Google File System

Lecture 14

Mattia, Anna, Prahalath

What is Google File System (GFS)?

Scalable distributed file system for large distributed data-intensive applications

Designed for Google's rapidly growing data processing needs

Largest GFS clusters have over 1000 storage nodes and over 300 TB of disk storage

Design Overview

Design Assumptions and Architecture Overview

Design -> Assumptions

- Physical parts are made from inexpensive components that often fail
- The system will store at least around 100 TB of data (optimized for multi-GB files)
- Workloads mostly comprised of large streaming reads or small random reads
  - Large streaming reads are optimized
- Workloads can also have large streaming writes or small random writes
  - Large streaming writes are optimized
  - Small random writes supported but are virtually nonexistent
- Must efficiently support multiple clients concurrently reading and writing same files
- High-sustained bandwidth > low latency

Design -> Architecture

- Single Master
- Multiple Chunkservers
- Multiple Clients
- Usually run on their own Linux machine, though can also be partitioned on single machine
- Files are divided into Chunks (64 MB)
    - Chunk handle: 64-bit global identifier assigned by Master upon chunk creation
    - Chunks stored on Chunkservers' local dicks
    - Each chunk is replicated on multiple Chunkservers (default is 3 replicas)
- Key feature: files are appended to, not overwritten
    - Foundation of optimization, integrity, and atomcity

Design -> Architecture -> Single Master

- Contains all system metadata
  - Namespaces
  - Access Control Information
  - File → Chunk mappings
  - Current location of Chunks
- Controls system-wide activities
  - Chunk lease management
  - Garbage collection
  - Chunk migrations
- Cannot directly interact with Chunks
  - Prevents bottleneck
- Instead, Master sends Chunkserver metadata to Client
  - Client then contacts Chunkservers directly

Design -> Architecture -> Chunk Size

- 64 MB in size, rather big
- Advantages:
  - Reads and writes on same Chunk require just one query to Master
    - Reduces Client-Master interactions
  - Client more likely to perform all operations on the same chunk
    - Reduces network overhead (one TCP connection)
  - Less metadata to store on Master
- Potential Disadvantage: Hotspots
  - Small files may be comprised of just one or few Chunkservers
  - These Chunkservers may become overloaded (i.e. hotspots) if many Clients are simultaneously accessing that same small file
  - Solution: replicate these files more than others and stagger application start times

Design -> Architecture -> Metadata

- Three main types
  - File and Chunk namespaces
  - Files → Chunk mappings
  - Locations of each Chunk's replicas
- All metadata kept in the Master's memory
  - Allows for Master operations to be fast
  - Allows for periodic scanning for garbage collection, re-replication, and chunk migration
- Persistent (on disk) metadata
  - Operation Log
    - File and Chunk namespaces
    - Files → Chunk mappings
  - Stores historical records of critical metadata changes and serves as logical timeline
  - Crucial for recoverability
- Master polls Chunkservers for its replicas of a given Chunk (not kept persistent)

System Interactions

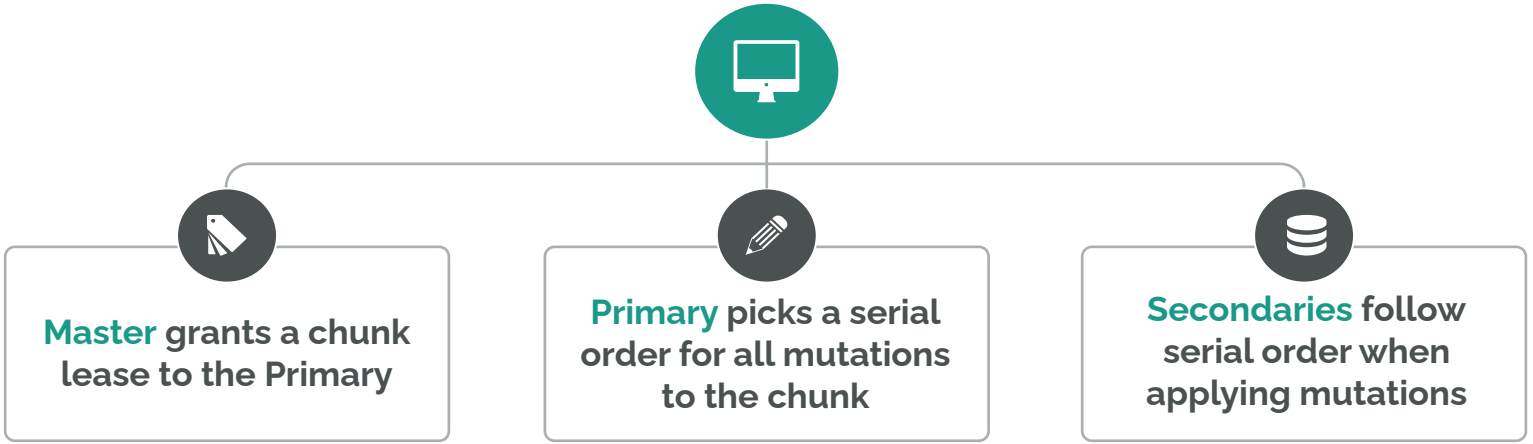
How the client, master, and chunkservers interact to implement data mutations, atomic record append, and snapshot. Designed to minimize the master's involvement in all operations.

