Distributed System Course Project Implementing Exactly-Once Semantics in RecordAppend for Google File System

Sahil Patel (2022101046)
Priet Ukani (2022111039)

November 27, 2024

Contents

Chapter 1

Introduction to Google File System

1.1 Background

The Google File System (GFS) is a distributed file system designed to manage large-scale data reliably and efficiently in a distributed environment. GFS introduced innovative techniques like chunk-based storage, replication for fault tolerance, and a master-chunk server architecture for scalability. This project replicates and extends GFS functionalities, including file upload, read operations, periodic server pings, and an exactly-once append semantic using Two-Phase Commit (2PC) for ensuring consistency.

1.2 Google File System Overview

The Google File System (GFS) is a robust DFS designed to meet Google's internal needs for handling vast datasets. It supports:

- **Large-Scale Data Processing**: Ideal for big-data workloads.
- **Fault Tolerance**: Automatic recovery from hardware failures.
- **High Throughput**: Optimized for sustained data-intensive operations.
- **Append Operations**: Frequent file appends rather than overwrites.

Chapter 2

Google File System Implementation

2.1 Core Features of GFS

- **Chunk-Based File Storage**: Files are divided into fixed-size chunks (e.g.,CHUNK_SIZE in our GFS implementation). Each chunk has a globally unique ID and is replicated across multiple chunk servers.
- **Master and Chunk Server Architecture**: The master server maintains metadata, including chunk locations and file-to-chunk mappings. Chunk servers store actual data and periodically report their status to the master.
- **Fault Tolerance**: Data replication ensures reliability in case of chunk server failures. Heartbeat signals from chunk servers enable the master to detect failures and trigger recovery.

2.2 Other features

- **Periodic Pings**
  Purpose: To maintain an updated list of active chunk servers. To allow the master to detect and handle chunk server failures dynamically.
  Implementation: Chunk servers send periodic pings to the master, including their active status, stored chunks etc The master updates its metadata with the received information and removes failed servers from the mapping.
- **Upload Operation**

  Purpose: To distribute chunks of a file across chunk servers while ensuring replication for fault tolerance.

  Implementation: The client sends the file to the master, which splits it into chunks. For each chunk: The master selects two chunk servers for replication. The chunk is sent to the first server, which replicates it to the second.

  Assumptions: Chunk servers have sufficient storage capacity. Network latency is consistent and negligible for uploads.

- **Read Operation**

  Purpose: To retrieve specific chunks of a file efficiently.

Implementation: The client sends a read request to the master, specifying the file and offset. The master retrieves the chunk metadata and returns the corresponding chunk server locations to the client. The client fetches the data directly from the chunk servers.

- **Exactly-Once Append Using Two-Phase Commit (2PC)**

  Purpose: The primary goal of implementing exactly-once append semantics using the Two-Phase Commit (2PC) protocol is to ensure atomicity and consistency in distributed systems, especially during concurrent write or append operations. In systems like GFS, where multiple chunk servers store replicas of the same data, it is crucial to ensure that all replicas either commit the operation together or roll back entirely, maintaining a consistent state across the system. This mechanism prevents duplicate writes and guarantees the integrity of data.

  Implementation: The implementation of 2PC for exactly-once append operations consists of two phases:

  1. **Prepare Phase**: - The master server initiates the append operation by sending a 'PREPARE' request to all involved chunk servers that store replicas of the target chunk. - Each chunk server performs validation checks to ensure that it can execute the operation. These checks may include verifying available space, ensuring the operation adheres to system constraints, and confirming the current state of the chunk. - Once validated, the chunk server responds to the master with a 'PREPARED' acknowledgment, signaling that it is ready to commit the operation. - If a chunk server encounters an issue (e.g., insufficient resources or conflicts), it sends a 'FAIL' response, halting the process.

