicas; that is, if the region has not been modified before the retry. This blank region is known as a *padding*, and the existence of padding and duplicates in replicas are what make them inconsistent.

Applications that use GFS are left with the responsiblilty of dealing with these inconsistent file regions. These applications can include a unique ID with each record to filter out duplicates, and use checksums to detect and discard extra padding.

There is also the possibility of a client reading from a stale replica. Each chunk replica is given a version number that gets increased for each successful mutation. If the chunkserver hosting a chunk replica is down during a mutation, the chunk replica will become stale and will have an older version number. Stale replicas are not given to clients when they ask the master for the location of a chunk, and they are not involved in mutations either.

Despite this, because a client caches the location of a chunk, it may read from a stale replica before the information is refreshed. The impact of this is low because most operations to a chunk are append-only. This means that a stale replica usually returns a premature end of chunk, rather than outdated data for a value.

## System Interactions

This section describes in more detail how the client, master and chunkservers interact to implement data mutations and atomic record appends.

**Writes**

When the master receives a modification operation for a particular chunk, the following happen:

a) The master finds the chunkservers which hold that chunk and grants a *chunk lease* to one of them.

- The server with the lease is called the *primary,* while the others are *secondaries.*
- The primary determines the order in which mutations are applied to all the replicas.

b) After the lease expires (typically after 60 seconds), the master is free to grant primary status to a different server for that chunk.

- The master may sometimes try to revoke a lease before it expires (e.g. to prevent the mutation of a file when the file is being renamed. )
- The primary may also request an indefinite extension of the lease as long as the chunk is still being modified.

c) The master may lose communication with a primary while the mutation is still happening. If this happens, it is fine for the master to grant a new lease to another replica as long as the lease timeout has expired.

Let's look at Figure 2 which illustrates the control flow of a write operation.

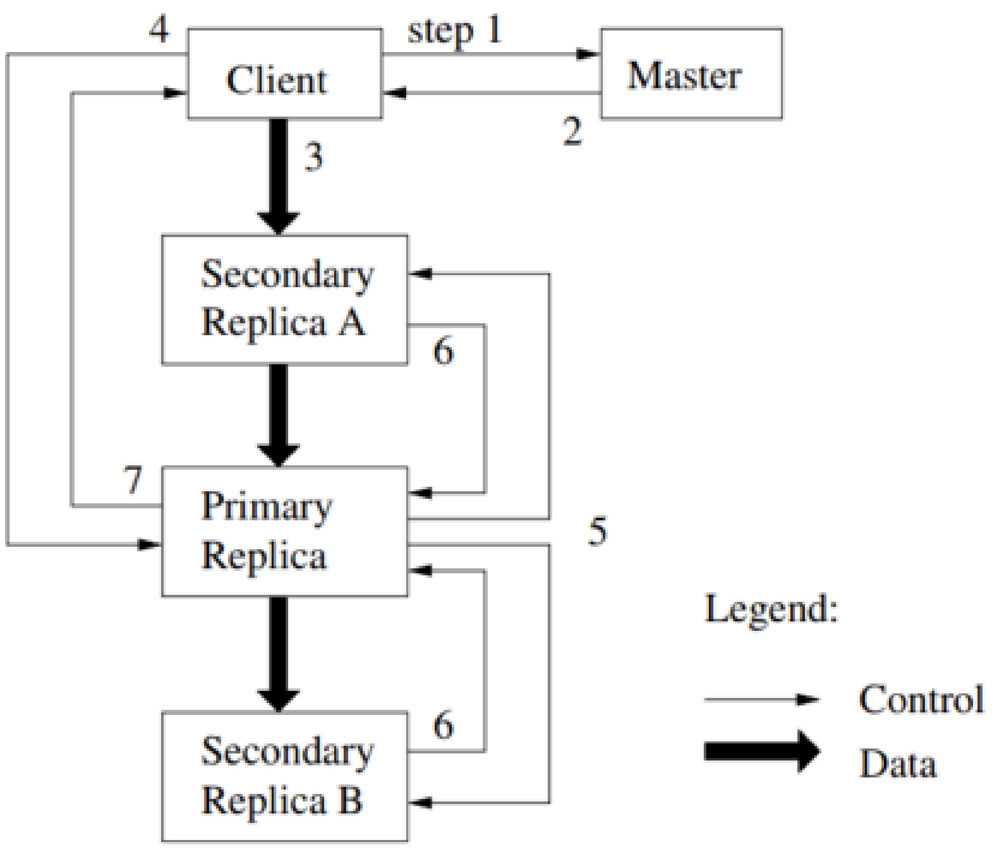


**Figure 2: Write Control and Data Flow**

The numbered steps below correspond to each number in the diagram.

