DISTRIBUTED SYSTEMS (CS3.401, MONSOON 2025)

PROJECT: EXACTLY ONCE SEMANTICS FOR RECORDAPPEND IN GOOGLE FILE SYSTEMS

Umang Patel (2022101037)
Vyakhya Gupta (2022101104)

International Institute of Information Technology, Hyderabad

December 1, 2025

Contents

ABSTRACT

The Google File System (GFS) pioneered the RecordAppend operation, a primitive allowing multiple clients to append data concurrently. However, the original architecture guarantees only ”at-least-once” delivery, often resulting in data duplication and application-level complexity. This report details the design and implementation of a simplified GFS prototype that enhances the RecordAppend operation to provide *exactly-once* semantics. By integrating client-side unique identifiers (UUIDs) and a strict primary-ordering mechanism with deduplication logic, our system prevents duplicate records during client retries and network partitions. We present a quantitative evaluation comparing the standard at-least-once approach against our exactly-once implementation, analyzing the trade-offs in throughput and latency.

1 Motivation and Background

The Google File System (GFS) provides a scalable, fault-tolerant, distributed file system designed for large-scale data processing. A key feature of GFS is its RecordAppend operation, which enables multiple clients to append data concurrently to shared files without expensive coordination. However, the original GFS implementation guarantees ”at-least-once” rather than ”exactly-once” semantics. This means that while data is guaranteed to be written, duplicate records may appear as a result of client retries and network failures.

In many modern distributed applications, such as logging systems, message queues, and stream processing frameworks, duplicate records can cause inconsistent results, inflated storage costs, and significant application-level complexity to handle deduplication. Achieving exactly-once semantics is non-trivial and requires careful management of network failures, idempotent retries, and consistency across all data replicas. This project aims to address this gap by designing and building a GFS prototype that supports exactly-once RecordAppend semantics, thereby improving data consistency and simplifying downstream processing.

2 Objectives

The goals of this project were as follows:

1. **Build a Minimal GFS-Like Prototype:** Implement a working distributed file system with a Master, multiple Chunk Servers, and a Client library, closely following the core ideas of the Google File System.
2. **Reproduce the Baseline RecordAppend Behavior:** Add support for the standard GFS `RecordAppend` operation, which provides at-least-once semantics, to serve as a reference point for later comparisons.
3. **Add Exactly-Once RecordAppend Semantics:** Extend the baseline design to ensure each append is applied exactly once, even under retries and failures. This includes:
   - assigning each request a unique identifier (UUID),
   - adding deduplication checks on Chunk Servers, and
   - using a two-phase commit mechanism between the Primary and its replicas to ensure consistent ordering.
4. **Evaluate the Impact on Performance:** Measure how exactly-once support affects throughput and latency compared to the at-least-once baseline.

3 System Design

3.1 High-Level Architecture

The system follows the classic GFS architecture, consisting of three distinct components interacting via RPC (Remote Procedure Calls):

- **Master Node:** The centralized coordinator. It maintains the file system namespace, access control, and the mapping of files to chunks. It also manages chunk leases and tracks the location of replicas.
- **Chunk Servers:** These servers store the actual file data. Files are split into fixed-size ”chunks” (64MB). Chunk servers handle read/write requests directly from clients to avoid bottlenecking the Master.
- **Client Library:** The interface used by applications. It communicates with the Master to locate chunks and directly with Chunk Servers for data transfer.

3.2 Exactly-Once Enhancement Design

Our core solution revolves around making the RecordAppend operation idempotent and consistent across replicas, achieved through a multi-step protocol.

1. **Client-Side Idempotence:** Each `RecordAppend` request initiated by the client will carry a unique identifier (e.g., a UUID). This ID allows the master and chunk servers to distinguish between a new request and a retry of a previous one.
2. **Master and Chunk Server Coordination:** The master server and the primary chunk server will track processed UUIDs to immediately reject duplicate requests, preventing reprocessing.
3. **Two-Phase Commit:** To guarantee atomicity across replicas, we will implement a two-phase commit-acknowledge protocol. The primary chunk server will first log the append and replicate it to a quorum of followers. Only after receiving acknowledgments (the first phase) will it commit the data and confirm success to the client (the second phase), preventing data loss or duplication if the primary fails.

