Distributed Systems

Christian J. Rudder

January 2025

Contents

This page is left intentionally blank.

Big thanks to **Prof. Anna Arpaci-Dusseau**
and **Prof. Ioannis Liagouris** for teaching CS351: Distributed Systems
at Boston University [1].

All illustration contain original assets.

Disclaimer: These notes are my personal understanding and interpretation of the course material. They are not officially endorsed by the instructor or the university. Please use them as a supplementary resource and refer to the official course materials for accurate information.

Prerequisites

This text assumes the reader has a basic understanding of computer science and programming. It will also assume they are somewhat familiar with computer architecture and operating systems at a high level. The text will review these concepts briefly for completeness, but it will not try to teach them from scratch or provide a full understanding of these topics.

The main focus will be on distributed systems, and will touch on:

- **Concurrency and Parallelism**
  - Concurrency, Parallelism, Threads
- **Consistency and Fault Tolerance**
  - Consistency, Fault-tolerance, Atomicity
- **Distributed Systems and Coordination**
  - Asynchrony, Coordination, Logical Time, Snapshots
- **Consensus Algorithms**
  - Raft, Paxos, Consensus
- **Replication and Data Management**
  - Replication, Sharding, Cluster
- **Protocols and Computing Models**
  - RPC, 2PC, Broadcast
- **Technologies and Tools**
  - MapReduce, Spanner, Dynamo, GFS, TLA+, Golang

0.1 Google File System (GFS)

Its 2003 and a research paper by Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, lay out what would become the Google File System (GFS). In the early 2000s, Google rapidly expanded its workload, scaling to support services like, web search, indexing, and data analytics. Traditional file systems were not suited for their needs. Hence, GFS was designed for the following:

Definition 1.1: Design Goals of GFS

Google File System (GFS) is a distributed file system, which aims to have:

- **Massive scalability:** Store vast amount of data across thousands of inexpensive commodity servers.
- **High availability:** Tolerant to frequent component failures **(replication needed)**.
- **High throughput:** Optimize for large data-streams of concurrent sequential reads and a majority of append-only writes.

Consistency Model: Weak consistency model for improved performance.

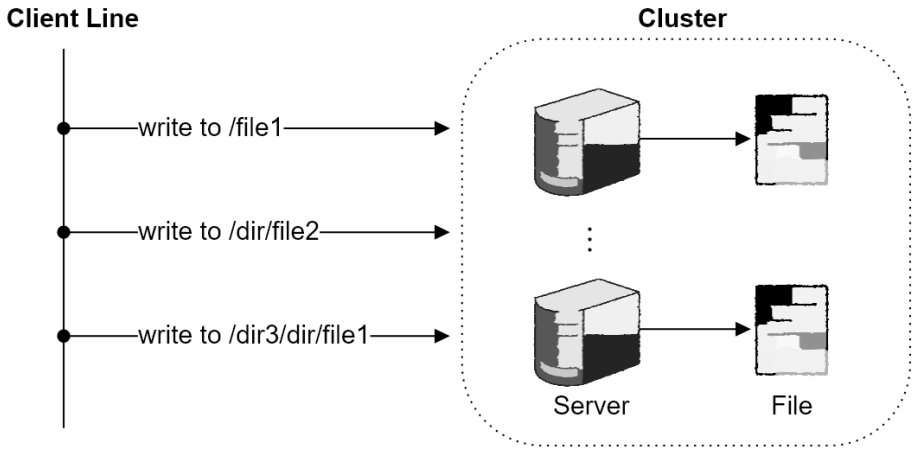

Figure 0.1: A high-level sketch of the Google File System (GFS) architecture.

Note: This is important as Google at the time was using cheap commodity hardware, which was prone to failure.

The following details how files are stored in GFS:

Definition 1.2: File Chunking

Files in GFS are divided into fixed-size chunks of 64MB. Each chunk is stored in a **chunk server**, identified by a unique 64 bit ID called a **chunk handle**.

To ensure fault tolerance, each chunk is **replicated** at least N times across different chunk servers (Typically $N = 3$).

