DISTRIBUTED SYSTEM COURSE PROJECT:

IMPLEMENTING EXACTLY-ONCE SEMANTICS IN RECORDAPPEND FOR GOOGLE FILE SYSTEM

MAYANK MITTAL (2022101094)
SHIVAM MITTAL (2022101105)
TEAM ID: 56

CONTENT

GOALS AND OBJECTIVES

Objective n° 1

Ensure Data Consistency and Atomicity: Implement an exactly-once record append using a Two-Phase Commit protocol to guarantee that data is consistently replicated across all chunk servers without partial updates.

Objective n° 2

Achieve Fault Tolerance and Reliability: Design mechanisms to detect, handle, and recover from failures, ensuring data integrity and maintaining system reliability during network issues or server crashes.

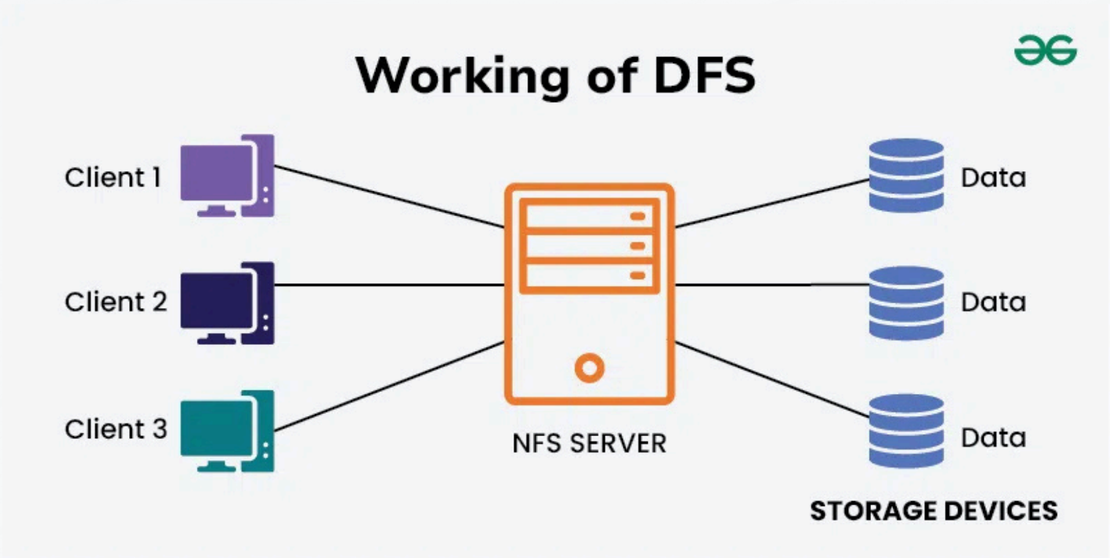
Objective n° 3

Enable Scalable and Efficient Data Management: Develop a distributed architecture with dynamic load balancing, replication, and synchronization to support large-scale data processing and high-performance operations.

DISTRIBUTED FILE SYSTEM

Distributed File Systems (DFS) are essential for managing large datasets across multiple servers. They provide key benefits such as:

- Scalability: Ability to handle increasing amounts of data by adding more nodes.
- Fault Tolerance: Ensuring data availability despite hardware failures.
- High-Performance Storage: Supporting parallel data processing and efficient I/O operations.
- Distributed Management: Enabling seamless data access across geographic locations.