1. The client asks the master for all chunkservers.

2. The master grants a new lease to a replica (if none exist), increases the chunk version number, and tells all replicas to do the same after the mutation has

been applied. It then replies to the client. After this, *the client no longer has to talk to the master*.

3. The client pushes the data to all the chunkservers, not necessarily to the primary first. The servers will initially store this data in an internal LRU buffer cache until the data is used.
4. Once the client receives the acknowledgement that this data has been pushed successfully, it sends the write request to the primary chunkserver. The primary decides what serial order to apply the mutations in and applies them to the chunk.
5. After applying the mutations, the primary forwards the write request and the serial number order to all the secondaries for them to apply in the same order.
6. All secondaries reply to the primary once they have completed the operation.
7. The primary replies to the client, indicating whether the operation was a success or an error. Note:
    - If the write succeeds at the primary but fails at any of the secondaries, we'll have an inconsistent state and an error is returned to the client.
    - The client can retry steps 3 through 7.

**Atomic Record Appends**

The system interactions for record appends are largely the same as discussed for writes, with the following exceptions:

- In step 4, the primary first checks to see if appending the record to the current chunk would exceed the maximum size of 64MB. If so, the primary pads the chunk, notifies the secondaries to the same, and then tells the client to retry the request on the next chunk.
- If the record append fails on any of the replicas, the client must retry the operation. As discussed in the Consistency section, this means that replicas of the same chunk may contain duplicates.
- A record append is successful only when the data has been written *at the same offset* on all the replicas of a chunk.

## Fault Tolerance

Fault Tolerance is achieved in GFS by implementing:

- *Fast Recovery* - The master and the chunkservers are designed to restore their state and start in a matter of seconds.
- *Chunk Replication:* Each chunk is replicated on multiple chunkservers on different racks. This ensures that some replicas are still available even if a rack is destroyed. The master is able to clone existing replicas as needed when chunkservers go offline or a replica is detected as stale or corrupted.
- *Master Replication:* The master is also replicated for reliability. A state mutation is considered committed only when the operation log has been flushed to disk on all master replicas. When the master fails, it can restart almost immediately.

In addition, there are *shadow masters* which provide read-only access to the filesystem when the file is down. There may be a lag in replicating data from the primary master to its shadows, but these shadow masters help to improve availability.

### Data Integrity

Checksumming is used by each chunkserver to detect the corruption of stored data.

From the course website [1]:

> A checksum algorithm takes a block of bytes as input and returns a single number that's a function of all the input bytes. For example, a simple checksum might be the sum of all the bytes in the input (mod some big number). GFS stores the checksum of each chunk as well as the chunk.
>
> When a chunkserver writes a chunk on its disk, it first computes the checksum of the new chunk, and saves the checksum on disk as well as the chunk. When a chunkserver reads a chunk from disk, it also reads the previously-saved checksum, re-computes a checksum from the chunk read from disk, and checks that the two checksums match.
>
> If the data was corrupted by the disk, the checksums won't match, and the chunkserver will know to return an error. Separately, some GFS applications stored their own checksums, over application-defined records, inside GFS files, to distinguish between correct records and padding. CRC32 is an example of a checksum algorithm.

[1] GFS FAQ - Lecture Notes from MIT 6.824

## Conclusion

This week's material brought some interesting ideas in the design of a distributed storage system. These include:

- The decoupling of data flow from control flow.
- Using large chunk sizes to reduce overhead.
- The sequencing of writes through a primary replica.

However, having a single master eventually became less than ideal for Google's use case. As the number of files stored increased by thousands, it became harder to fit all the metadata for those files on the master. In addition, the number of clients also increased, leading to too much CPU load on the master.

Another challenge with GFS at Google was that the weak consistency model meant applications had to be designed to cope with those limitations. These limitations led to the creation of Colossus as a successor to GFS.

## Further Reading

- Lecture 3: GFS - MIT 6.824 Lecture Notes.
- The Google File System by Firas Abuzaid (Stanford Lecture Notes)
- GFS: The Google File System by Brad Karp (UCL Lecture Notes)

- GFS: Evolution on Fast-forward - 2009 Interview with a Google engineer about the origin and evolution of the Google File System.

mit-6.824 distributed-systems learning-diary

## A small favour

Did you find anything I wrote confusing, outdated, or incorrect? Please let me know by writing a few words below.

Your name

Your email address

What should I know?

Send Message

## Follow along

To get notified when I write something new, you can subscribe to the RSS feed or enter your email below.

Your email address Subscribe