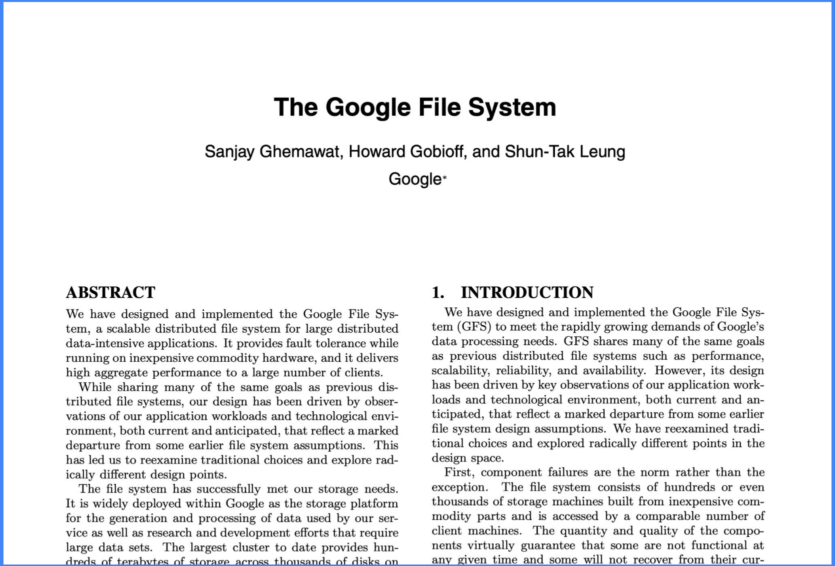

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google*

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their cur-

fault tolerant frens

Google File System

@MaanavKhaitan, @liamzebedee

Overview

Google File System is a **horizontally scalable** distributed file system designed by Google in the early 2000’s.

Readings

https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

https://static.googleusercontent.com/media/research.google.com/en//people/jeff/Stanford-DL-Nov-2010.pdf

Motivation: Why GFS?

- **Context**: building Google in early 2000's. 20PB (by today's standards) web crawl dataset. Can't store on a single drive. Buy a bunch of 1TB disks and wire them up as a storage pool.

- **Challenge**: how to build something resembling a file system from this?

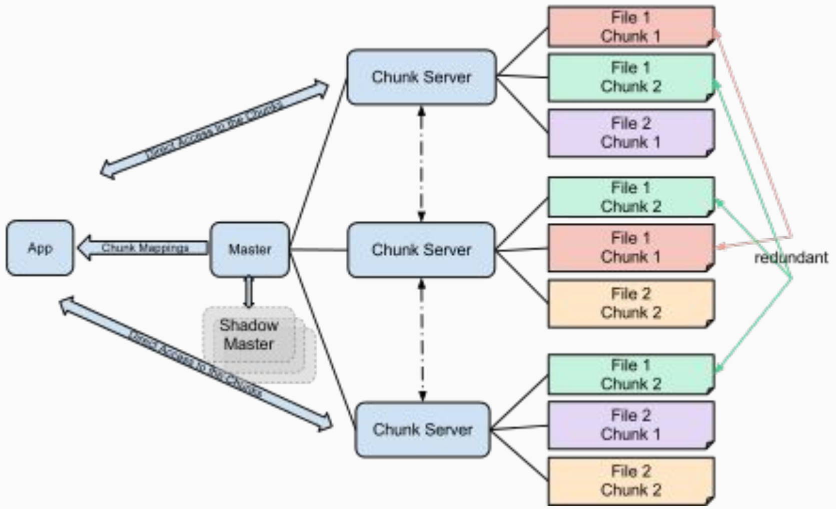
- **Objectives:**
    - Designed for streaming reads, not random access.
    - Designed to deal with inexpensive hardware which fails often.
    - Designed to have sharding.

Assumptions

- System stores modest number of large files.
  - Few million files, each typically 100 MB or larger in size.
- Workloads primarily consist of:
  - **Large streaming reads**, individual operations typically read hundreds of KBs, more commonly 1 MB or more.
  - **Large, sequential writes** that append data to files.
  - **Small random reads** typically reads a few KBs at some arbitrary offset.
- **Concurrent-safe.** The system must efficiently implement well-defined semantics for multiple clients that concurrently append to the same file.
- **Bandwidth > Latency.** High sustained bandwidth is more important than low latency.