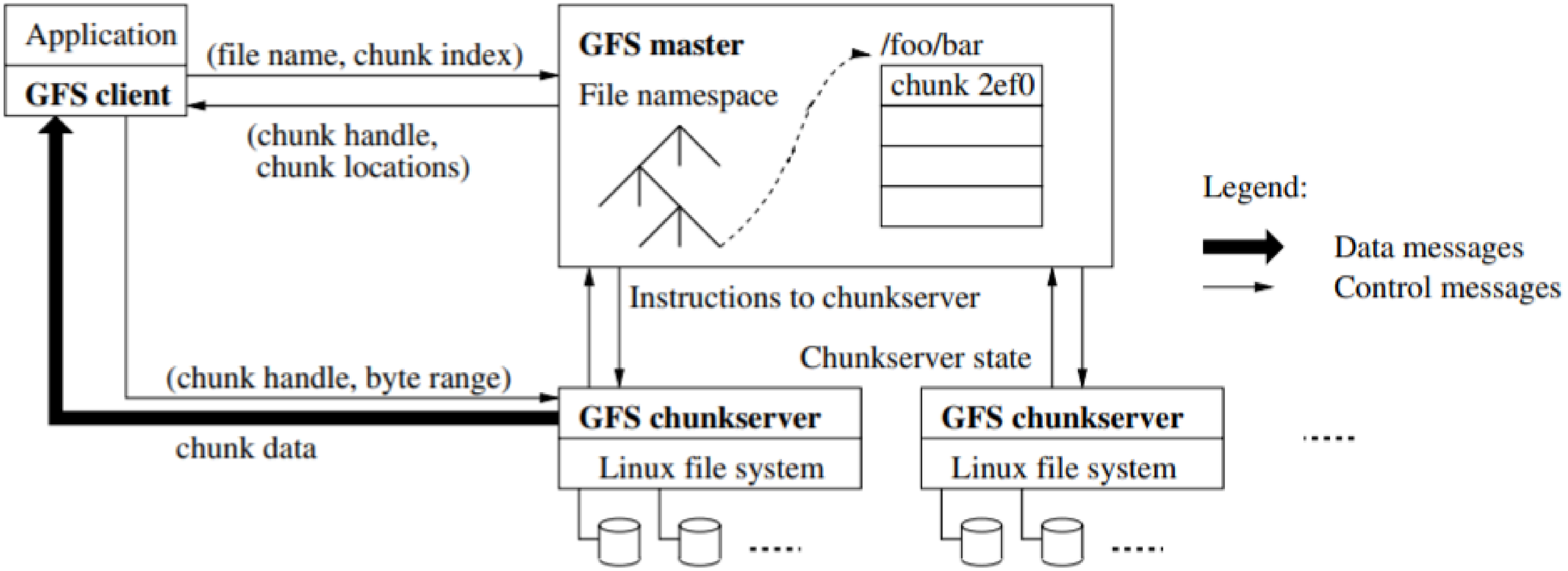
The Google File System

The Google File System (GFS) is a scalable, distributed file system designed by Google to handle large-scale data storage and processing workloads. It provides high fault tolerance, high throughput, and scalability across many inexpensive commodity servers, making it ideal for managing massive amounts of data in distributed environments. GFS is optimized for Google's internal data needs, such as search and data-intensive operations like web indexing and analytics.

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.

WORKFLOW IN GFS



GOOGLE FILE SYSTEM ARCHITECTURE

Google distributed file system architecture comprises three key components:

1) Master Server

The master server handles the system's metadata, including:

- Metadata Management: Tracks file and chunk locations.
- Client Coordination: Directs clients to the appropriate chunk servers.
- Configuration Management: Monitors system-wide configurations.

2) Chunk Servers

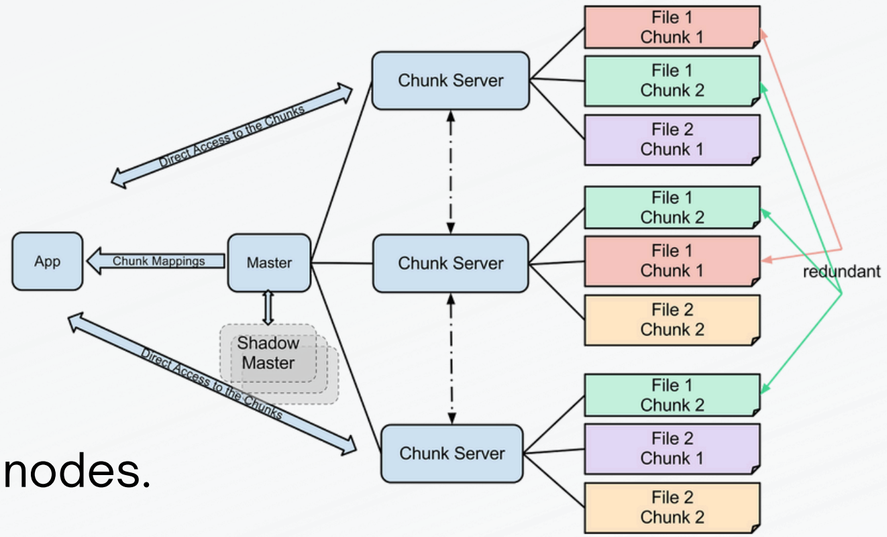
Chunk servers store the actual data and manage replication:

- Data Storage: Holds file chunks.
- Replication Management: Ensures data is replicated across multiple nodes.
- Read/Write Operations: Handles client requests for data.

3) Clients

Clients interact with the master and chunk servers to perform file operations:

- File Operations: Read, write, and append data.
- Communication: Send requests to the master and chunk servers.



SEMANTICS: AT-LEAST-ONCE VS EXACTLY-ONCE

At-Least-Once Semantics in GFS :-

GFS traditionally uses at-least-once semantics, meaning:

- Each operation is retried if the server fails to acknowledge it.
- It guarantees that the operation will be performed at least once.
- Potential downside: Duplicate writes may occur if an operation is retried after a partial success.

Drawbacks:

- Data consistency issues due to multiple appends.
- Clients need to handle deduplication.