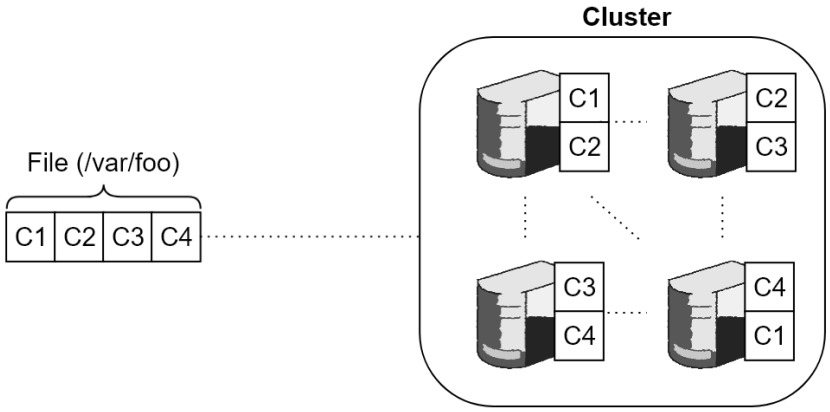

Figure 0.2: Simplified view of chunking in GFS, where a file is divided into 4 chunks (C1–C4), replicated twice across different servers.

We decide to store metadata on a dedicated server to act as our coordinator.

Definition 1.3: Metadata Management

The GFS **master** serves **one cluster**, storing the following metadata in memory:

- **File & Chunk Namespaces:** Hierarchical directory tree of files and the global chunk-ID namespace.
- **File→Chunk Mapping:** For each file, the ordered list of chunk handles.
- **Chunk Replica Locations:** Connecting chunk handles to their chunk servers holding its replicas via [IP:port].
- **Access Control Information:** File permissions and ownership attributes.

In particular, mappings are persisted via an operation log. Chunk locations are reconstructed by polling chunk servers at startup or when they join.

Now to discuss how client's read data from GFS:

Definition 1.4: Client Interaction

Clients interact with GFS via a two-step process:

- **Metadata Operations:** Clients query the master with the (file name, chunk index), receiving the chunk handle and its replica locations.
- **Data Operations:** The client caches such information and then directly communicate with the chunk servers.

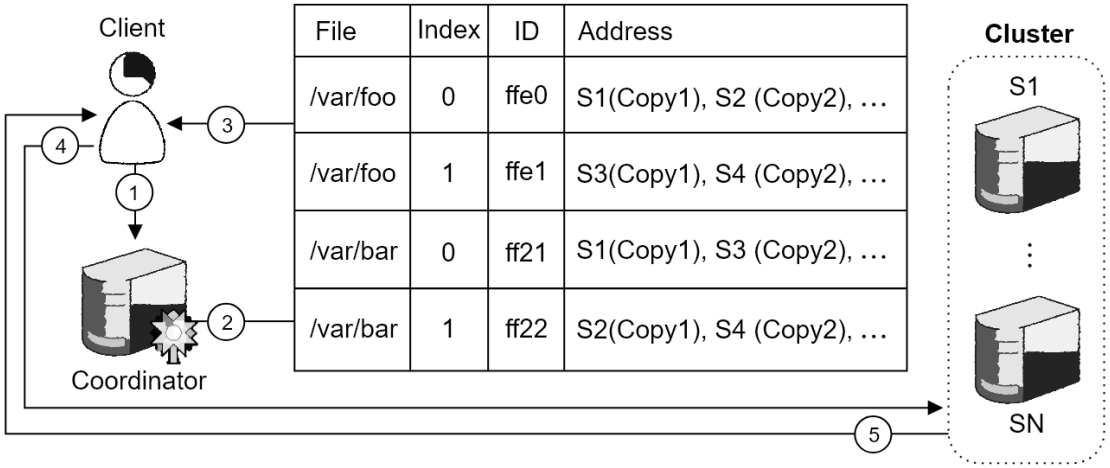

Figure 0.3: A Simplified view of client interaction with GFS. (1) Client sends a request to the master for metadata. (2) The master finds the chunk handle and its replica locations. (3) The master returns the chunk handle and replica locations to the client. (4) The client sends a request to the chunk server for data. (5) The chunk server returns the data to the client.