3.3 Key Features

- **Storage Architecture:** Implements 64MB chunk-based storage with default 3-way replication for high availability and fault tolerance.
- **Consistency Guarantees:** Provides exactly-once append semantics using a modified two-phase commit protocol supplemented by strict idempotency tracking.
- **Replication Model:** Utilizes a Primary-Secondary replication scheme coordinated via time-bound leases to prevent split-brain scenarios.
- **Fault Tolerance:** Includes Write-Ahead Logging (WAL) for Master crash recovery, heartbeat-based failure detection, and persistent state management for ChunkServers.
- **Maintenance:** Features automatic garbage collection for orphaned chunks and a comprehensive benchmarking suite.

4 System Architecture

4.1 High-Level Overview

The system follows a master-worker architecture comprising three primary components: the Client Application, the Master Server, and multiple ChunkServers.

The interaction flow is designed to decouple metadata operations from data transfer, ensuring the Master does not become a bottleneck:

1. **Metadata Query:** Clients contact the Master to retrieve chunk metadata (locations and leaseholders).

2. **Lease Coordination:** The Master grants leases to Primary ChunkServers to coordinate mutations.

3. **Heartbeats:** ChunkServers send periodic heartbeats to the Master to report health and inventory.

4. **Data Pipeline:** Data is pushed linearly through the replica chain (Client $\rightarrow$ Replica A $\rightarrow$ Replica B $\rightarrow$ Replica C).

5. **Control Flow:** The Client sends the write command to the Primary, which coordinates the commit across all replicas.

4.2 Component Distribution

4.2.1 Master Server

The Master Server acts as the single point of metadata coordination. It maintains the global file system namespace and the mapping of files to chunks. Crucially, it does not handle data read/write operations. Its responsibilities include lease management, garbage collection, and failure detection via heartbeats.

4.2.2 ChunkServers

ChunkServers are the workhorses of the system. They store actual file data as 64MB chunks on the local disk. They handle read and write requests from clients and replicate data to other ChunkServers to maintain the desired replication factor (default: 3). They also maintain persistent state to track chunk versions and append history.

4.2.3 Clients

The Client component provides the interface for applications to interact with the file system. It is implemented as a library offering high-level APIs, a Command-Line Interface (CLI), and an interactive REPL for testing and benchmarking.

5 Core Components

5.1 Master Server Architecture

The Master is the ”brain” of the system, managing all metadata in memory while ensuring durability through logging.

5.1.1 Data Structures

The Master maintains several critical data structures:

- **FileMap:** A mapping of filenames to an ordered list of chunk handles.
- **ChunkMap:** Stores metadata for each chunk, including its version, current primary, and lease expiration time.
- **ChunkLocations:** A reverse index mapping chunk handles to the set of ChunkServers holding a replica.
- **LeaseManager:** Manages the assignment and expiration of primary leases.
- **OperationLog (WAL):** A sequential log of all metadata mutations (creation, deletion, chunk allocation) used for crash recovery.

5.1.2 Key Responsibilities

1. **Namespace Management:** Handles file creation, renaming, and deletion.
2. **Chunk Location Tracking:** Keeps an up-to-date map of which server holds which chunk, updated via heartbeats.
3. **Lease Management:** Grants temporary leases (60 seconds) to chunks to define a ”Primary” for mutation ordering.
4. **Failure Detection:** Monitors ChunkServers; if a server misses heartbeats (10s timeout), it is marked dead.
5. **Garbage Collection:** Identifies chunks that are no longer referenced by any file and schedules them for deletion.

5.2 ChunkServer Architecture

ChunkServers manage the physical storage and the low-level consistency of data.

5.2.1 Internal State

- **Data Storage:** Chunks are stored as standard files (e.g., `chunk_<handle>.chk`) in the local data directory.
- **Version Control:** Each chunk has a version number. This is incremented by the Master during lease assignment and must match across replicas to ensure data consistency.
- **Buffer Management:** An LRU-based buffer holds incoming data before it is committed. This supports the separation of data flow (pipelining) from control flow.
- **Append State Tracking:** To support exactly-once semantics, the server tracks the status of recent append operations (Preparing, Prepared, Committed) indexed by their unique idempotency IDs.
- **Persistent State:** Critical metadata (chunk versions, append states) is periodically synced to a ‘chunkserver_state.json‘ file to survive restarts.

6 Operations and Workflows

6.1 File Creation

When a client requests to create a file, the Master checks if the filename already exists in the `FileMap`. If not, it creates a new entry with an empty list of chunks. This operation is immediately written to the Write-Ahead Log (WAL) to ensure it survives a Master crash. The Master then returns a success status to the client.