  2. **Commit/Abort Phase**: - Upon receiving 'PREPARED' acknowledgments from all involved chunk servers, the master sends a 'COMMIT' request to finalize the operation. - Each chunk server applies the append operation and sends a confirmation back to the master. - If the master receives any 'FAIL' responses during the prepare phase or detects a timeout, it sends an 'ABORT' request to all chunk servers, rolling back the operation and restoring the previous state. - This two-phase process ensures that either all servers commit the operation or none do, preserving atomicity.

  Assumptions: - **Network Reliability**: Although 2PC is designed to handle failures, it assumes that network failures are infrequent. In cases of intermittent failures, retries are employed to reestablish communication. - **Server Compliance**: All chunk servers strictly adhere to the 2PC protocol, ensuring predictable responses during prepare and commit phases. - **Master Authority**: The master server acts as the single coordinator, maintaining a global view of the operation and deciding its final outcome.

  Benefits: - **Atomicity**:By using 2PC, the system guarantees that the append operation is either fully applied across all replicas or not applied at all. - **Consistency**: Ensures that all chunk servers maintain a consistent state for each chunk, critical for distributed file systems. - **Concurrency Control**:Prevents conflicts during simultaneous append operations by introducing a centralized control mechanism.

- **Replication Strategy**: Our replication strategy ensures data durability and availability:

– **Three-Way Replication**: Each chunk is stored on three different servers.
– **Primary Replica**: One server coordinates the write process.
– **choosing servers**: chunk servers are chosen randomly out of total chunk servers.

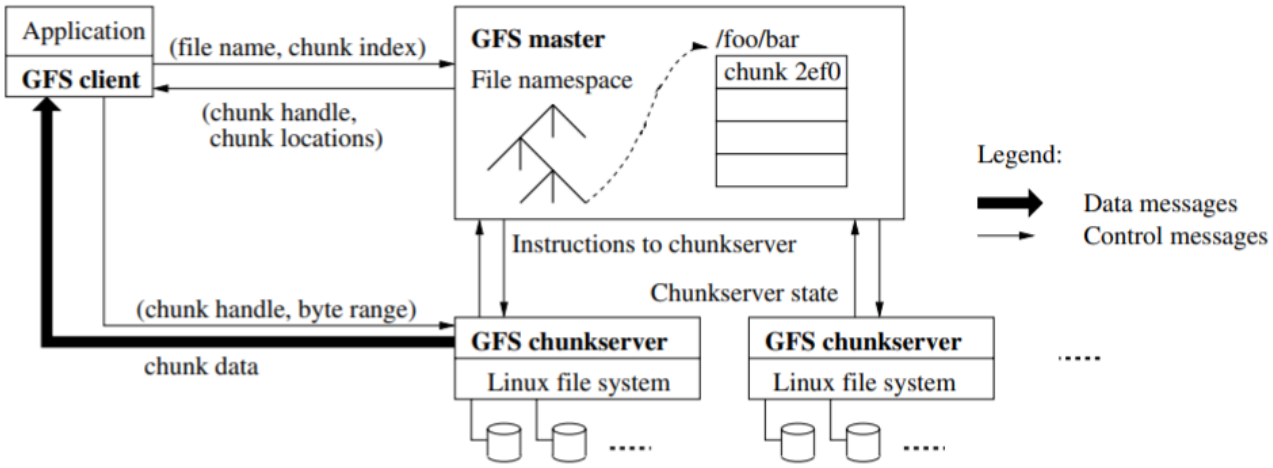

Figure 2.1: Architecture in Picture

Chapter 3

Challenges and Solutions

3.1 Synchronizing Chunk Metadata During Server Failures

Challenge: Ensuring that metadata remains consistent when chunk servers fail is critical for maintaining the integrity of the distributed file system. Failures can lead to stale or missing metadata, affecting system reliability and availability.
Solution: To address this, periodic pings from chunk servers to the master server are implemented. These pings provide the master with real-time information about active chunks and server statuses. This ensures fault tolerance and data consistency, even in the event of server failures.