Exactly-Once Semantics in Our Project :-

We implement exactly-once semantics to ensure:

- Each operation is performed exactly once, even in the event of failures.
- Avoidance of duplicate data writes.
- Stronger consistency guarantees across distributed nodes.

Implementation Techniques:

- Transaction IDs: Unique IDs for each operation.
- Two-Phase Commit: A coordination protocol ensuring atomic operations

EXACTLY-ONCE SEMANTICS IN RECORD APPEND OPERATION

Advantages :-

- Strong consistency.
- Simplified client logic (no de-duplication needed)

Applications of Exactly-Once Semantics :-

- Financial systems: Ensures accurate transaction records.
- Messaging systems: Guarantees messages are delivered once.
- Logging systems: Prevents duplicate log entries.

EXACTLY-ONCE SEMANTICS IN RECORD APPEND OPERATION

Implementation: Two-Phase Atomic Commit Protocol

- **Prepare Phase:**
- Generate a unique transaction ID.
- Check if all chunk servers have sufficient storage, can replicate data, and have no conflicts.
- Proceed only if all nodes confirm readiness to prevent partial updates.
- **Commit Phase:**
- Append data to all chunk servers synchronously.
- Ensure atomicity, where all nodes either commit or abort together.
- Update the master server’s metadata and roll back on failure.
- **Failure Handling:**
- Detect failures using transaction logs and retries.
- Roll back partial writes to maintain data integrity.
- Abort the transaction and notify the client if necessary.

REPLICATION STRATEGY

Our replication strategy is designed to ensure high availability and durability of data:

- **Three-Way Replication:** Each file chunk is replicated across three different servers to ensure redundancy. This allows the system to tolerate up to two server failures without data loss. Replication provides both fault tolerance and load balancing by distributing read operations across replicas.
- **Primary Replica:** Among the three replicas, one is designated as the primary replica. This primary server coordinates all write and append operations to ensure consistency. The primary replica acts as the leader and manages updates to secondary replicas, enforcing a consistent ordering of writes.
- **Consistency Checks:** Periodic background tasks perform consistency checks across replicas to detect and resolve data mismatches. If a replica falls out of sync, the system triggers a re-replication process to restore consistency. These checks ensure data integrity over time.
- **Automatic Recovery:** In the event of a server failure, the system automatically identifies the lost replica and initiates a recovery process. A new replica is created by copying data from the remaining replicas. This self-healing mechanism is critical for maintaining data availability

CORE OPERATIONS IMPLEMENTATION

File Operations: Read, Upload, Write, Create and Record Append

In our system, the core file operations are implemented as follows:

- **Read Operation:** The client sends a read request to the master server, which redirects it to the appropriate chunk servers. The client then retrieves the data from the replicas, choosing the nearest one to minimize latency.
- **Upload Operation:** The client uploads a file by dividing it into chunks and sending each chunk to a set of chunk servers. The master server coordinates the process and maintains metadata about the file and its chunks.
- **Write Operation:** Writes are handled similarly to uploads but may involve updating existing chunks. The primary replica manages the write operation, ensuring consistency across replicas through a synchronized commit process.
- **Create Operation:** The client can create new files or directories. The master server updates the metadata and assigns chunk servers to store the initial chunks of the file.
- **Record Append:** The record append operation allows clients to append data to a file. This is especially useful for log files and other append-only datasets

IMPORTANT CONSIDERATIONS

1) Network Communication

- TCP Sockets: Used for reliable client-master-chunk server communication.
- Persistent Connections: Long-lived connections reduce setup overhead.
- Serialization: Protocol buffers ensure efficient data transmission.
- Timeouts & Retries: Handle network errors with retries after timeouts.
- Bidirectional Channels: Enable dynamic, real-time communication.

2) Performance Considerations

- Scalability:
  - Sharding: Splits data across servers for parallel processing.
  - Load Balancing: Distributes requests evenly to avoid bottlenecks.
  - Caching: Stores frequently used data in memory for faster access.
- Latency & Throughput:
  - Low Latency: Fast responses via optimized data paths.
  - High Throughput: Supports large data volumes with parallel operations.

3) **Problem Solving & Assumptions**

- Challenges: Addressed network failures, atomicity, and duplicate requests.
- Approaches Tried:
  - Optimistic Concurrency: High retries led to inefficiency.
  - Centralized Transaction Manager: Effective but a bottleneck.
  - Two-Phase Commit: Chosen for simplicity and consistency.
- Assumptions: Reliable messaging and consistent server state after restarts.

KEY CONSIDERATION

FUTURE WORK

Future improvements could focus on:

Automatically adjusting the number of chunk servers

DYNAMIC SCALING

Reducing overhead in two-phase commits.

OPTIMIZED COMMIT PROTOCOLS

Using machine learning to predict failures.

ADVANCED FAULT DETECTION