Data Mutations

Operations that changes the contents/metadata of a chunk i.e. append, write, snapshot

Each mutation is performed on all the chunk's replicas – How is this managed?

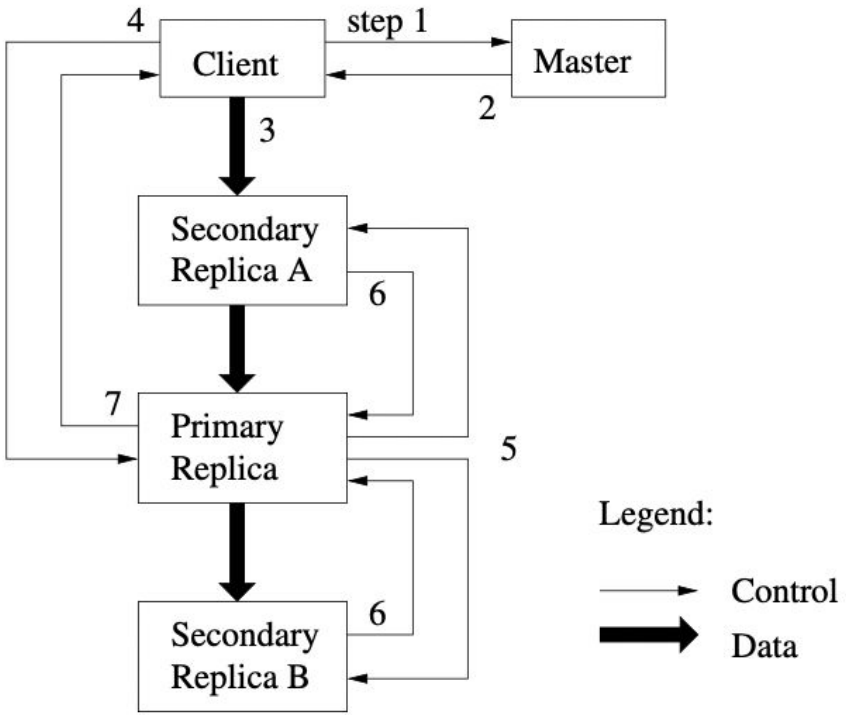
Lease Mechanism: Maintaining Mutation Order



Using this mechanism, the client typically only needs to interact with the master once. This minimizes management overhead at the master

Lease Mechanism:

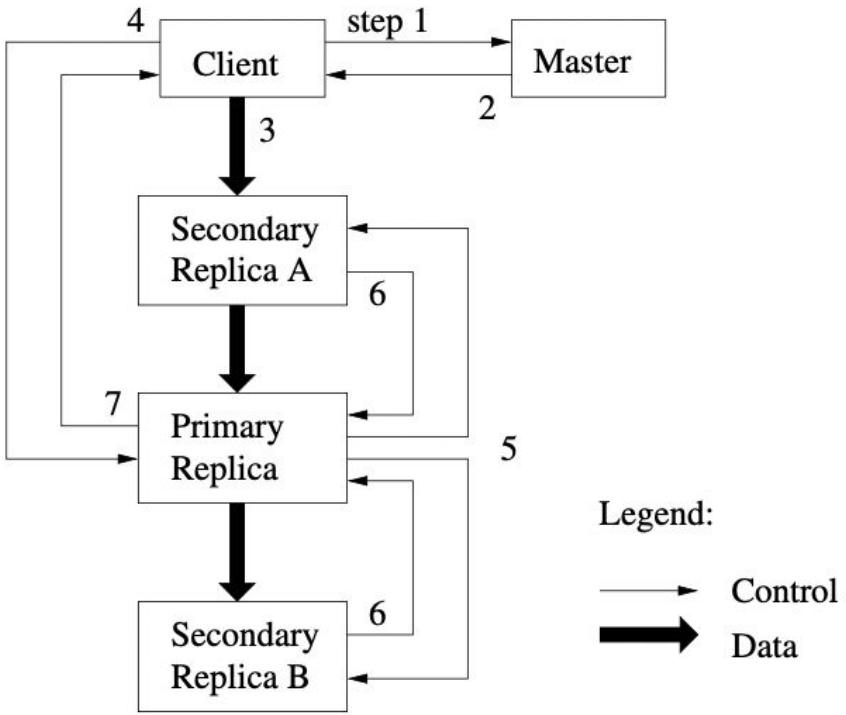
Illustrated by Following the Control and Data Flow of a Write Operation



Data flow and Control flow is Decoupled

Control flows from client -> primary -> secondaries

Data flows linearly through chain of chunkservers in a pipelined fashion



Data flow is designed to fully utilize each machine's network bandwidth, avoid network bottlenecks, and high-latency links

- Data flows linearly in a pipelined fashion:
  - Allows full utilization of each machine's network bandwidth
- Data flow is based on network topology:
  - Each machine sends data to the "closest" machine in network topology
  - Improves performance, avoids network bottlenecks and high-latency links
- Data transfer is pipelined over TCP connections
  - Chunkservers start forwarding data immediately after it receives data
  - In ideal circumstances, 1MB can be distributed in 80 ms

Atomic Record Append

Client only specifies the data. Data is appended atomically at an offset determined by GFS, offset is returned to client.

Allows concurrent appends by many clients on different machines.

Guarantees that data is written at least once as an atomic unit.

Snapshot

Makes a copy of a file/directory tree instantaneously, without interrupting ongoing mutations.

Used to quickly create branch copies of huge data sets or to checkpoint the current state.

Implemented using standard copy-on-write techniques.

Master Operations

Overview of Namespace Management and Replication

Ensuring Concurrency and Consistency with Namespace Management and Locking

- Master Operations can take an extended period of time to complete, so we want multiple operations to run at the same time. However, we want to preserve the integrity of the operations
  - A lookup table that maps each full pathname to metadata and Lock portions of the namespace for proper serialization
  - Example of a given operation on the following pathname “/d1/d2/…/dn/leaf”
    - Operation will typically acquire a read lock for the following paths: “/d1”, “/d1/d2”, …., “/d1/d2/…/dn”
    - Operations will then acquire a write lock and/or read lock for “/d1/d2/…/dn/leaf”
  - Can we add multiple files are to the same directory?
    - Yes, read lock on the paths and write lock on the exact path being created

Ensuring File Integrity with Chunk Replicas

Creation

(1) Place new Replicas on chunkservers with low disk utilization

(2) Avoid doing too many creations in the same chunkserver

(3) Replicas across racks in the server

Re-Replication

(1) Have a user-specified goal for the number of replicas for the files

(2) Prioritize replicas for files that are farther from their goal

(3) Boost replicas for chunks that could hinder client progress

Rebalancing

(1) Master Moves Replicas for better disk space and load balancing

(2) Fills up new chunkservers rather than adding chunks to existing chunkservers

(3) Removes chunks from chunkservers with low free space

Reliable and Simple Garbage Collection

Mechanism

- Master logs the deletion of a file immediately, and path is changed to a hidden name
- Still able to access the deleted file explicitly, until regular scan of the file system permanently removes those namespaces

Stale Replica Detection

- If a chunkserver is unable to update its replica (and version number), it's stale
- When the chunkserver comes back, the master labels it to be stale
- The stale replica is hidden from user and removed during garbage collection mechanism