In our case we have specific needs that allow staleness to be tolerated:

Definition 1.5: GFS – Stale Reads are Allowed

Given the application of GFS's workload, it is acceptable to have stale reads. This greatly improves performance, as we don't have to have additional round trips to ensure the data is up to date.

For example, we may be gathering analytics which improve search results. Retrieving stale data doesn't break the system. When the latest data does arrive, we notice improved performance.

The client reads and writes data in two slightly different ways:

Definition 1.6: Read and Write Routines

Reads: The clients retrieves a server location from the master, reading directly from them.
Writes: Follow a three-step process:

- **Metadata Retrieval:** Clients query the master with the (file name, chunk index), receiving a list of replica locations. The master chooses one server to be the **primary replica**, granting it a **lease**, which defines the time period such server may act as the primary replica (typically 60s).
- **Data Transfer:** From the list of replicas, the client sends the data to the **closest replica** (not necessarily the primary replica). Such replica propagates the data to the next closest server, and so forth.
- **Commit Phase:** The client sends a **commit** request to the primary replica, which then notifies the other replicas to apply the changes. The primary replica then sends an **ACK** to the client.

In particular, the primary server follows a specific routine:

Definition 1.7: Primary Server Routine

The **primary** replica orchestrates concurrent writes with strict per-mutation ordering:

- **Lease Validation:** Upon each `WriteChunk` RPC, check that the primary's lease (granted by the master) is still valid. If it has expired, reject the request so the client can **refresh** metadata.
- **Replica-log ACKs:** The client waits for all replicas to ACK that the log has been replicated. Then it tells the primary to commit.
- **Per-Mutation Sequencing & Serialization:** Each incoming write is assigned a consecutive sequence number (possibly from multiple clients), which will be applied in sequential order.
- **Commit and Reply:** On command, the primary server applies the mutation in serial order, telling the secondaries to do the same; **However**, the primary sends the ACK to the client immediately even before any replicas even acknowledge the commit.

  This means **stale data** may be latent in the system. Though, our master will eventually catch this, and re-replicate the data.
- **Error Handling:** If any secondary fails to replicate the log in time (not the apply phase), abort the mutation, notify the client, and rely on the client's retry logic.

We illustrate the write process in the figure below:

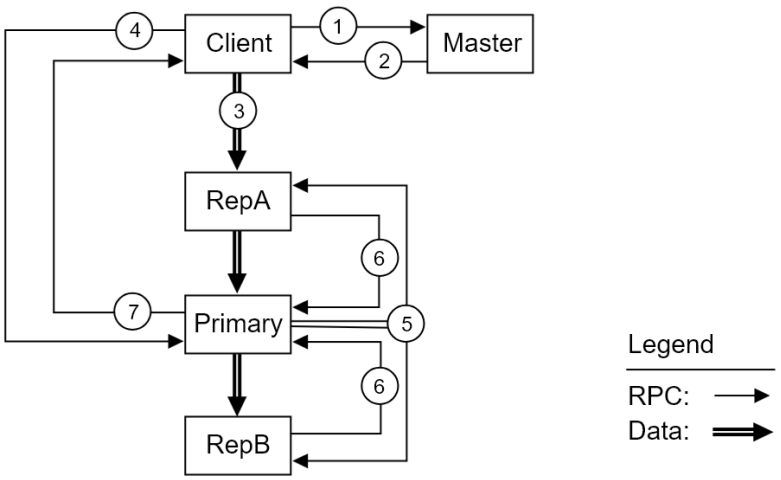

Figure 0.4: A Simplified view of the write process in GFS. (1) Client sends a request to the master for metadata. (2) The master finds the chunk handle and its replica locations. (3) The client sends a request to the closest replica who propagates it through the network. (4) After receiving ACKs from all replicas, the client sends a commit request to the primary replica. (5) The primary replica sends a commit request to all replicas. (6) The replicas send ACKs back to the primary replica. (7) The primary replica sends an ACK back to the client.

Now we need to ensure there are N copies of a chunk at all times:

Definition 1.8: Chunkserver Heartbeat Routine

Chunkservers send periodic **HeartBeat** RPCs to the master, conveying:

- **Held Chunk Handles & Versions:** The set of chunk IDs stored locally along with each chunk's current version, so the master can detect **missing or stale replicas**.
- **Lease Extension Requests:** Any primary replicas include lease-renewal requests for the chunks they lead.

The master's response piggy-backed on the heartbeat (reply to such heartbeat) may grant new leases or issue commands (e.g. initiate re-replication). If a chunk server's heartbeat is not received within a timeout, the master marks it dead and schedules re-replication to restore N live replicas per chunk.

The master handles persistent state and snapshots for log reduction in the following way:

Definition 1.9: Master Checkpoint & Operation-Log Routine

The **master** persists its metadata through two complementary mechanisms:

- **Operation Log:** An append-only record of every metadata-mutating request (e.g. file/directory creation, chunk allocation). Each entry is synchronously written to the master's local disk and replicated to remote machines.
- **Periodic Checkpoints:** In a background thread, the master periodically snapshots its entire in-memory state (file and chunk namespaces plus file→chunk mappings), writes the checkpoint to disk, and safely truncates the operation log up to that point.

On startup (or after a crash), the master loads the latest checkpoint and then replays any subsequent operations from the log to reconstruct its full metadata state.

In short, the master must keep record of any critical changes to the metadata, and sufficiently back it up to ensure it can recover. Though, having a single master is a bottleneck. Hence, we keep a replica in the background:

Definition 1.10: Shadow Master

A **shadow master** is a standby replica of the GFS master that provides high availability:

- **Operation-Log Mirroring:** Maintains a copy of the master's operation log, replicated remotely.
- **State Replay:** Continuously **replays** (applies) logged operations to keep its in-memory metadata (namespaces and file→chunk mappings) nearly up to date.
- **Read-Only Serving:** Can answer client metadata queries (e.g. lookups) in a read-only fashion, offloading the active master.
- **Failover Ready:** On active-master failure, a simple DNS switch (switching the network pointer to the master) promotes the shadow to become the new active master.
- **Eventual Consistency:** May lag slightly behind the active master due to asynchronous log replication and replay.

Though we must point out:

Theorem 1.1: Non-Atomicity and Non-Serializability Across Chunk Boundaries

In GFS, writes that span multiple chunks are neither atomic nor globally serializable. Concretely, there exist two concurrent write operations W_1 and W_2 and chunk indices $i \neq j$, such that the primary for chunk i orders W_1 before W_2, while the primary for chunk j orders W_2 before W_1.

However, concurrent writes are atomic and serializable **within a single chunk**.

To point out, and emphasize:

Definition 1.11: One Primary for each Chunk

In GFS, each chunk is assigned exactly one **primary** replica by the master via a lease.