6.2 Chunk Allocation

When a file grows beyond the current chunk boundary, a new chunk must be allocated:

1. The Client requests allocation for a specific filename.
2. The Master generates a unique 64-bit chunk handle.
3. The Master selects 3 ChunkServers to host the replicas. This selection effectively utilizes a random distribution among ”live” servers to balance load.
4. The Master updates the `FileMap` and logs the ‘add_chunk‘ operation to the WAL.
5. The Master returns the new handle and the locations of the 3 replicas to the Client.

6.3 Lease Management

Leases are the mechanism used to maintain consistency during writes. A lease grants a specific ChunkServer the right to determine the order of mutations for a chunk.

- **Lease Request**: When a Primary receives a mutation request, it must hold a valid lease. If it does not, or if the lease has expired, it requests one from the Master.
- **Granting**: The Master checks if a valid lease exists. If not, it increments the chunk lease version and grants a new lease to the requester for 60 seconds.
- **Extensions**: If the Primary is actively mutating data, it can request a lease extension to hold the lock longer.

6.4 Heartbeat Protocol

The system relies on a heartbeat mechanism for health monitoring. Every x seconds, ChunkServers send a `HeartBeat` message to the Master containing their address and a list of all chunks they possess (including version numbers).

1. The Master updates the ‘LastHeartbeat‘ timestamp for that server.
2. The Master checks the reported chunk versions against its own metadata. If a ChunkServer reports an old version, the Master marks that replica as ”stale” and it will no longer be used for read/write operations.
3. The Master piggybacks control commands (like ‘delete_chunk‘) on the heartbeat response.

7 Exactly-Once Append Semantics

7.1 Problem Statement

Standard GFS provides ”at-least-once” semantics. If a client appends a record and the network fails before the acknowledgement is received, the client must retry. In GFS, this retry results in the record being written a second time, creating duplicates. Our system enforces ”exactly-once” semantics: a record appears in the file exactly one time, regardless of retries.

7.2 Idempotency Identifiers

The foundation of this guarantee is the **Idempotency ID**. The client generates a unique identifier (e.g., a UUID or a timestamp-random pair) for every append operation. This ID remains constant across retries of the same operation.

7.3 Record Append Workflow (Two-Phase Commit)

The append operation follows a rigorous protocol to ensure atomicity and idempotency.

1. **Data Push:** The client pushes the data payload to all replicas. The data is cached in a temporary buffer on the servers, identified by a Data ID.
2. **Primary Request:** The client sends an `AppendRecord` RPC to the Primary, containing the Chunk Handle, Data ID, and the Idempotency ID.
3. **Idempotency Check:** The Primary first checks its local state. If an append with this Idempotency ID is already marked as ‘Committed‘, it immediately returns the existing offset and success status.
4. **Phase 1: PREPARE**:
   - The Primary reserves space for the record at the current offset.
   - It creates a local append state record: ‘status=Preparing‘.
   - It forwards a ‘PREPARE‘ command to all Secondaries.
   - Secondaries reserve space, verify they have the buffered data, and acknowledge.
5. **Phase 2: COMMIT**:
   - Upon receiving acknowledgements from all Secondaries, the Primary writes the data to the reserved offset on its local disk.
   - It updates the append state to ‘Committed‘.
   - It forwards a ‘COMMIT‘ command to all Secondaries, which then write their data to disk and update their state.
6. **Response:** The Primary returns the offset and committed chunk version to the client.

7.4 Append State Lifecycle

Each ChunkServer maintains a state machine for append operations:

- **New:** No record exists.

- **Preparing:** Space is reserved, but data is not written.
- **Prepared:** Space is reserved, state is saved, and secondaries have acknowledged readiness.
- **Committed:** Data is durable on disk.
- **Pruned:** Old states are removed after 128 newer appends to prevent memory leaks.

7.5 Persistence

To ensure these guarantees survive server crashes, the append states are persisted to ‘chunkserver_state.json‘. If a server crashes after committing but before responding, it will reload the ‘Committed‘ state upon restart. When the client retries, the server will find this persistent record and return success without duplicating the data.

8 Failure Handling and Recovery

8.1 ChunkServer Crash

If a ChunkServer crashes, the Master detects the failure via missing heartbeats (after a timeout).

- If the crashed server was a Primary, its lease will to expire.
- Clients attempting to write will fail and must request new metadata from the Master.
- The Master will grant a lease to a different replica, incrementing the version number to ensure the system moves forward.

8.2 Master Crash

The Master uses a Write-Ahead Log (WAL). On restart, it replays the log to rebuild the `FileMap` and `ChunkMap`. It does not persist locations of chunks; instead, it waits for ChunkServers to send heartbeats (Heartbeat Re-sync) to rebuild the `ChunkLocations` map dynamically.

8.3 Network Partition and Split Brain

If a Primary becomes isolated from the Master but can still talk to clients, it might try to continue serving. However, leases have a hard expiration (60s). Once the lease expires,

the Primary can no longer serve writes. The Master will grant a new lease to a reachable replica with a higher version number. When the old Primary rejoins, its lower version number will cause the Master to mark it as stale, preventing split-brain data corruption.

8.4 Lost Acknowledgement

A common failure mode is where the write succeeds, but the success response is lost.

1. The Client times out and retries the append with the *same* Idempotency ID.
2. The Primary checks its append state map.
3. It finds the entry with ‘status=Committed‘.
4. It returns the original offset and version to the client.
5. **Result:** The client sees success, and the file contains only one copy of the data.

8.5 Partial Replica Failure

If one secondary fails during the `PREPARE` phase:

1. The Primary receives an error or times out.
2. The Primary sends an `ABORT` command to all replicas.
3. All replicas truncate the reserved space and delete the append state.
4. The operation fails, and the client handles the retry (potentially with a new set of replicas).

8.6 Secondary Failure After Prepare

If a secondary acknowledges the `PREPARE` command but crashes before receiving `COMMIT`:

- The Primary and other Secondaries commit successfully.
- The crashed Secondary has a ”hole” (reserved space with no data).
- When it restarts, it will likely have a stale chunk version compared to the others.
- The Master will detect this version mismatch via heartbeats and mark the replica as stale, eventually triggering garbage collection or re-replication.

9 Garbage Collection

9.1 Orphan Chunk Detection

When a file is deleted, the chunks associated with it become ”orphans.” The Master does not delete them immediately. Instead, it periodically scans its metadata to identify chunks that are present in the `ChunkMap` but not referenced by any file in the `FileMap`.

9.2 Grace Period and Deletion

Identified orphans are placed in a garbage collection queue with a timestamp. They are given a grace period (default 120 seconds) to allow for any in-flight operations to complete and to recover from accidental deletions. Once the grace period expires:

1. The Master removes the chunk metadata from memory.
2. The Master adds a ‘delete‘ command to the command queue for every ChunkServer holding a replica of that chunk.
3. The ChunkServers receive this command in their next heartbeat response and delete the physical files from disk.

10 Evaluation

In this section, we evaluate the performance of our GFS prototype. We focus primarily on quantifying the overhead introduced by the *exactly-once* semantics for `RecordAppend` operations compared to the baseline *at-least-once* implementation. We also report baseline performance for standard Read and Write operations to establish system stability.

10.1 Experimental Setup

The experiments were conducted on a distributed cluster environment. We varied the number of concurrent clients across a range to measure scalability. The key metrics recorded were:

- **Throughput:** Measured in operations per second (ops/sec) for appends, and Megabytes per second (MB/s) for bulk reads/writes.
- **Latency:** The average time taken to complete a single operation.

10.2 Baseline Performance (Read/Write)

We first benchmarked standard Read and Write operations to ensure the underlying chunk server architecture was performant.

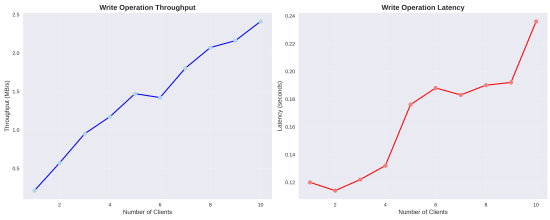


Figure 1: Write Operation Performance

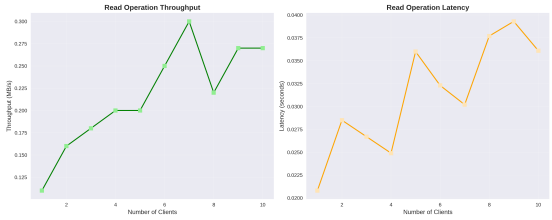


Figure 2: Read Operation Performance

As shown in Figure 1, write throughput scales linearly with the number of clients, demonstrating effective utilization of available bandwidth. Write latency increases moderately under high load. Read operations (Figure 2) showed more variance and a lower peak throughput compared to writes. This disparity is likely due to the random access patterns used in the test workload, which prevents effective pre-fetching.