How does GFS work?

- Each file is divided into fixed-size “chunks”
- These chunks are distributed across a bunch of chunkservers
- There is **one** master which stores mappings from:
    - (file → chunk IDs)
    - (chunk ID → chunkservers which store this chunk)



Reading data

1. Client sends request to master
2. Master sends list of chunkservers to client
3. Client sends request to receive data from chunkservers
4. Chunkservers return data back to client

Writing data

Each chunk is assigned one chunkserver as primary.
The primary chunkserver has a short-lived (60s) lease with the master.
The primary decides writes.

1. Client connects to master, looks up all chunkservers for a chunk.
2. Client pushes data to chunkservers.
3. Client sends WRITE to primary chunkserver.
4. Primary sends WRITE commands to all secondaries.
5. After all secondaries ACK, the primary applies the write and returns success to client.
    a. *If even a single secondary fails*, the primary responds with ERROR to the client

Serialisation

If this was a blockchain, what is the ordering of writes for each chunk?

the **global mutation order** is defined:

1. first by the lease grant order chosen by the master, and
2. within a lease by the serial numbers assigned by the primary.

eg.

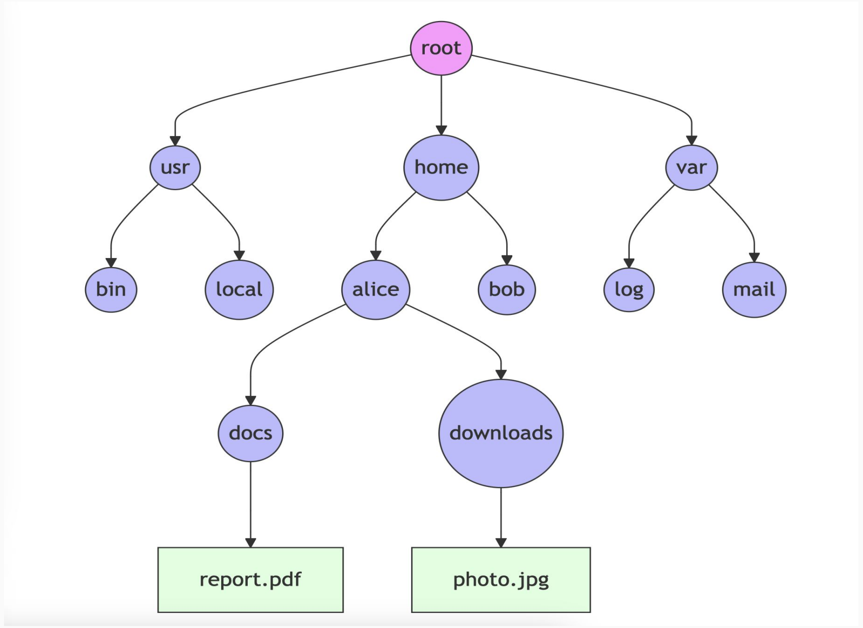
(lease-1, serial-number=4)
(lease-1, serial-number=5)
(lease-1, serial-number=6)
(lease-2, serial-number=7)
(lease-2, serial-number=8)

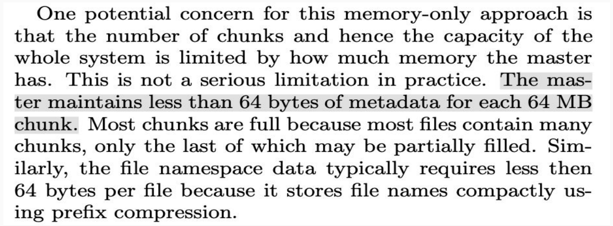
Core ideas

- 1 master, **1000s of workers**.
- Entire metadata (file system prefix trie) **fits in RAM**.
- **Chunk replication**: Each chunk is stored on multiple chunkservers for redundancy.
- Separate **control flow from data flow**.
- Master **doesn’t store chunk location mapping on disk**; instead it queries the chunkservers when the master restarts.
- Master stores an **“operation log”** on disk (write-ahead log).
- **Application-defined replication** factor for files.
- **Relaxed** consistency model.
- **Smart** client library.