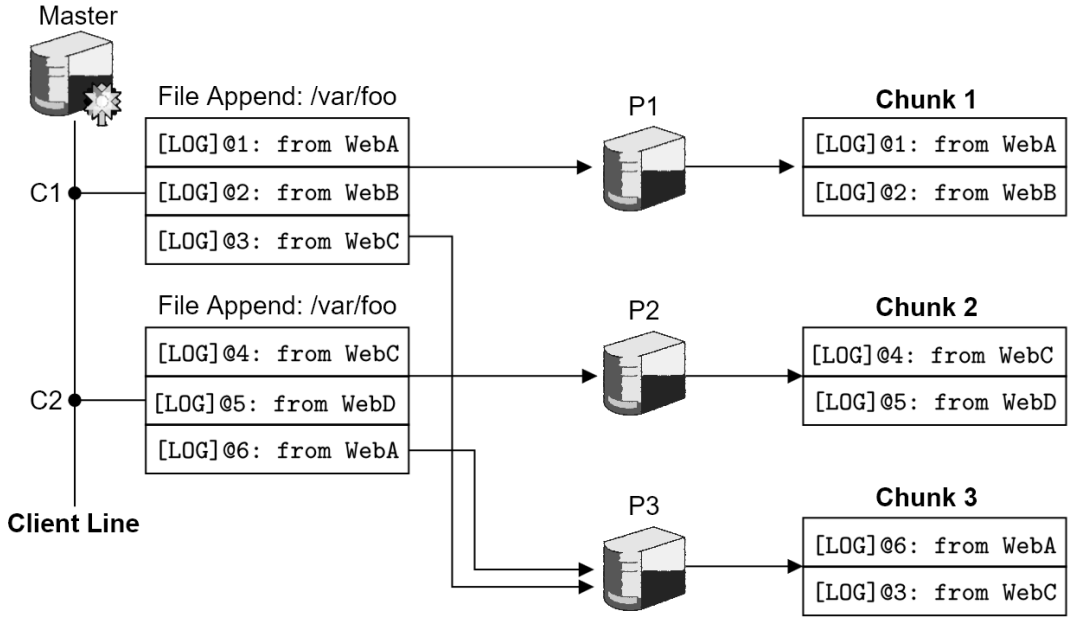

Figure 0.5: GFS record-append across chunk boundaries (client→master→client→primary). Each chunk holds up to two log records. Clients C_1, C_2 issue appends `@1`–`@6`. Under primary P_1 (chunk 1), `@1`,`@2` succeed while the rest are rejected. Each client then re-queries the master and retries on primary P_2 (chunk 2), where `@4`,`@5` succeed but `@3`,`@6` are rejected. A final refresh to primary P_3 (chunk 3) accepts `@6`,`@3` in arrival order, illustrating that independent primaries can impose different orders, and that multi-chunk appends are non-atomic and non-serializable.

Below we summarize the key points of GFS through an illustration:

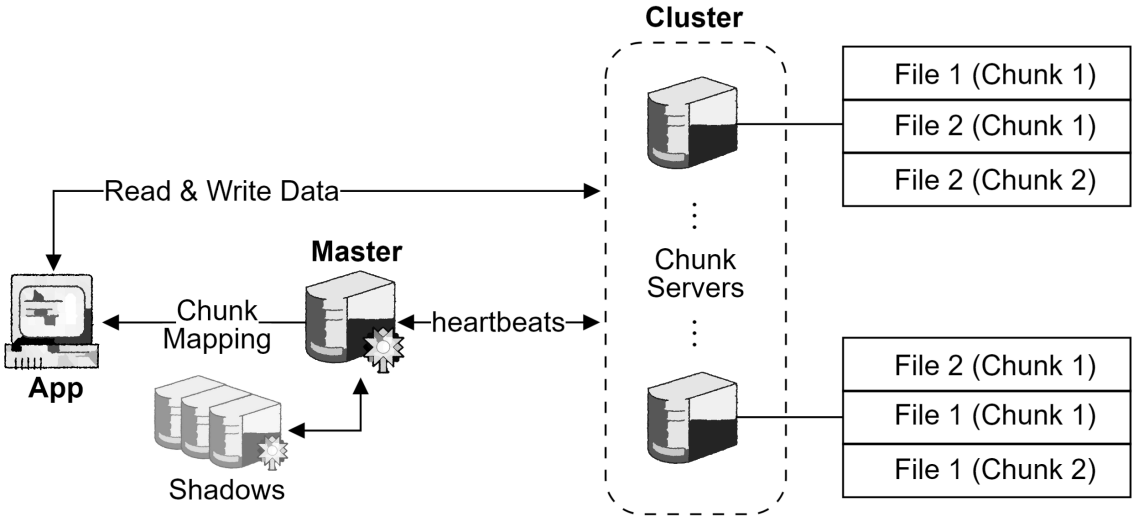

Figure 0.6: High-level GFS architecture. An application (`App`) first contacts the **master** to obtain file→chunk mapping (via file name and chunk index), then directly reads from or writes to the appropriate **chunk servers**, which store fixed-size, replicated chunks (e.g. chunks of File 1 and File 2) across the cluster. The master keeps all metadata in memory, persists mutating operations in an append-only operation log, and receives periodic **HeartBeat** RPCs from chunk servers to monitor replica state and grant leases. **Shadow masters** asynchronously replay the operation log to maintain a nearly up-to-date, read-only metadata copy for load-sharing and fast failover.

Below you may find the GFS paper:
https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

Bibliography

[1] Ioannis Liagouris. Cs351: Distributed systems. Lecture notes, Boston University, Spring Semester, 2025. Boston University, CS Department.