10.3 RecordAppend Analysis: At-Least-Once vs. Exactly-Once

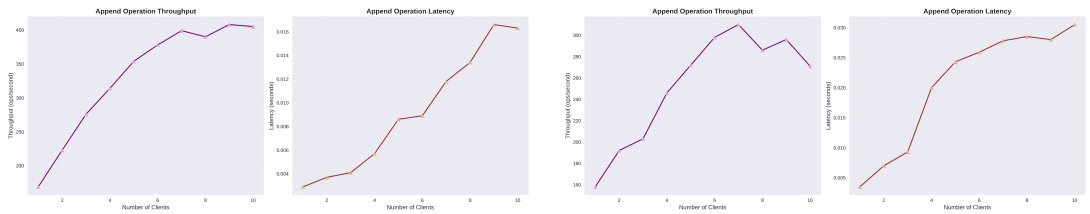
The core of our evaluation compares the standard GFS append (At-Least-Once) against our enhanced implementation (Exactly-Once).

10.3.1 Throughput Comparison

Figure 3 illustrates the throughput differences.

- **At-Least-Once (Baseline):** The system exhibits strong scalability. Throughput increases steadily as the number of clients grows. The curve is relatively linear, indicating minimal coordination overhead.
- **Exactly-Once:** The throughput is lower, as expected. It saturates earlier than the baseline before degrading slightly under maximum client load.

The exactly-once mechanism introduces a noticeable overhead in peak throughput. The drop-off observed at higher concurrency levels suggests that the primary chunk server becomes a bottleneck due to the additional CPU cycles required for UUID deduplication checks and the two-phase commit logic.



(a) Baseline (At-Least-Once) (b) Enhanced (Exactly-Once)

Figure 3: Comparison of RecordAppend Throughput and Latency

10.3.2 Latency Analysis

The latency cost of exactly-once semantics is more pronounced than the throughput cost.

- **Baseline:** Latency grows linearly and predictably with the number of clients.
- **Exactly-Once:** Latency grows more rapidly compared to the baseline.

This substantial increase in latency is attributed to the strict ordering requirements. In the baseline model, the primary can forward data to replicas immediately. In the exactly-once model, the primary must wait for a full acknowledgement cycle (commit phase) to ensure the operation is logged persistently before responding to the client, effectively doubling the network round-trips required for a successful return.

10.4 Performance Bottlenecks

1. **Two-Phase Commit**: Adds an extra network round-trip compared to standard GFS, adding 50ms latency.
2. **Disk Sync**: The system performs ‘fsync()‘ calls to persist append states for correctness, which is a slow I/O operation (10-20ms).
3. **Lease Acquisition**: The first write to a chunk incurs the latency of a Master RPC to acquire the lease.

11 Limitations and Future Work

11.1 Current Limitations

- **Single Master:** The Master is a single point of failure. While the WAL allows for recovery, the system is unavailable during the restart process.

11.2 Future Enhancements

- **Distributed Master**: Implementing a consensus algorithm like Raft for the Master would eliminate the single point of failure.
- **Distributed Consensus**: Implement Raft or Multi-Paxos for leader election and log replication. This allows the system to make progress as long as a quorum of replicas is alive, increasing avalibility.
- **Load Balancing**: Ooptimizing resource utilization across Chunkservers. Prevent hotspots by distributing file chunks more evenly
- **Geographical Separation**: Adapt for non-localized, geographically distributed environments. Address higher latency and consistency trade-offs when replicas span across different regions/datacenters.

12 Conclusion

Implementing exactly-once semantics introduces a fundamental trade-off between operation latency and data consistency.

While standard ”at-least-once” systems prioritize raw throughput by allowing duplicates during network partitions, our approach prioritizes strict data integrity.

The introduction of the Two-Phase Commit (2PC) protocol and UUID verification adds measurable network overhead, sacrificing the speed of a pipelined write for the guarantee that the file state remains deterministic across all replicas.

References

[1] Ghemawat, S., Gobioff, H., & Leung, S. (2003). The Google File System. *SOSP ’03: Proceedings of the nineteenth ACM symposium on Operating systems principles*.

[2] Ongaro, D., & Ousterhout, J. (2014). In search of an understandable consensus algorithm. *2014 USENIX Annual Technical Conference (USENIX ATC 14)*.

[3] Lamport, L. (2001). Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32*, 4 (Whole Number 121, December 2001), 51-58.

[4] Liskov, B., & Cowling, J. (2012). Viewstamped Replication Revisited. *MIT-CSAIL-TR-2012-021*.