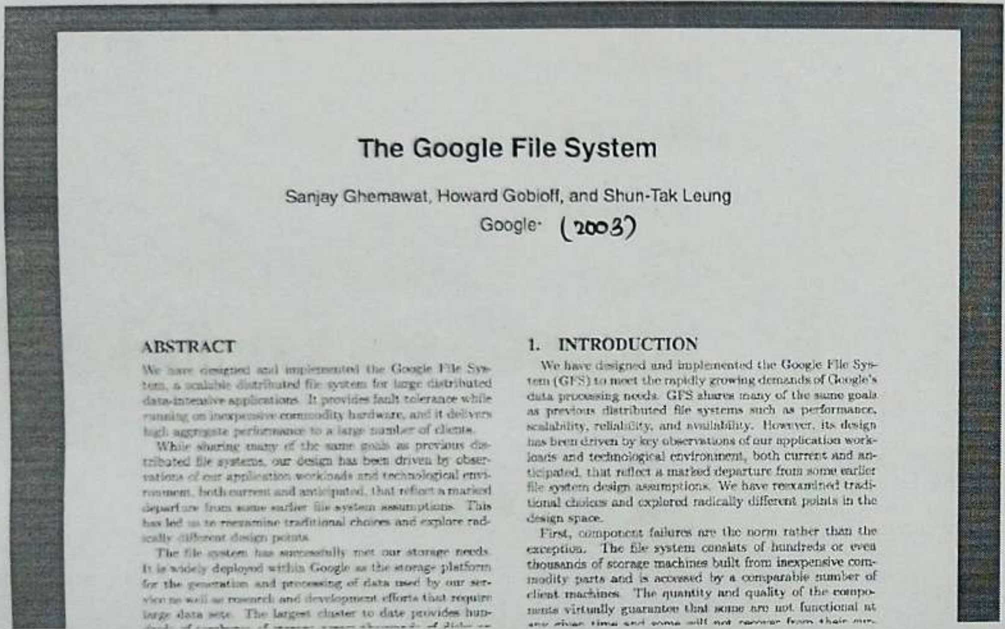
GFS is a distributed file system designed by Google ...

- It helps Google store and process web data efficiently



The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung
Google (2003)

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hun-

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at

Paper describing
Google File System

hadoop

Hadoop is an open source software framework that is used for storing and processing large amounts of data in a distributed computing environment.

Used as a basis to design Hadoop → HDFS, YARN

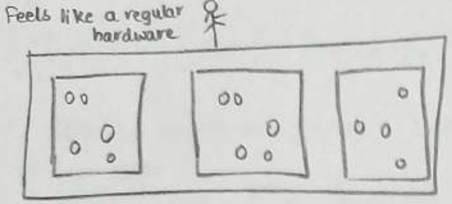
It is designed to handle BigData and is based on Map Reduce programming model, which allows for the parallel processing of large datasets.

Why was GFS needed?

① Massive Scale: Google deals with petabytes of data (1 PB = 10^3 TB)
Traditional file systems couldn't handle this

② Fault Tolerance: Hardware failures (disk crashes, machine failures) are common at Google's scale, so GFS needed to recover data automatically.

③ High Throughput: Google needed fast read/write operations to process search indexes, logs and big datasets efficiently



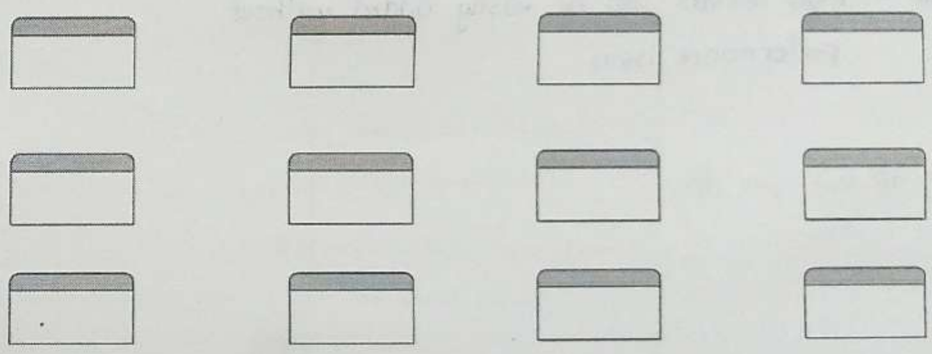
Data is stored across thousands of disks on thousands of machines → commodity hardware (no specialised hardware requirement)

Commodity Hardware

Concurrency accessed by 1000s of clients

Failures are common

- disk / network / server
- OS bugs
- human errors (Mistakes in system administration can lead to data loss)



① Disk failure : Hardware wear out over time, leading to data loss

② Network failure : Connectivity issues can prevent machines from communicating

③ Server Failures : Machines crash due to power outages, overheating, etc

Commodity servers are cheap & can be made to scale horizontally with right software

(adding more servers) instead of buying few POWERFUL machines

INSTEAD OF PREVENTING FAILURES, GFS is designed to recover from them automatically.

What is Commodity Hardware?

* Commodity Hardware refere to cheap, off-the-shelf servers instead of high-end, expensive machines

* These servers are not super-reliable and may often fail

Why use Commodity Hardware?

Cost Effective - Buying thousands of cheap machines is cheaper than a few expensive ones

Scalable - More servers can be easily added without performance issue

→ one failure per day

① Google's Search Engine ~~as~~ stores copies of billions of web pages, which require massive storage

② Google analyzes server logs in bulk to detect issues or optimize performance.

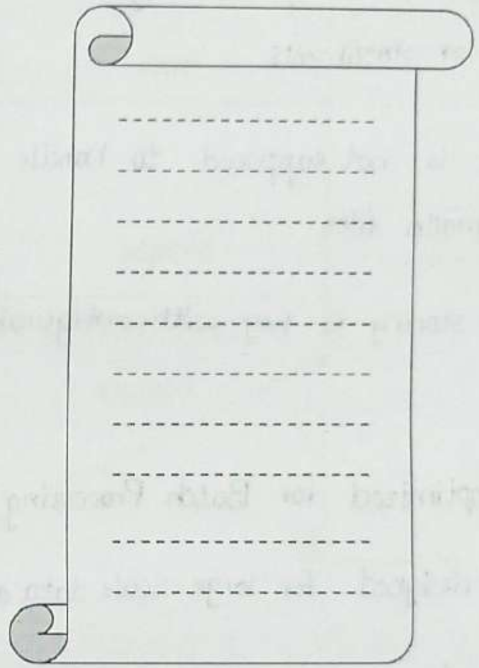
③ Google uses **MapReduce**, which breaks big problems into smaller tasks and process them in parallel.

Large files

100MB to multi-GB files

- Crawled web documents
- Batch processing

processing large amounts of data in groups (batches) instead of handling each request one by one in real time



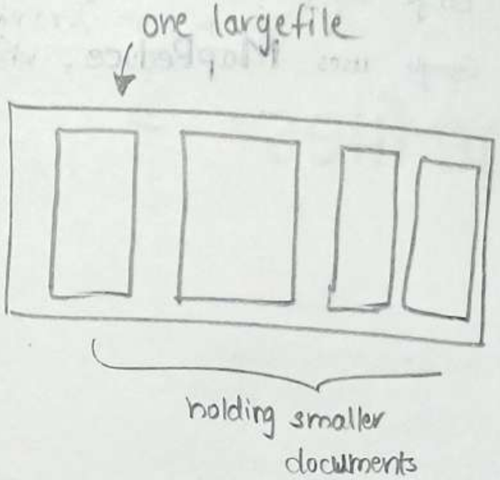
Commodity servers are <u>cheap</u> & can be made to scale horizontally with right software

② HUGE FILES :

↳ Multi-GB files are common

Each file may internally hold multiple files or documents.

↳ FS is not supposed to handle billions of smaller files



Hence, chunking & storing is key with configurable block size

Why is GFS optimized for Batch Processing?

Since GFS was designed for large-scale data analysis, it prioritizes:

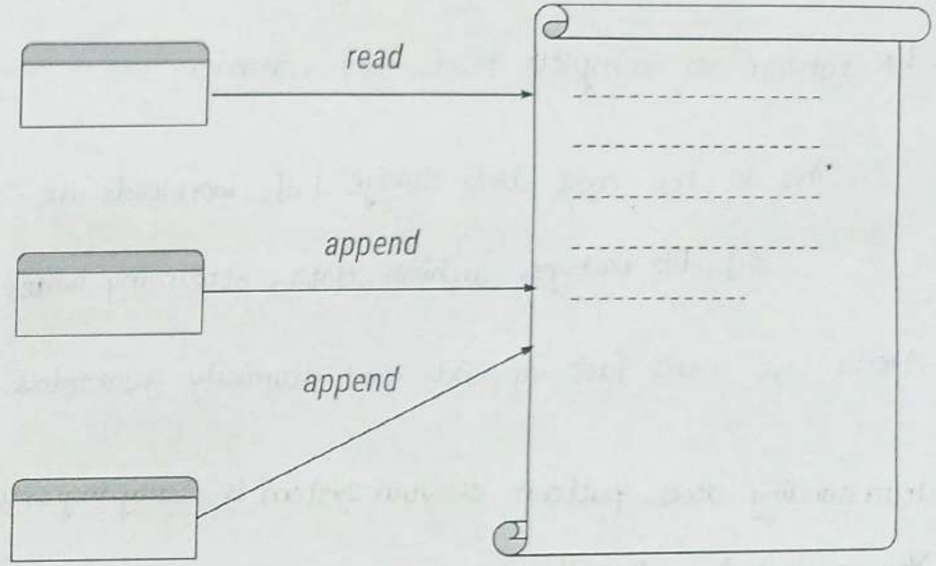
i> High Throughput (handling large data at once)

ii> Efficient sequential reads/writes (to process entire files in batches)

iii> Appending data instead of modifying it (since logs & indexing require continuous data addition)

Mutations are append & not overwrite

File Operations

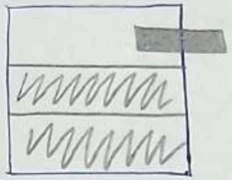


Read + Append only

- No random writes
- Mostly sequential reads

Think of web crawling - Keep appending crawled content & Use batch processing (reads) to create index

Mutations are append & not overwrite



↳ random writes within the file very infrequent

↳ alterations are append operations

↳ random or complete reads are common

This is how most blob storage | dfs workloads are

e.g DB backups, archival data, streaming writes (appends) (logs)

Hence we need fast appends and atomicity guarantees → multiple clients append to same file

* Understanding access pattern of your system is really important

* You cannot have it all

↳ design the most optimal system, under the given constraints & relaxations

Design Assumptions:

① underlying hardware is **cheap, commodity and bound to fail**

② small files are supported but No need to optimize for that

③ Large streaming reads and small random reads are common

④ **Sequential writes** append to the file

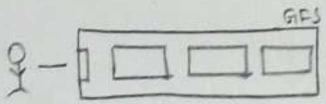
⑤ Random writes are supported but NOT optimized

⑥ **Atomicity** is essential but should be efficient

⑦ Latency NOT a concern but bandwidth

↳ for individual reads & writes

POSIX like Interface (Portable Operating System Interface) ensures compatibility & portability of software across diff. Unix-like system



Familiar file system interface;

not exactly POSIX complaint (create, delete, open, read, close and write operations)

Additional Operations :

Snapshot : Copy a file efficiently (directory tree)

Record : allows multiple clients to append concurrently without additional locking

Scanned with ACE Scanner

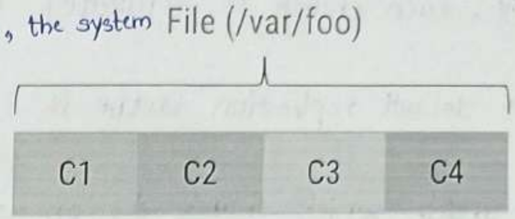
Breaking files into chunks helps in:

i> parallelism: multiple chunks can be processed simultaneously, speeding up tasks

ii> load balancing: chunks are distributed across diff servers, preventing any one server from being overloaded.

Chunks

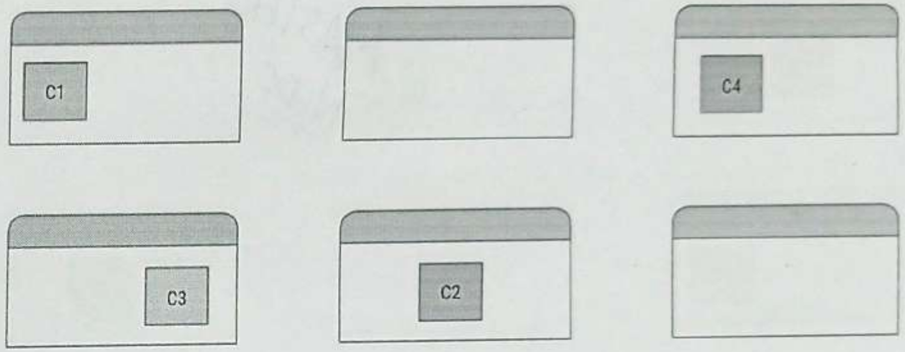
iii> Fault Tolerance: if one chunk is lost or a server fails, the system can recover from backups or replicas



iv> Easy storage allocation: instead of storing large files in one place, they are split into smaller parts, making storage management easier.

Files split into chunks

- Each chunk of 64MB
- Identified by 64 bit ID
- Stored in Chunkservers



Chunkservers - Chunks of single file are distributed on multiple machines

v> Easy Migration: Chunks can be moved b/w servers without affecting the entire file, making system upgrades or maintainance smoother

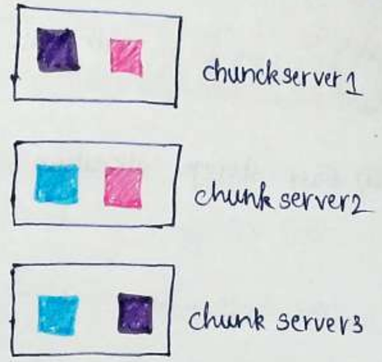
vi> Reduced Network Overhead: Only necessary chunks are accessed or transferred, reducing unnecessary data movement & improving efficiency

vii> Smaller Metadata on "Master": Instead of tracking full files, the master node only tracks chunk locations, making metadata smaller & easier to manage.

Thus, the file is ~~stored~~ chunked and stored across the cluster. The machines where these chunks are stored are called <u>chunk servers.</u>

For reliability, each chunk is replicated across multiple chunk servers.

↳ default replication factor is 3



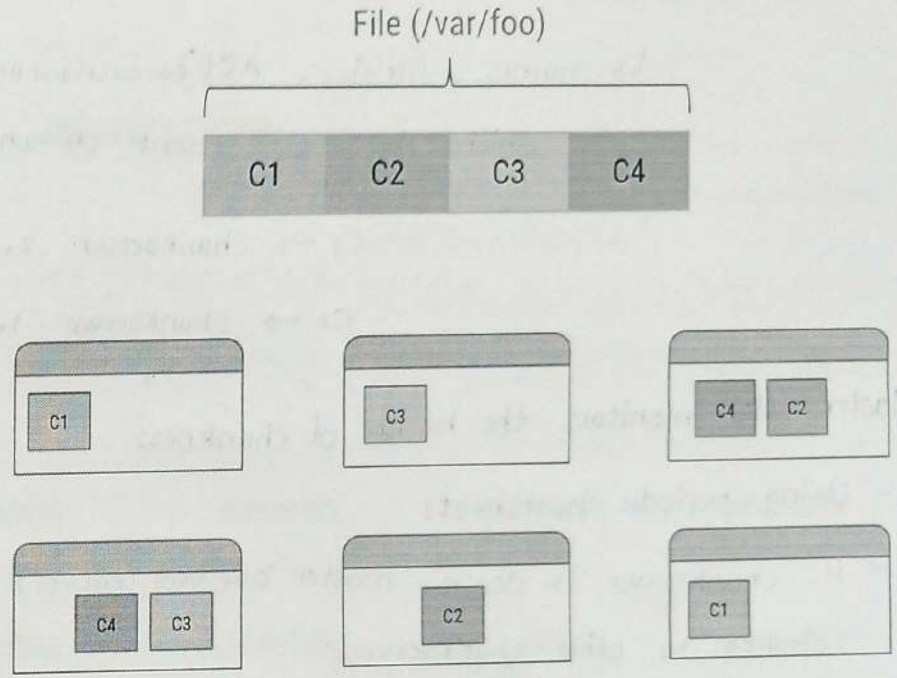
↳ replication factor = "2"

How do we decide which chunk lies on which server?

↳ we need a brain

MASTER NODE

Replicas



Files split into chunks

- Replica count by client
- commodity server failures

Replicas ensure durability of data if chunkserver goes down

① fault tolerance
② data availability
③ load balancing

Replica count by client

In systems like GFS, the master usually decides the no. of replicas based on factors like:

- File type (critical files may have more replicas)
- Ready-heavy or write-heavy usage patterns
- System policies for redundancy

Clients do not directly control the replica count but may specify replication needs in some cases.

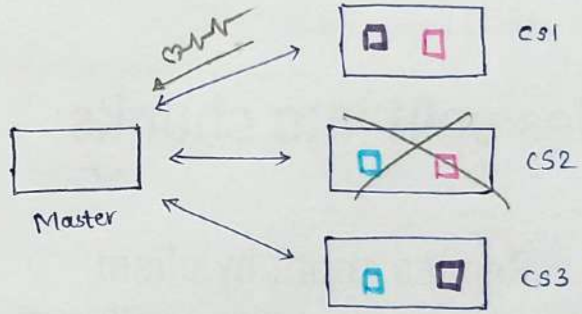
MASTER NODE

The metadata about the file and the chunks is stored on the Master

↳ names, filesize, ACL (access control list), chunk servers and which chunk is present on which chunk servers

$C_1 \rightarrow$ chunkserver 2, 3 and 9

$C_2 \rightarrow$ chunkserver 1, 9 and 18



Since replication factor is 2
Now pink, blue is gone

from CS1 copy pink and paste in CS3

from CS3 copy blue and paste in CS1

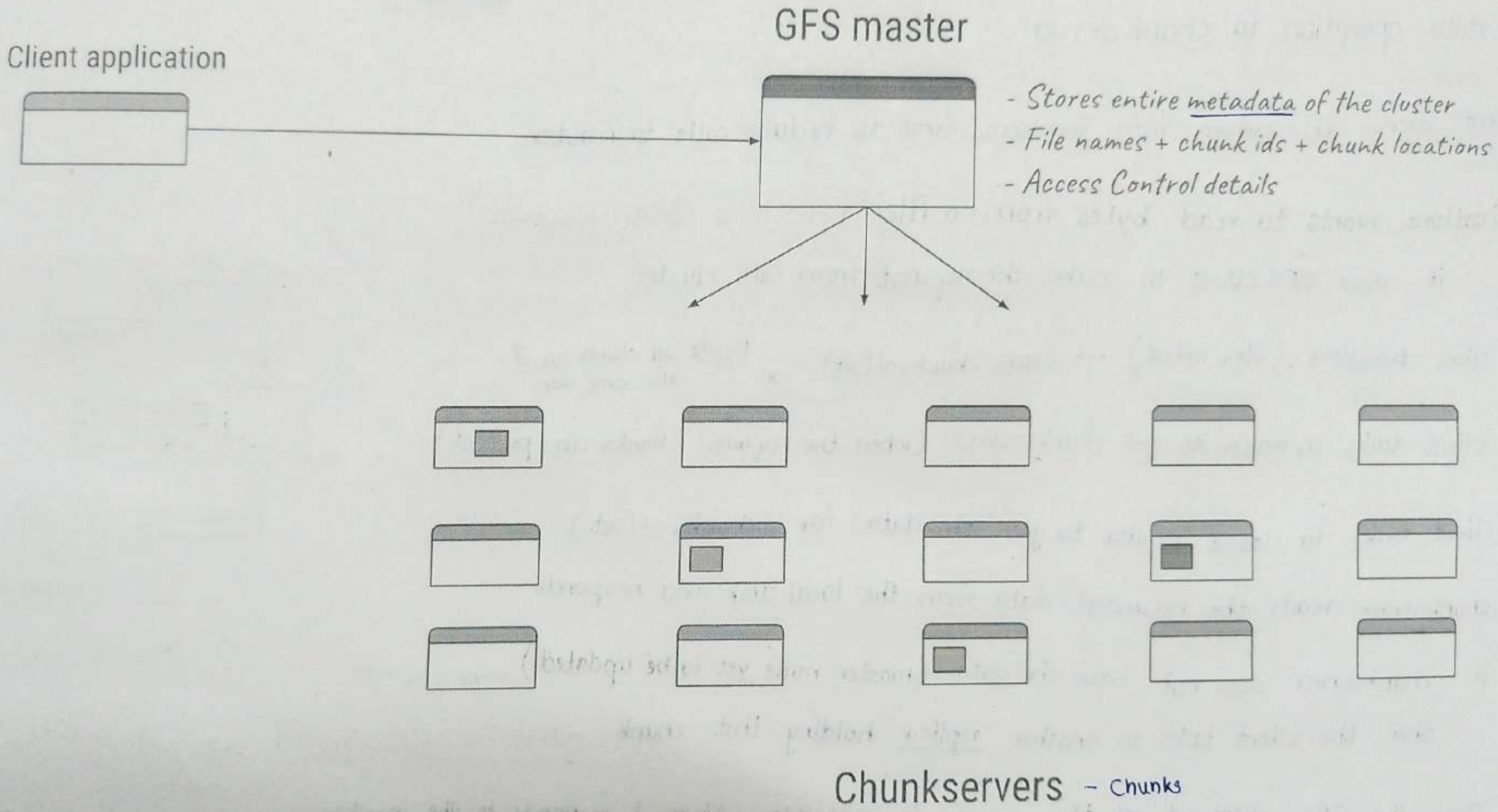
* Master also monitors the health of chunkness
  - Using periodic heartbeats
  - if chunkserver is down, master balances (moves) chunks to other chunkservers

* Master maintains ACL for each file / namespace

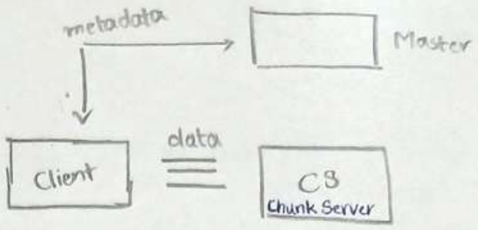
  Ensures only that should, would access the files

  ↳ all the request read/write goes the master

* Master assigns a 64 bit unique id to each chunk


GFS master
Client application
- Stores entire metadata of the cluster
- File names + chunk ids + chunk locations
- Access Control details
Chunkservers
- Chunks

Typical Request Flow



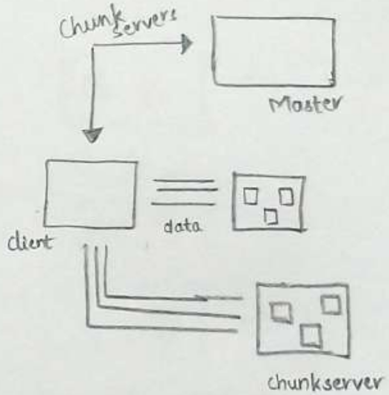
↳ metadata operations go to master

↳ data operation to chunk server

* client caches chunkserver info for some time to reduce calls to master

Applications wants to read bytes from a file (Flow of a READ Request)

it uses GFS client to make the request from GFS cluster



① client translates (file, offset) → (file, chunk, offset) Recall: all chunks are of the same size

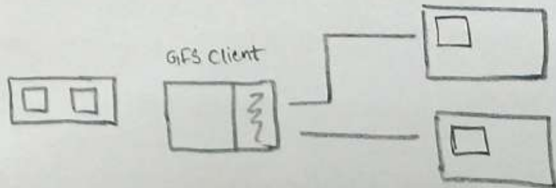
② client talks to master to get chunkservers (when the required chunks are present)

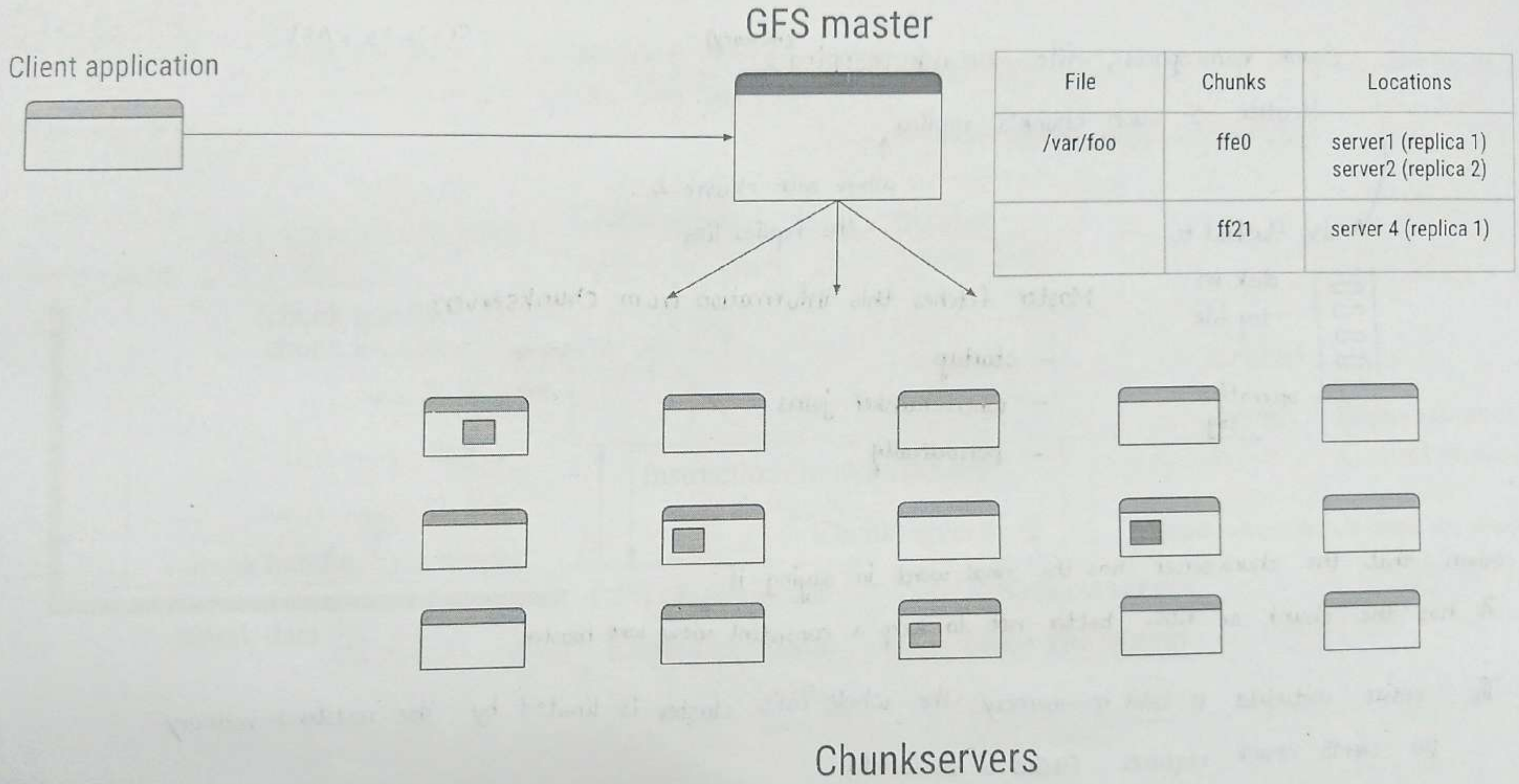
③ client talks to closest replica to get the data for (chunk, offset)

④ chunkserver reads the requested data from the local disk and responds

⑤ if chunk server does not have the data (master node yet to be updated) (maybe it deleted)
then the client talks to another replica holding that chunk

⑥ Once the GFS client got all the chunks it assembles them & responds to the invoker





File	Chunks	Locations
/var/foo	ffe0	server1 (replica 1) server2 (replica 2)
	ff21	server 4 (replica 1)

Where and how the Master stores the metadata?

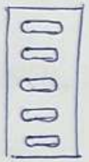
The master node stores metadata in-memory

chunk namespaces, file → chunk mapping, (memory)

location of each chunk's replica

where each chunk & the replica lies

also flushed to



disk in log file

operation log

↪ prefix compressed

/path/file/a.txt

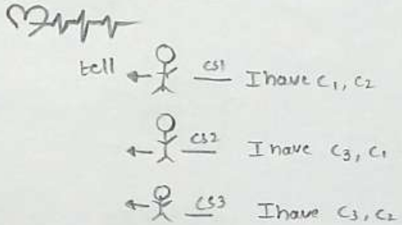
↳ size, last modified at,

c_1, c_2 c_3, ACL

} optimizations

Master fetches this information from chunkservers

- startup
- when chunker joins
- periodically



Given that the chunkserver has the final word in saying if it has the chunk or not, better not to keep a consistent view wrt master

The entire metadeta is held in-memory the whole GFS cluster is limited by the master's memory

per 64MB chunk requires 64GB of metadata

1GB of metadata could hold data about

10^6 GB = 1000 TB = 1 PB !! (storing data is afterall not that bad)

Scanned with ACE Scanner

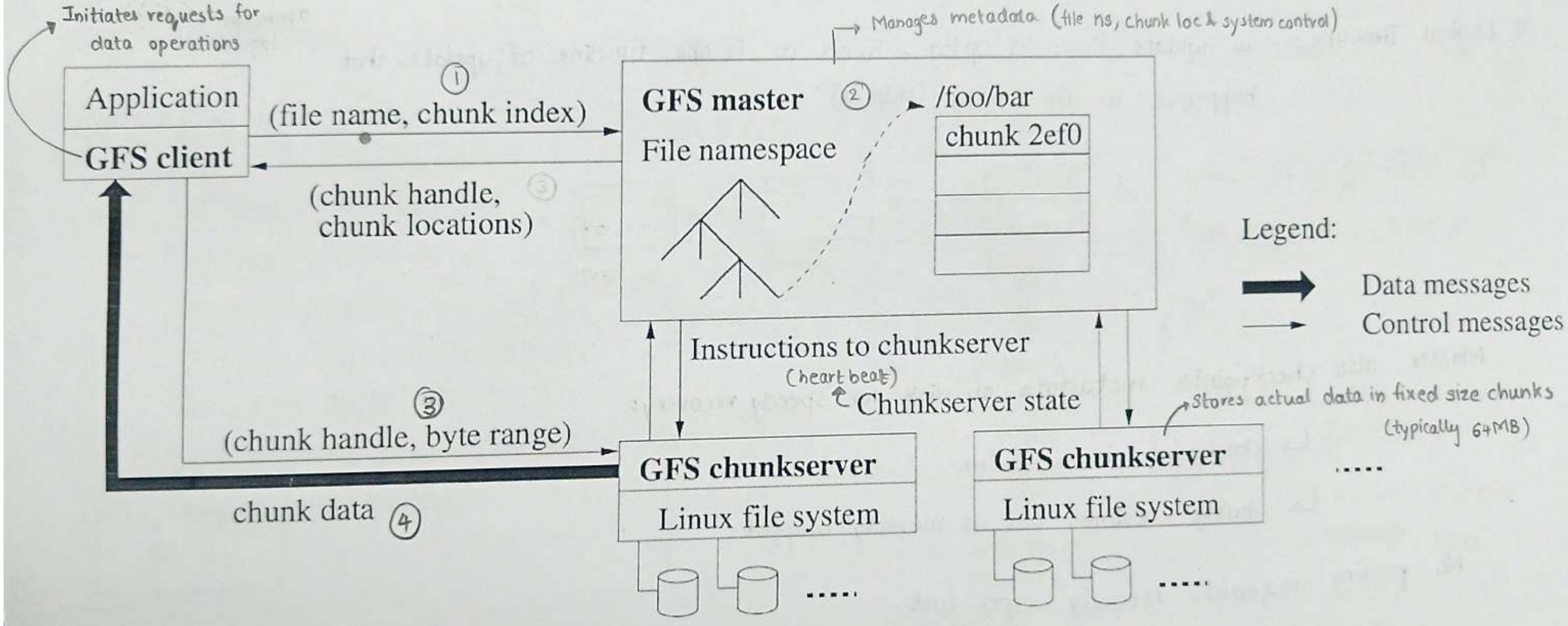
Workflow

① CLIENT REQUEST : The GFS Client sends a request to the GFS Master to retrieve the location of a specific chunk of a file

② CHUNK LOCATION INFO : The GFS Master responds with the chunk handle (a unique ID) and a list of chunkserver loc. that store the requested data

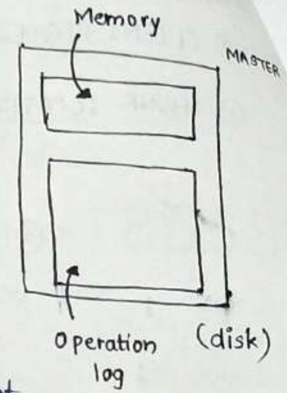
Reads

③ DATA REQUEST : The GFS Client directly communicates with the GFS Chunkserver using the chunk handle and
(chunk handle/byte range) specifies the byte range needed.



④ DATA TRANSFER (chunk Data) : The GFS Chunkserver transfers the required data chunk directly to the client, improving efficiency.

OPERATION LOG and CHECKPOINTING

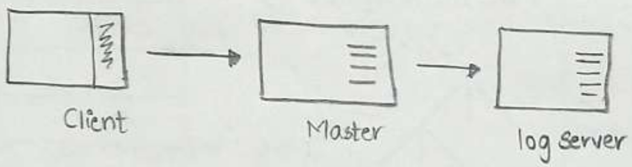


All updates to the file system are applied to the log before updating the in-memory metadata

1) Recovery: recover any operation that were in progress (replay)

2) Logical Timeline: all update flow in oplog, hence it is one timeline of updates that happened in the cluster (ordered)

Operation log is replicated remotely on 'log server'

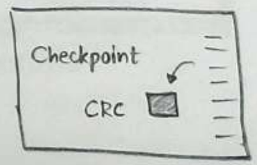


(log)

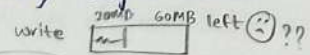
Master also checkpoints metadata on disk for speedy recovery:

↳ Checkpoint is a - B tree

↳ during restore, tree is memory mapped



No passing required, recovery super fast.



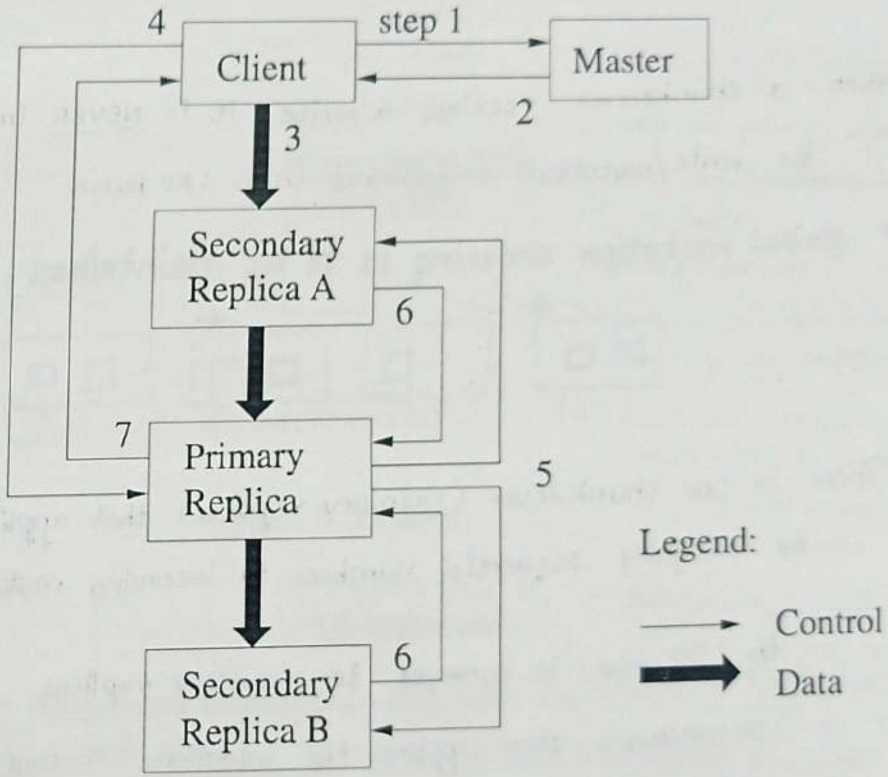
↗ verification code

* Uses checksums to validate integrity of the checkpoint → Checkpoint is created through an async thread to not block incoming mutations

* corrupt checkpoints are discarded

Writes

1. Ask for locations to write
2. Get replicate locations
3. Write data to closest replica.
4. Request commit to primary
5. Primary instructs order of writes to secondaries
6. Secondaries acknowledge
7. Primary ack to client

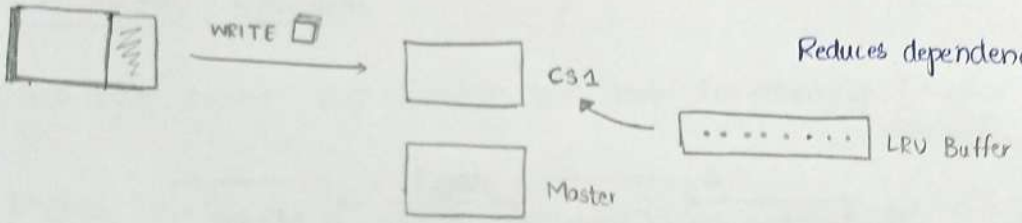


Scanned with ACE Scanner

Writes, LRU Buffer and Lease Management

Least Recently Used

Chunk server receives the writes from client directly (not master) to reduce no. of hops



Reduces dependency on master node

When a chunkserver receives a write, it is NEVER immediately applied

The write/mutation is buffered in a LRU Buffer

* **global mutation ordering is to be maintained, all replicas consistent**

one CS can be primary replica for multiple chunks as well

For a chunk: There is one chunkserver (primary replica) that applies the changes & inform others by assigning sequential numbers to incoming mutations

- the sequence is conveyed to secondary replicas
- Secondaries then applies the mutations in that order and reply to primary

But how do we decide, which REPLICA is primary?

Lease Management

Each chunk is replicated across cluster with some replication factor, say 3.

When the write is initiated to which replica does the write go?

↳ all of them? too slow & prone to fail

↳ any one at random? unpredictable

↳ one of them? (deterministic)

Master grants chunk lease to one of the replicas - Primary Replica for handling write operations

Factors Influencing the selection:

1. Replica Location: Chooses a replica close to the client to min. network latency
2. Load Balancing: prefers replicas on less-loaded machines or racks to distribute traffic evenly
3. Availability: Selects a replica on a healthy and responsive server
4. Data Freshness: Chooses the most up-to-date replica to avoid inconsistencies

The lease has an expiry, but primary can continue to extend it. If primary dies, the lease is attached to some other replica

↳ all these communication happens over **HEARTBEAT** messages.

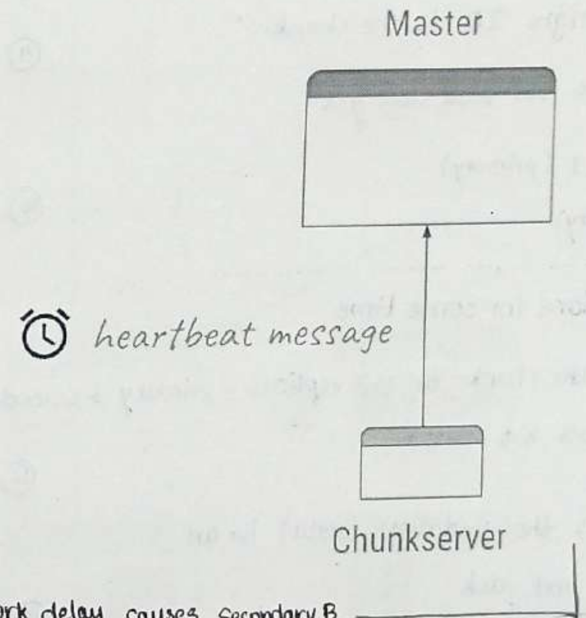
Heartbeats

What about two updates (mutations) are recieved for the same chunk?

How will we ensure the correct order of operation?

(Lease & Lazy Apply)

Regular heartbeats to ensure chunkservers are alive



① Lease Mechanism

Only the primary with the lease can serialize writes for the chunk

How it ensures order:

- The primary assigns a sequence number to every mutation
- Even if two clients send updates simultaneously, the primary decides the order based on when it receives the requests
- Secondary replicas apply mutations strictly in the sequence order received from the primary

For example, Client A and Client B send updates to the same chunk.

- Primary receives Client A's mutation first → Assign sequence number #1001
- Client B's mutation comes next → Assigned seq no. #1002
- All replicas apply #1001 before #1002 ensuring correct order

② Lazy Apply (Asynchronous Update Propogation)

- Secondary replicas data asynchronously from the primary
- They apply updates after receiving them, but only in the order assigned by the primary.
- They acknowledge each mutation back to the primary only after successful application.

For example, Primary replica commits an update (sequence #1001) → Sends to secondaries

- A network delay causes Secondary B to recieve it later than Secondary A
- Secondary B still applies the earlier mutation first (due to the seq number) before moving to the next update

FLOW of a WRITE REQUEST

① Client sends the write through GFS client lib

② GFS client splits the write into chunks (64MB each)

③ GFS client talks to Master to get chunk servers

↳ Primary Replica and secondary nodes

↳ Master assigns ID to the chunk

* Master does not update its local state yet

chunk 1 → CS 1 (primary)

CS2 (secondary)

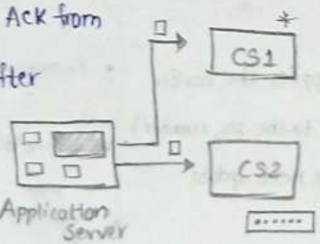
④ Client caches the locations for some time

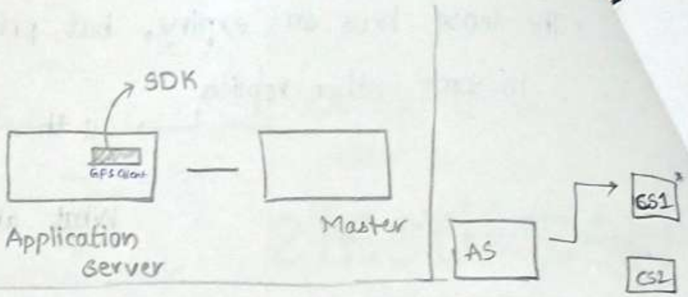
⑤ GFS client writes the chunk to the replicas - primary & secondary

* order does not matter

⑥ Each chunkserver, holds the mutation (data) in an internal LRU buffer and ack

⑦ The client waits to receive ACK from all replicas. But it proceeds after receiving ACK from majority.

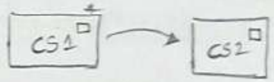




⑧ Once ACK is received from all (majority) client sends WRITE request to primary replica

⑨ Primary replica assigns serial number to this mutation and applies changes to its own state

⑩ Primary replica forwards the WRITE request to all secondary replicas



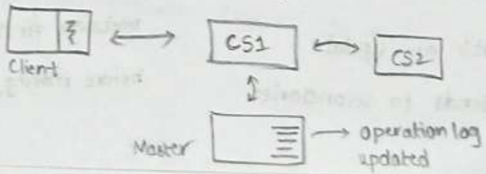
⑪ Secondary ACK primary suggesting completion of operation

⑫ Primary replica now tells the master about the mutation

↳ mutation, chunk, location, version no.

⑬ Master node now writes this mutation in OPERATION LOG and ACK primary replica

⑭ Now primary replies to the client

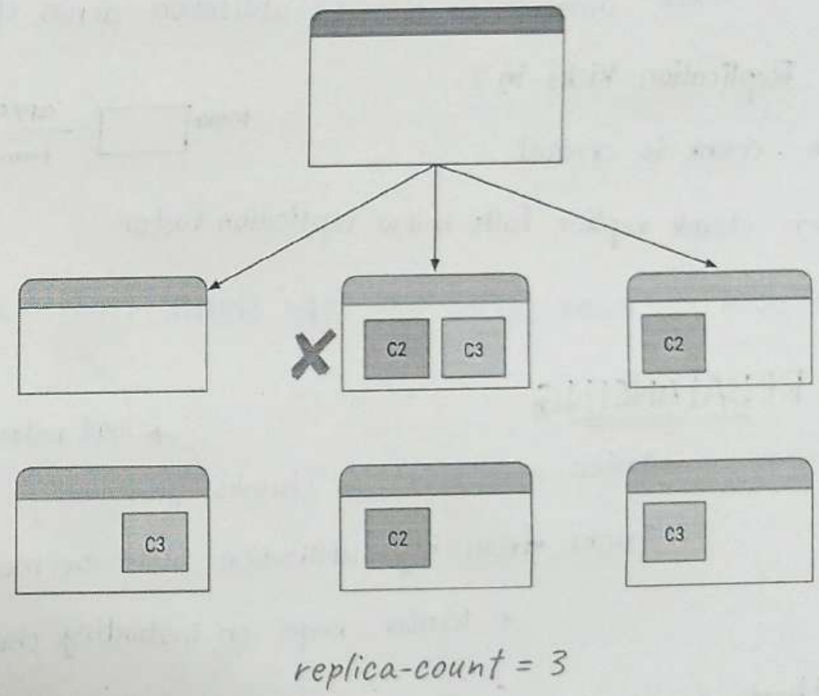


Most errors are retryable and after some attempts, the entire write is retried.

Ensure chunk replica count

If chunkserver is down, master ensures all chunks that were on it are copied on other servers.

Ensures replica counts remains same.

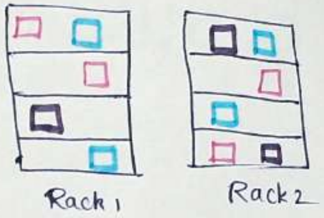


CHUNK DISTRIBUTION :

Chunks are replicated and distributed across cluster, such that

- maximize data reliability
- maximize network bandwidth utilization

Just distributing is not enough we need to consider



Hence chunks are spread across Racks

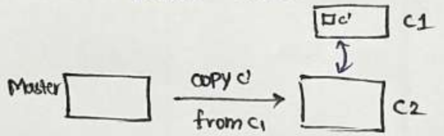
↳ utilizes rack's bandwidth → load not on just one rack

↳ fault tolerance across racks

* cross RACK writes are costly, but okay

↳ have almost equal disk utilization across cluster

When Replication kicks in?



① When chunk is created

② When chunk replica falls below replication factor

Master tells a chunk server to copy chunk from a replica (specific)

CHUNK REBALANCING

Master rebalances the chunks periodically → load balancing

Chunks from high utilization node are moved to low one

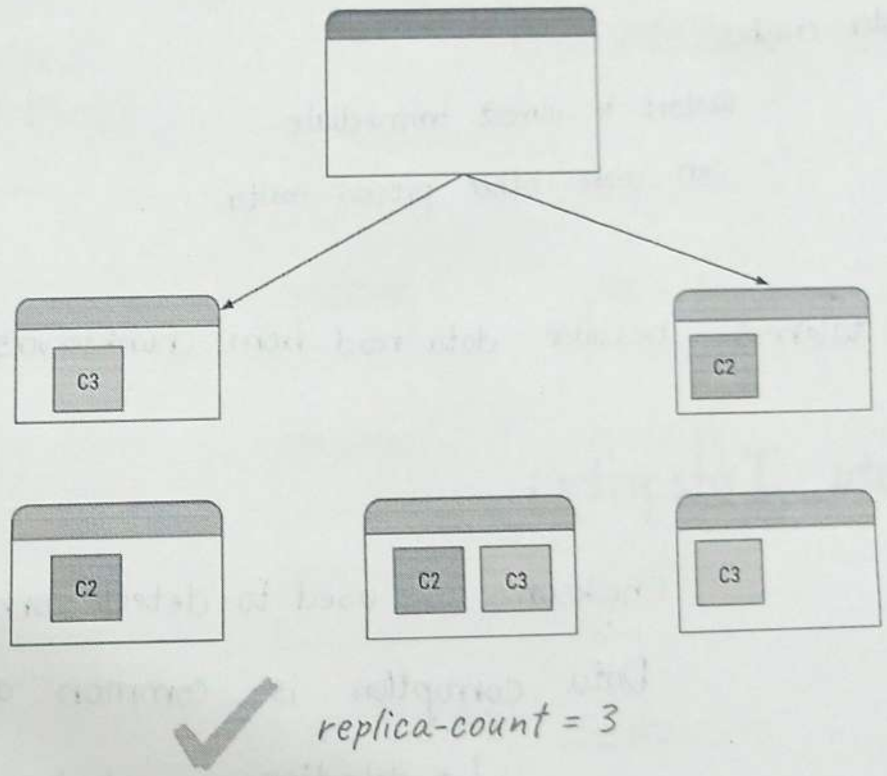
* Master keeps on instructing chunk servers

Heartbeat from chunkservers tell master about the chunks

Ensure chunk replica count

If chunkserver is down, master ensures all chunks that were on it are copied on other servers.

Ensures replica counts remains same.



High Availability:

1. Recovery is fast because of checkpoint on BTree & memory mapped load
2. Chunks are replicated and stored,
   any node going down does not affect availability
3. Master state is replicated
   Operation logs ~~patchs~~ and checkpoints are replicated on multiple machines
4. Master crashes
   - Restart is almost immediate
   - can make other process master

DNS: master. gfs. cluster
↳ ~~10.0.0.7~~
↳ 10.0.0.9

5. No staleness because data read from chunkservers

Data Integrity:

Checksums are used to detect corruptions

Data corruption is common and two phase to handle them

↳ detection: → checksums

↳ correction: → replicas of chunk will help in recovery

64MB Chunk → 64KB block
↳ 32 bit checksum of each block is tracked

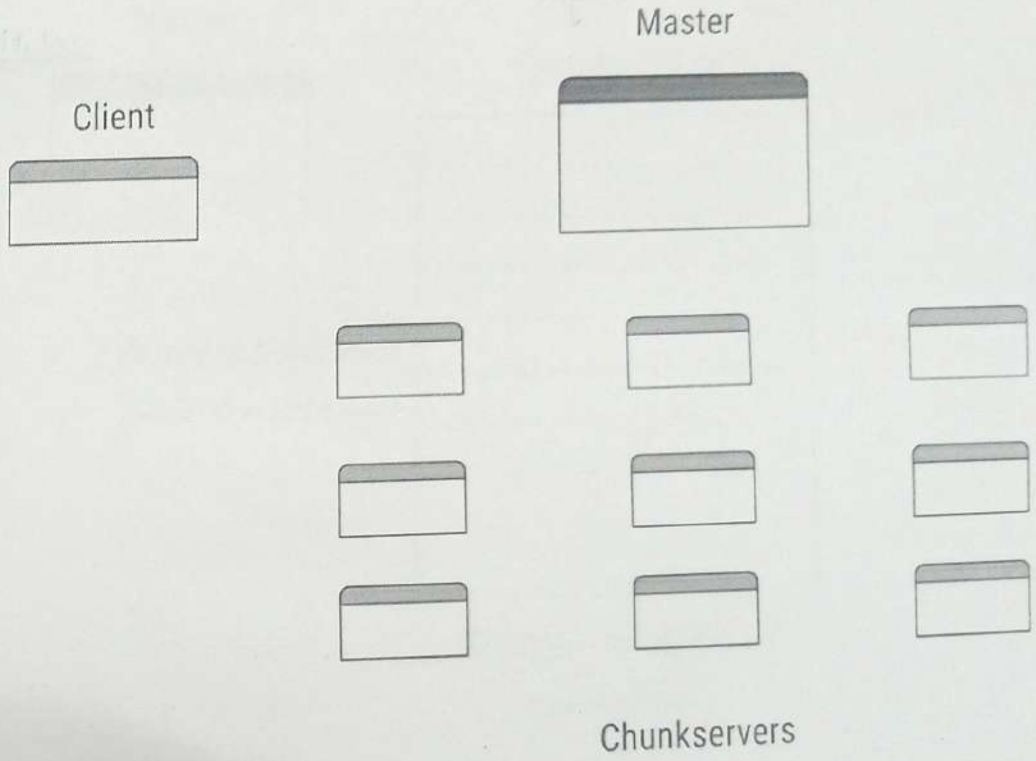
* **Check sums** are checked & verified during reads and corruptions are not propogated

Scanned with ACE Scanner

Single master for multi-TB cluster

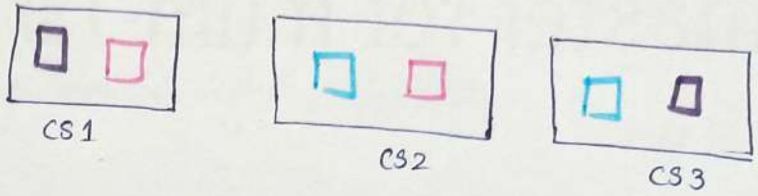
Large chunk size

- 64MB chunk
- Reduced meta-data
- Reduces client interactions
- Client caches location data



HANDLING HOT SPOTS

if one of the chunk is heavily accessed, the chunkserver becomes a hot spot and fragile



Solution: for such data increase the Replication Factor

Operations Log

Record of all ops

- Checkpointed regularly
- Happens in background thread
- Used if master crashes
- Rebooted master replays log