Core ideas

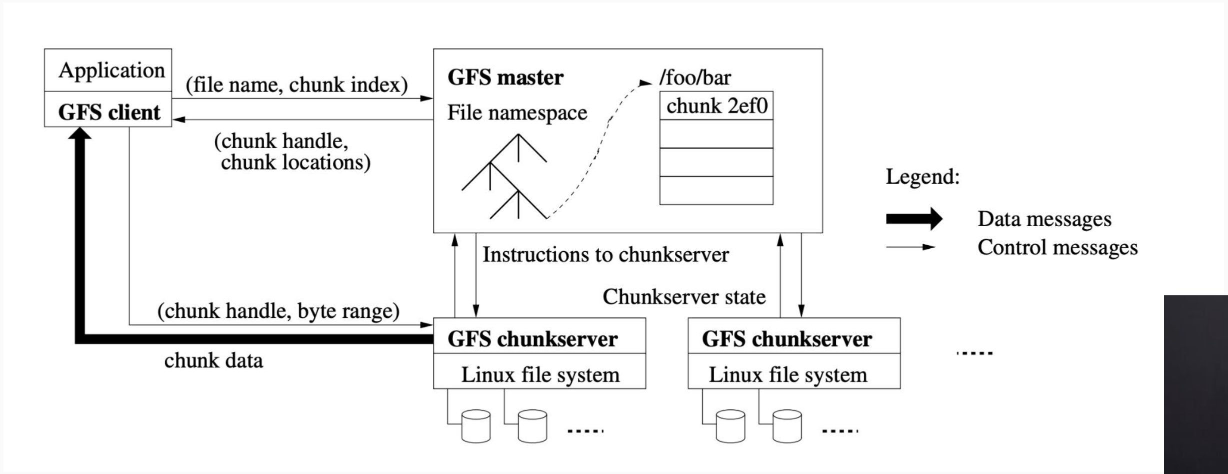
- 1 master, **1000s of workers**.
- Entire metadata (file system prefix trie) **fits in RAM**.




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.

Core ideas

- **Chunk replication**: Each chunk is stored on multiple chunkservers.



Core ideas

- Separate **control flow from data flow**.

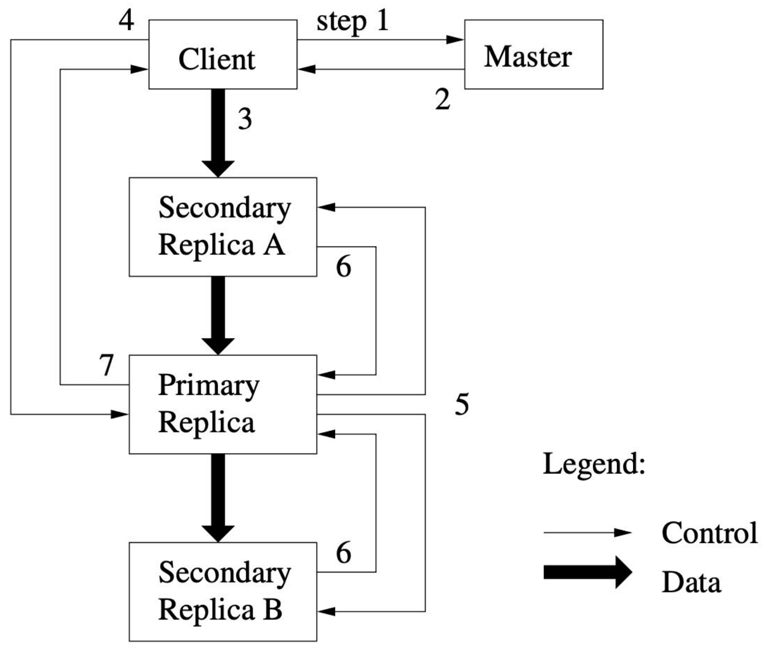
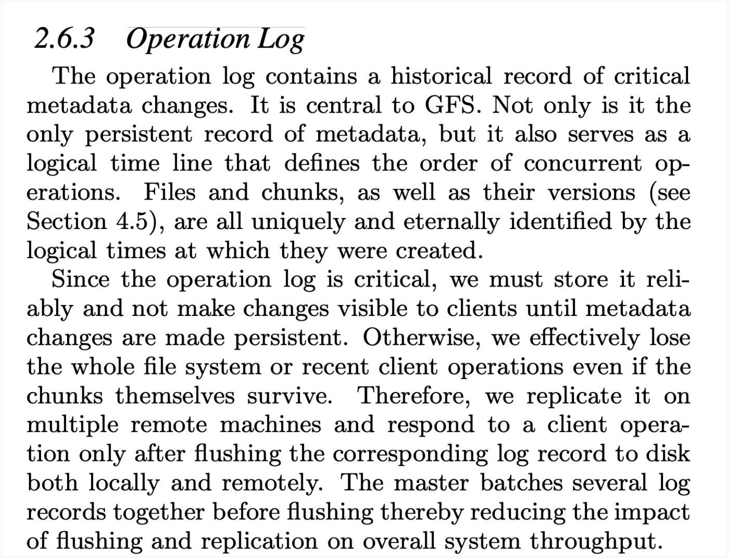


Figure 2: Write Control and Data Flow

Core ideas

- Master **doesn't store chunk location mapping on disk**; instead it queries the chunkservers when the master restarts.
- Master stores an **"operation log"** on disk (write-ahead log).
- Application-defined replication factor for files.



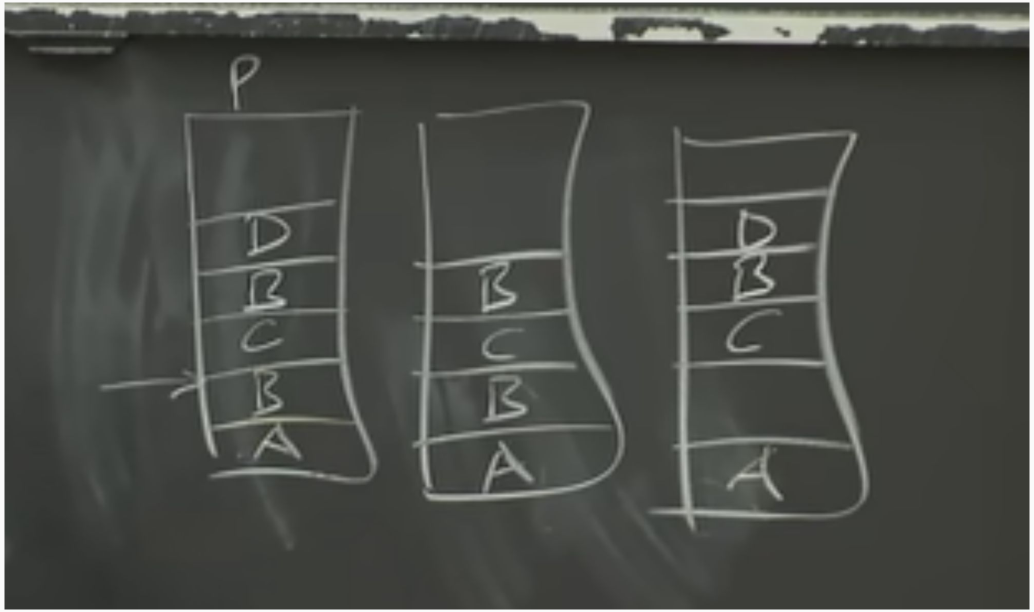
2.6.3 Operation Log

The operation log contains a historical record of critical metadata changes. It is central to GFS. Not only is it the only persistent record of metadata, but it also serves as a logical time line that defines the order of concurrent operations. Files and chunks, as well as their versions (see Section 4.5), are all uniquely and eternally identified by the logical times at which they were created.

Since the operation log is critical, we must store it reliably and not make changes visible to clients until metadata changes are made persistent. Otherwise, we effectively lose the whole file system or recent client operations even if the chunks themselves survive. Therefore, we replicate it on multiple remote machines and respond to a client operation only after flushing the corresponding log record to disk both locally and remotely. The master batches several log records together before flushing thereby reducing the impact of flushing and replication on overall system throughput.

Core ideas

- Relaxed consistency model.



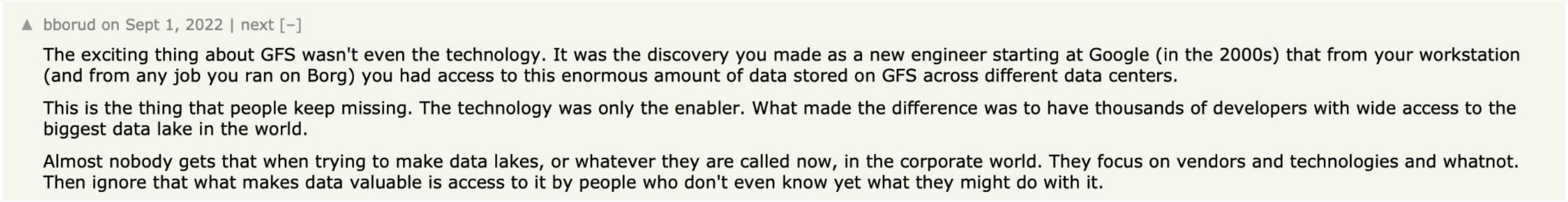
- What is the GFS consistency model?
  - A file region is **consistent** if all clients will always see the same data, regardless of which replicas they read from
  - A region is **defined** after a file data mutation if it is *consistent* and clients will see what the mutation writes in its entirety.
  - After a sequence of successful mutations, the mutated file region is guaranteed to be **defined** and contain the data written by the last mutation
  - Scenarios:
    - *Non-concurrent*. When a mutation succeeds without interference from concurrent writers, the affected region is defined (and by implication consistent): all clients will always see what the mutation has written.
    - *Concurrent*. Concurrent successful mutations leave the region undefined but consistent: all clients see the same data, but it may not reflect what any one mutation has written. Typically, it consists of mingled fragments from multiple mutations
    - *Failure*. A failed mutation makes the region in- consistent (hence also undefined): different clients may see different data at different times.
  - After a sequence of successful mutations, the mutated file region is guaranteed to be defined and contain the data written by the last mutation.
  - Failure modes:
    - GFS achieves this by (a) applying mutations to a chunk in the same order on all its replicas (Section 3.1), and (b) using chunk version numbers to detect any replica that has become stale because it has missed mu- tations while its chunkserver was down (Section 4.5). Stale replicas will never be involved in a mutation or given to clients asking the master for chunk locations

Core ideas

- **Smart** client library.
    - Client library isn't just API wrapper, actually does a lot of work behind the scenes.
        - Caches chunkserver locations.
        - Pushes data to each chunkserver before writing.
        - the master includes the chunk version number when it informs clients which chunkserver holds a lease on a chunk … the client verifies the version number when it performs the operation so that it is always accessing up-to-date data

Summary

Google File System is based


▲ bborud on Sept 1, 2022 | next [–]

The exciting thing about GFS wasn't even the technology. It was the discovery you made as a new engineer starting at Google (in the 2000s) that from your workstation (and from any job you ran on Borg) you had access to this enormous amount of data stored on GFS across different data centers.

This is the thing that people keep missing. The technology was only the enabler. What made the difference was to have thousands of developers with wide access to the biggest data lake in the world.

Almost nobody gets that when trying to make data lakes, or whatever they are called now, in the corporate world. They focus on vendors and technologies and whatnot. Then ignore that what makes data valuable is access to it by people who don't even know yet what they might do with it.

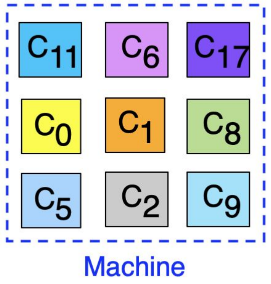
Extras

Core ideas

Pattern: Multiple Smaller Units per Machine

- Have each machine manage many smaller units of work/data
  - typical: ~10-100 units/machine
  - allows fine grained load balancing (shed or add one unit)
  - fast recovery from failure (N machines each pick up 1 unit)

- Examples:
  - map and reduce tasks, GFS chunks, Bigtable tablets, query serving system index shards


C_{11}
C_6
C_{17}
C_0
C_1
C_8
C_5
C_2
C_9
Machine