Fault Tolerance and Diagnosis

Handling frequent component failures

Ensuring High Availability: Fast Recovery and Replication

Fast Recovery

- Both the master and the chunkservers are designed to restore their state and restart in seconds, regardless of how they terminated
- Does not distinguish between normal and abnormal termination

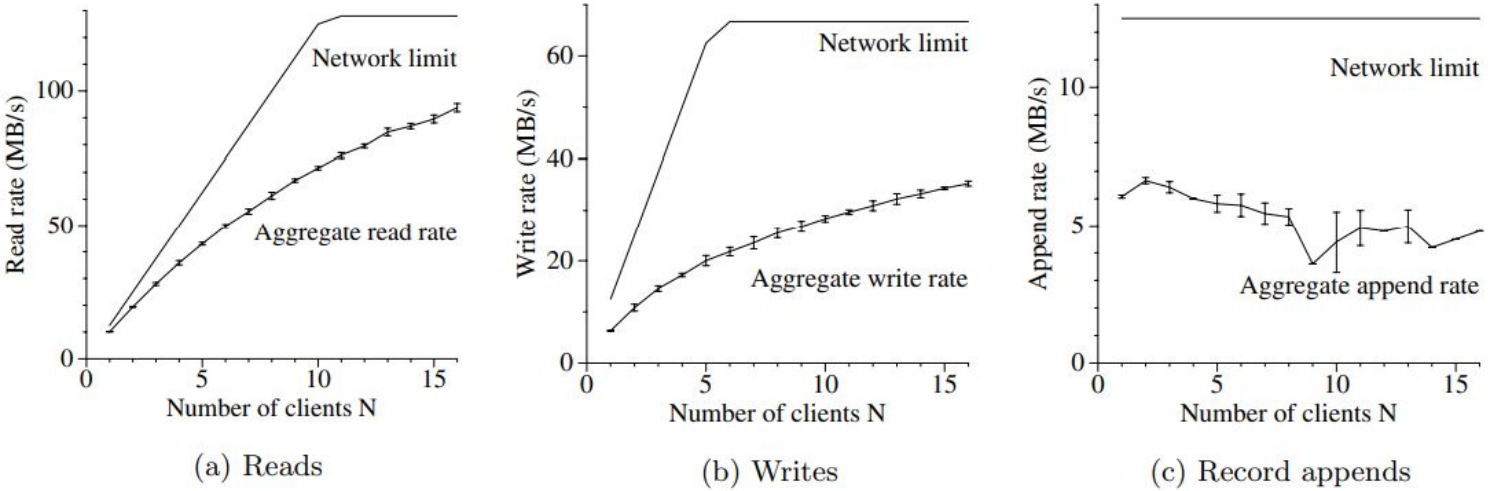
Replications

- Each Chunk is replicated on multiple chunkservers
- If one replica is corrupted, we can recover using other chunk replicas
- The master state, including the operation log and checkpoints, are replicated on multiple machines for reliability
- Mutation is committed only after the log record is flushed to disk locally and on all master replicas

Measurements

Performance of the GFS and Usage Experience

Micro-Benchmark on a Simple GFS Cluster



Real-World Examples

Cluster	A	B
Chunkservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	155 TB
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	48 MB	60 MB

Cluster	A	B
Read rate (last minute)	583 MB/s	380 MB/s
Read rate (last hour)	562 MB/s	384 MB/s
Read rate (since restart)	589 MB/s	49 MB/s
Write rate (last minute)	1 MB/s	101 MB/s
Write rate (last hour)	2 MB/s	117 MB/s
Write rate (since restart)	25 MB/s	13 MB/s
Master ops (last minute)	325 Ops/s	533 Ops/s
Master ops (last hour)	381 Ops/s	518 Ops/s
Master ops (since restart)	202 Ops/s	347 Ops/s

Cluster A : Research & Development

- Smaller files and closely managed

Cluster B: Production Data Procession

- Larger files with lots of dead files

Usage Experiences and Conclusion

- A few **issues** originated with disk incompatibility with the linux drivers and need to control the extent of effect on client's operation has on the rest of the users.
- Overall designed a large-scale distributed file system that treats hardware failures as the norm and works around this issue with a novel repair mechanism, while providing high throughput to users.