3.2 Handling Concurrent Appends

Challenge: Concurrent append operations pose a risk of data conflicts and overwrites if not managed properly. Ensuring atomicity and consistency in such scenarios is a complex task, especially when multiple clients attempt to append data to the same chunk.
Solution: The Two-Phase Commit (2PC) mechanism was implemented to guarantee atomicity. The master coordinates appends by ensuring all chunk servers involved in the operation either commit the change or abort it collectively. This prevents overwrites and ensures that concurrent operations do not compromise data integrity.

3.3 Efficient Chunk Server Selection

Challenge: Dynamically balancing the load across chunk servers is essential for optimal performance and scalability. Without effective load balancing, some servers may become overloaded while others remain underutilized.
Solution: The master employs a random selection strategy to distribute chunks among active chunk servers. This approach ensures even load distribution and avoids hotspots, contributing to system efficiency. By leveraging real-time information about server statuses from periodic pings, the master adapts chunk allocation dynamically as the system evolves(future work).

Chapter 4

Ideas Attempted but Unsuccessful

4.1 Lazy Replication Mechanism

Idea: The lazy replication mechanism aimed to optimize resource utilization by deferring the replication of chunks until it was absolutely necessary. Instead of ensuring immediate replication after a write or append operation, the mechanism would prioritize other critical tasks, such as responding to client requests, and delay replication to less active periods.

Reason for Failure: While this approach seemed efficient in theory, it introduced significant complexity in tracking chunks that had not been replicated. The system needed to maintain a detailed state of all unreplicated chunks, which increased the overhead on the master server. Moreover, in high-load scenarios, deferred replication led to consistency issues, particularly when chunk servers failed before replication was completed. This inconsistency violated the replication guarantees of the system, ultimately making the lazy replication mechanism impractical for the GFS architecture.

Chapter 5

Future Work

- **Dynamic Load Balancing**: Currently, the chunk server selection process uses a random distribution mechanism to balance load across servers. However, this can lead to uneven resource utilization in scenarios with high traffic or unbalanced file sizes. Future improvements could involve implementing a dynamic load-balancing strategy that considers real-time server metrics such as CPU usage, memory availability, and network bandwidth. This would enhance the overall system performance and reliability.
- **Chunk Compression**: To optimize storage utilization and reduce network overhead, chunk-level compression can be introduced. By compressing chunks before storage, the system can save significant disk space, especially for large files with redundant data. Additionally, transmitting compressed chunks over the network can improve latency and bandwidth efficiency. The trade-offs between compression time and storage savings need to be carefully analyzed for optimal performance.
- **Enhanced GFS Core Functionalities**: The current implementation provides fundamental operations such as upload and read. Future work could involve adding critical functionalities like:
  - *Delete Operation:* Allow clients to remove files and their associated metadata, ensuring that chunks are also properly deallocated from chunk servers.
  - *Write Operation:* Extend the system to support in-place updates of chunks, a feature that is not currently available in the append-only model. This would require synchronization mechanisms to maintain consistency across replicas.
- **Fault Tolerance and Recovery Improvements**: While the current implementation ensures metadata consistency during server failures, future enhancements could focus on automatic recovery mechanisms. For instance, failed chunk servers could trigger immediate replication of their chunks to other active servers, reducing downtime and maintaining replication factors.
- **Security Enhancements**: Implement authentication protocols to verify the identity of clients and chunk servers, preventing unauthorized access.
- **Support for Cross-Data Center Replication**: Extend the system to handle geographically distributed chunk servers. This would involve implementing mechanisms for cross-data center replication and ensuring low-latency reads and writes across regions.

Chapter 6

Conclusion

We successfully implemented exactly-once semantics in a distributed file system, overcoming traditional challenges of at-least-once guarantees. Our system provides strong consistency and robust performance across multiple servers.

report link :`https://www.canva.com/design/DAGXtQOlMqI/CXpwunBcdJbSmKvvElQ7FA/edit?utm_content=DAGXtQOlMqI&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton`