Distributed File Systems (DFS)

(Big Data Analytics)

Distributed File System (DFS)

Local File System	Distributed File System
Manages files on a single machine	Manages files on a multiple machines
LFS uses Tree format to store Data.	Provides Master-Slave architecture or Cluster based architecture for Data storage.
Generally store the data as single block in single machine	Store the data in multiple blocks in multiple machines
It is not reliable because LFS data does not replicate the Data files.	It is reliable because in DFS data blocks are replicated into different DataNodes.
Files can be accessed directly in LFS.	Files can not be accessed directly in DFS because the actual location of data blocks are only known by NameNode.
LFS is cheaper because it does not needs extra memory for storing any data file.	DFS is expensive because it needs extra memory to replicate the same data blocks.
LFS is not appropriate for analysis of very big file of data because it needs large time to process.	DFS is appropriate for analysis of big file of data because it needs less amount of time to process as compare to Local file system.

Distributed File System (DFS)

- DFS differs from typical file systems (i.e., FAT32,FAT64, NTFS) in that it allows direct host access to the same file data from multiple locations. Indeed, the data behind a DFS can reside in a different location from all of the hosts that access it.
- A Distributed File System (DFS) as the name suggests, is a file system that is distributed on multiple file servers or multiple locations.
- Since more than one client may access the same data simultaneously, the server must have a mechanism in place (such as maintaining information about the times of access) to organize updates so that the client always receives the most current version of data and that data conflicts do not arise.
- Distributed file systems typically use file or database replication (distributing copies of data on multiple servers) to protect against data access failures.
- Examples of distributed file systems:
    - Sun Microsystems' Network File System (NFS),
    - Novell NetWare,
    - Microsoft's Distributed File System,
    - Google File System

Design Goals

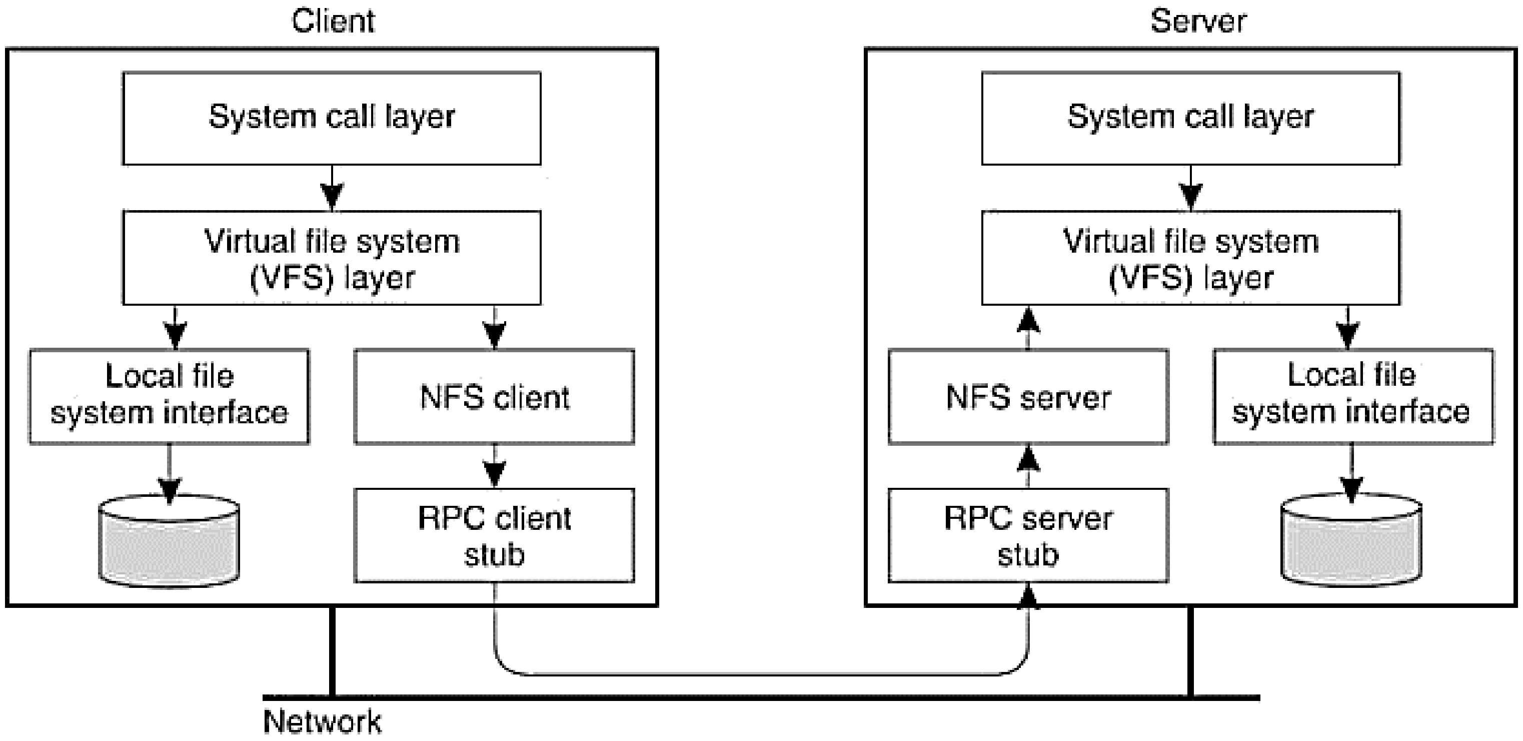
A DFS should provide:

- **Network transparency:**
    - Hide the details of *where* a file is located.
- **High availability:**
    - Ease of accessibility irrespective of the physical location of the file.
- **Scalability:**
    - Number of users, Servers, and files handled etc.
- **Concurrency:**
    - Handle concurrent access by different clients in the network

DFS Architecture

Client-Server based Architecture (Example: Network File System (NFS))

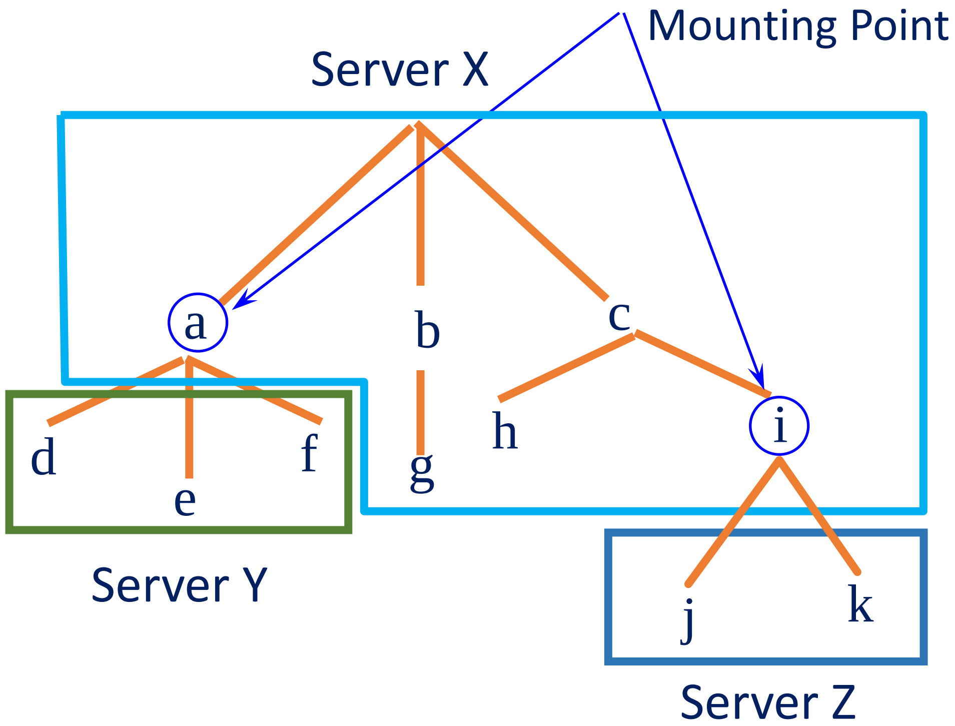
- Client-Server is one of the common architecture in DFS.
- Sun Microsystem's Network File System (NFS) being one of the most widely-deployed.
- NFS has been implemented for a large number of different operating systems.


Client
System call layer
Virtual file system (VFS) layer
Local file system interface
NFS client
RPC client stub
Server
System call layer
Virtual file system (VFS) layer
NFS server
Local file system interface
RPC server stub
Network

The basic NFS architecture for LINUX/UNIX systems

Distributed File System (DFS)

- **DFS:** It is a file system that is distributed on multiple file servers or multiple locations.
- **Mechanisms for building DFS:** (Mounting, Caching, Hints, Bulk transfer, and Security)



Name: File or directories
Name resolution: process of mapping the Name to Physical Storage
Namespace: Collection of names (files/Directories)

Mounting: Mounting makes file systems, files, directories, and devices available for use at a particular location. It is the only way a file system is made accessible. It allows the binding together of different file namespaces to form a single file system structure (Single hierarchical namespace).

Caching: When files are in different servers, caching might be needed to improve the response time. A copy of data (in files) is brought to the client (when referenced). Subsequent data accesses are made on the client cache. (Client cache and server cache)

Hints: Data must be consistent in the cached memory. An alternative approach to maintaining consistency is to treat cached data as hints. A time-stamp based method is used for validate cache block before they are used.

Bulk data transfer: Helps in reducing the delay due to transfer of files over the network. Obtain multiple number of blocks with a single seek

Encryption: Establish a key for encryption with the help of an authentication server.

Mechanisms for DFS

Caching:

- Performance of distributed file system, in terms of response time, depends on the ability to “get” the files to the user.
- When files are in different servers, caching might be needed to improve the response time.
- A copy of data (in files) is brought to the client (when referenced). Subsequent data accesses are made on the client cache.
- Client cache can be on disk or main memory.
- Data cached may include future blocks that may be referenced too.
- Caching implies DFS needs to guarantee consistency of data.

Mechanisms for DFS

Hints:

- Caching is an important mechanism to improve the performance of a file system. But for guaranteeing the consistency of the cached items requires elaborate and expensive client/server protocols.
- An alternative approach to maintaining consistency is to treat cached data as hints.
- A time-stamp based method is used for validate cache block before they are used.
- With this scheme, cached data is not expected to be completely consistent, but when it is, it can dramatically improve performance.
- For maximum performance benefit, a hint should nearly always be correct.

Mechanisms for DFS

Bulk data transfer:

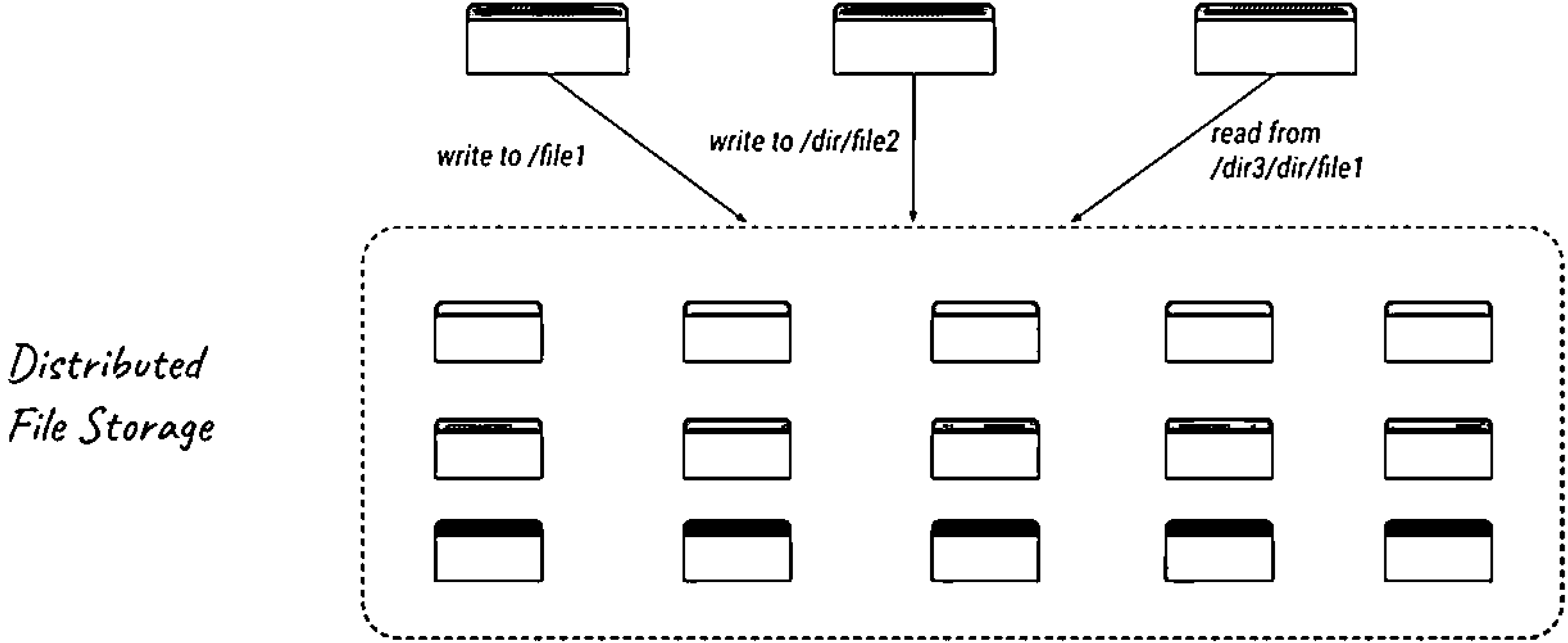
- helps in reducing the delay due to transfer of files over the network.
- *Bulk*:
  - Obtain multiple number of blocks with a single seek
  - Format, transfer large number of packets in a single context switch.
  - Reduce the number of acknowledgements to be sent.

- **Encryption:**
  - Establish a key for encryption with the help of an authentication server.

Google File System (GFS) (Cluster based Architecture)

- Google uses the GFS to organize and manipulate huge files and to allow application developers the research and development resources they require.
- The GFS is unique to Google and is not for sale.
- Google doesn't reveal how many computers it uses to operate the GFS. In official Google papers, they only says that there are "thousands" of computers in the system.
- GFS:
  - Support and deal with large files.
  - Scalable
  - Use the general file commands: Open, Create, Read, Write and Close.
  - Also support specialized command: Append and Snapshot
- File is divided into number of “Chunks” and each one is associated “Chunk handle”
- GFS balance the workload of the chunks.

Google File System (GFS)



1 cluster = 100s of commodity servers

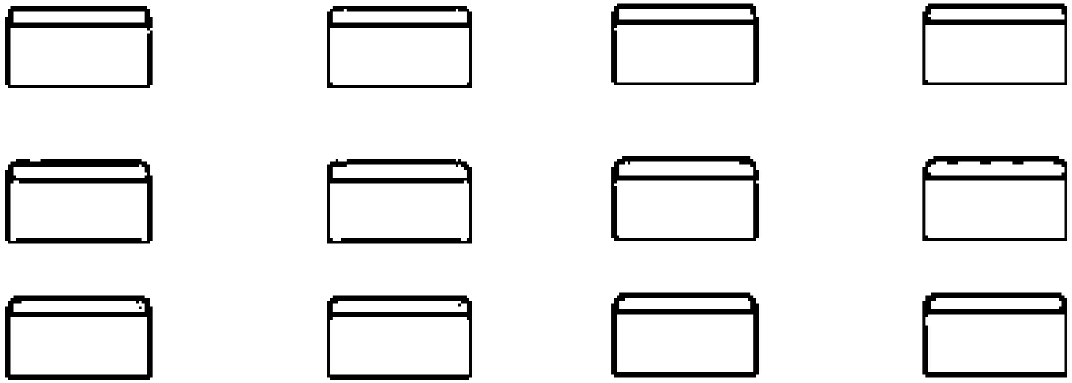
Google File System (GFS) – Design Consideration

Commodity Hardware

Failures are common

- disk / network / server
- OS bugs
- human errors

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

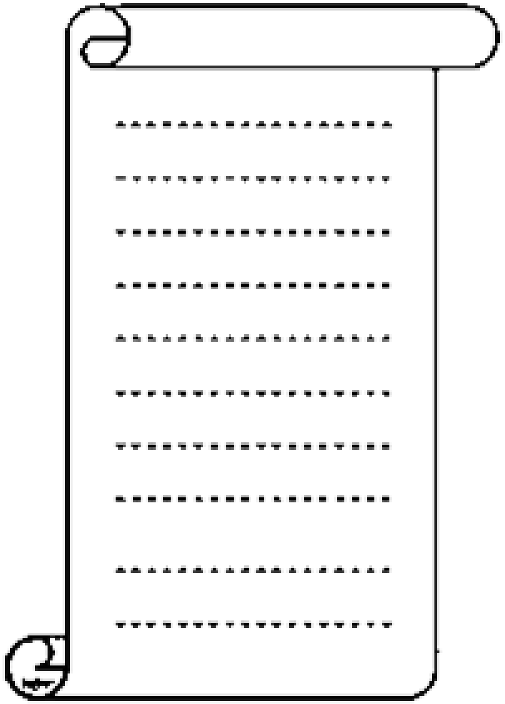


Google File System (GFS) – Design Consideration

Large files

100MB to multi-GB files

- Crawled web documents
- Batch processing

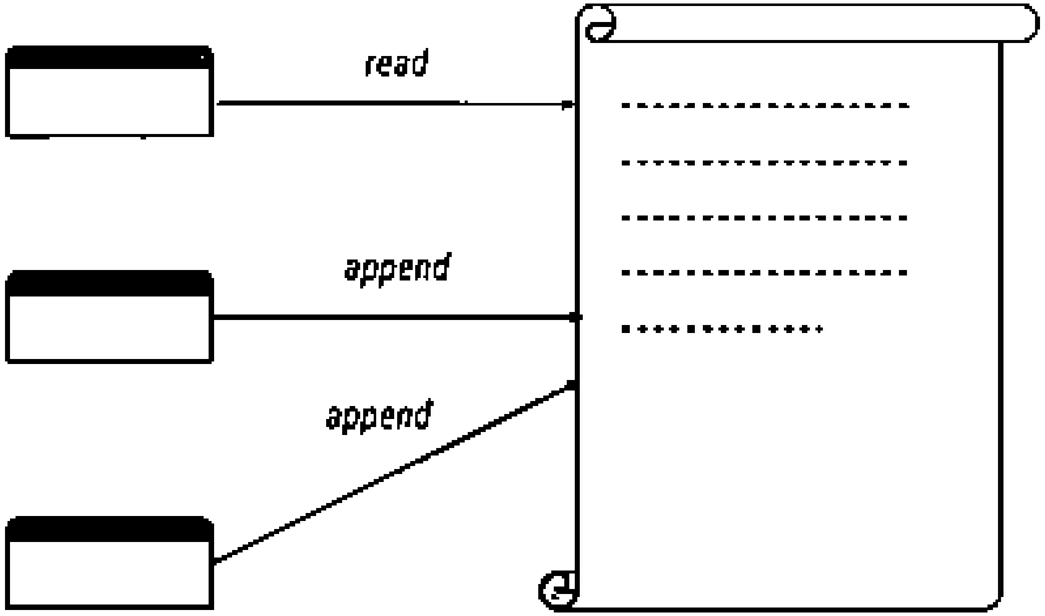


Google File System (GFS) – Design Consideration

File Operations

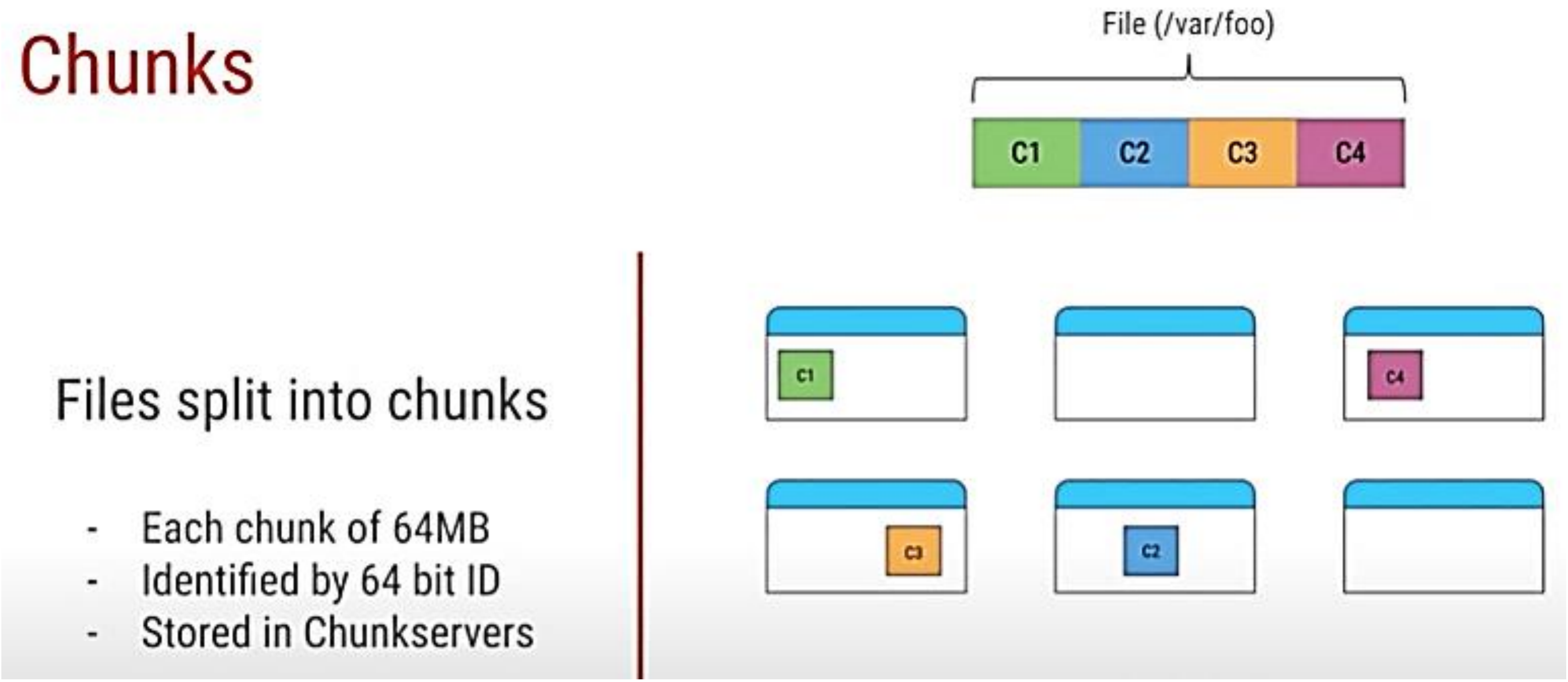
Read + Append only

- No random writes
- Mostly sequential reads



Think of web crawling – keep appending crawled Content & use batch processing (Reads) to create index

Google File System (GFS) – Design Consideration



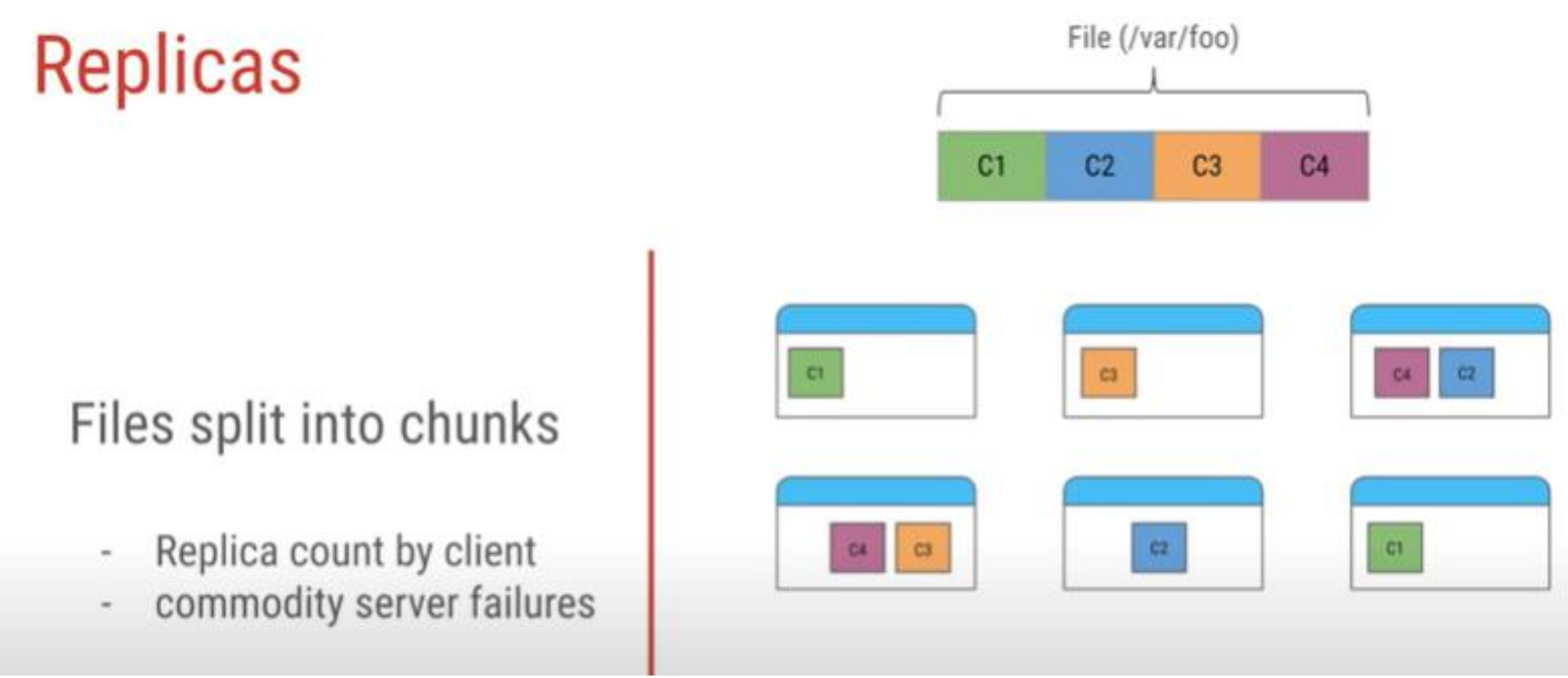
Chunks

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

Google File System (GFS) – Design Consideration



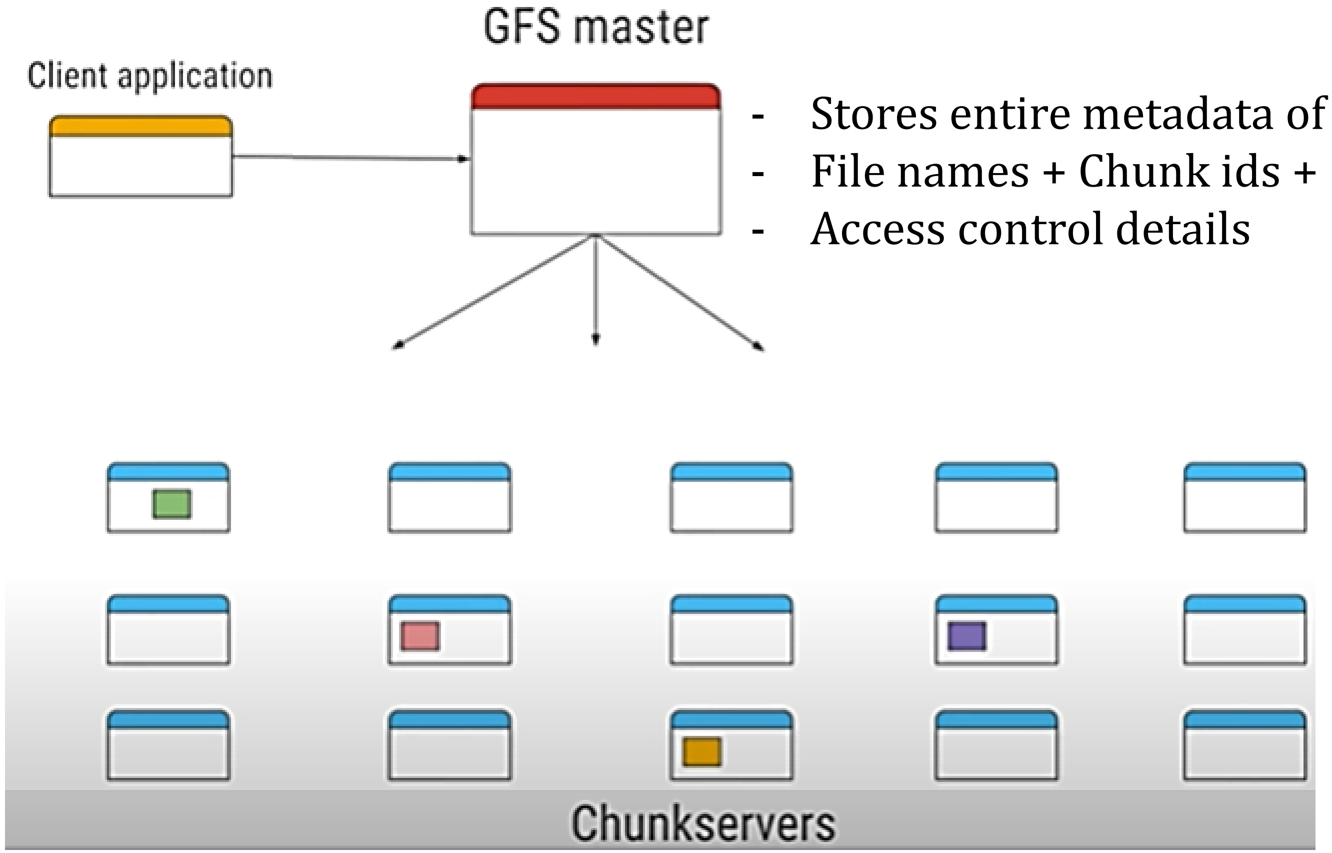
Replicas

Files split into chunks

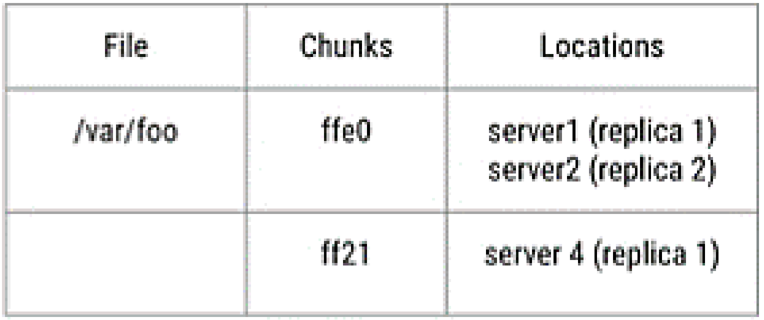
- Replica count by client
- commodity server failures

Replicas ensure durability of data if chunkserver goes down

Google File System (GFS) – Design Consideration

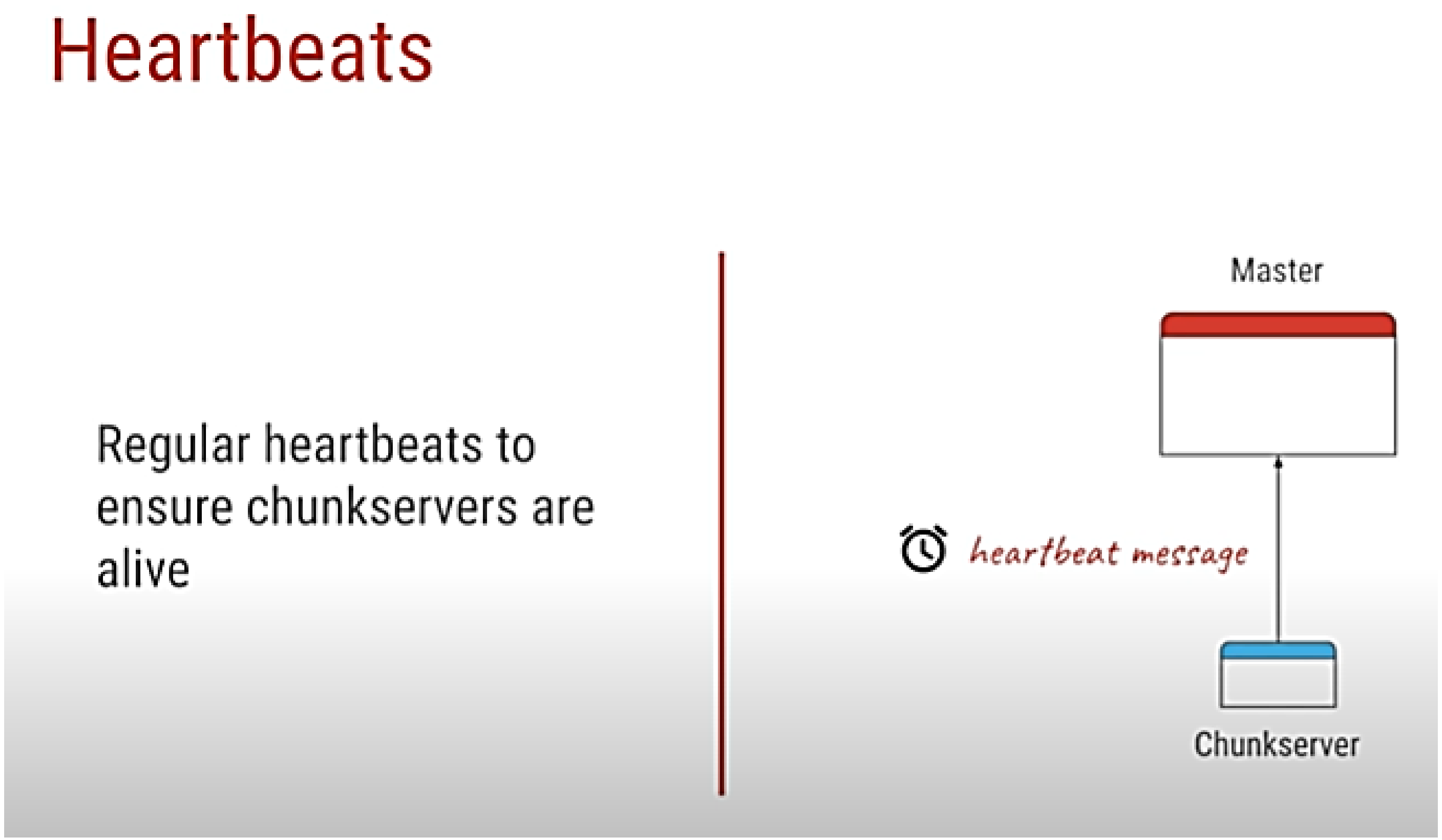


- Stores entire metadata of the cluster
- File names + Chunk ids + Chunk Locations
- Access control details



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

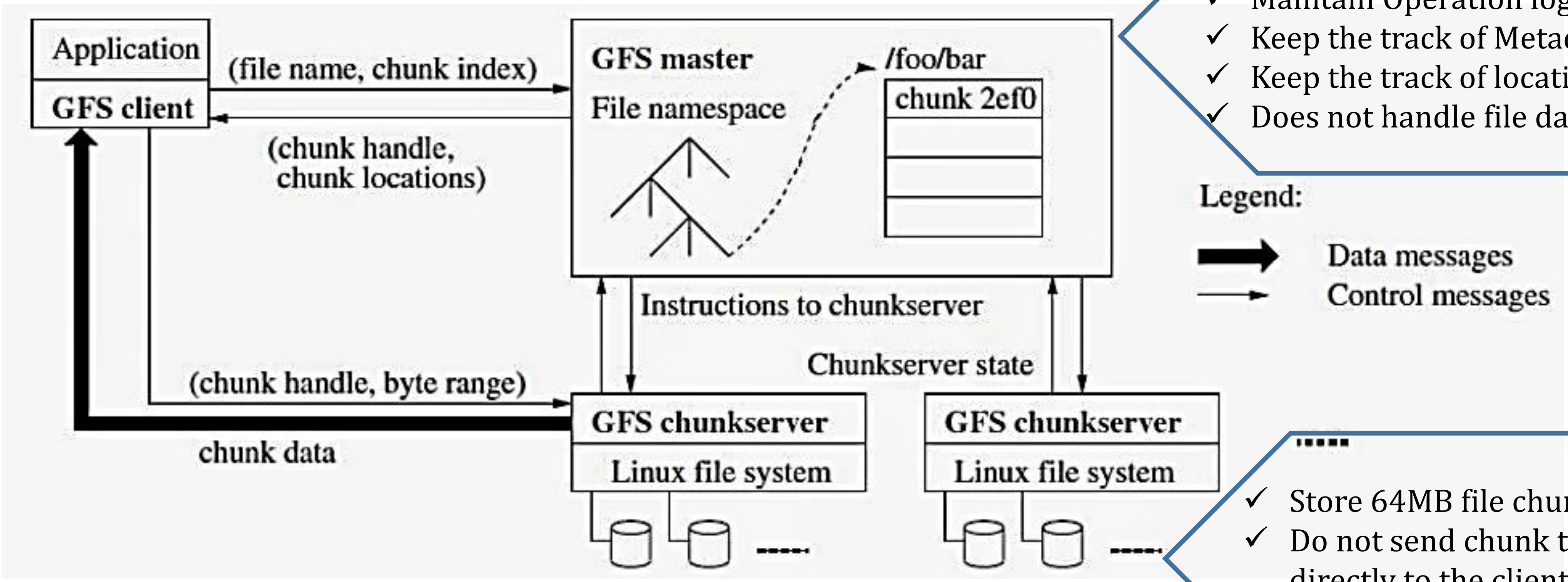
Google File System (GFS) – Design Consideration



Heartbeats

Regular heartbeats to ensure chunkservers are alive

GFS Architecture



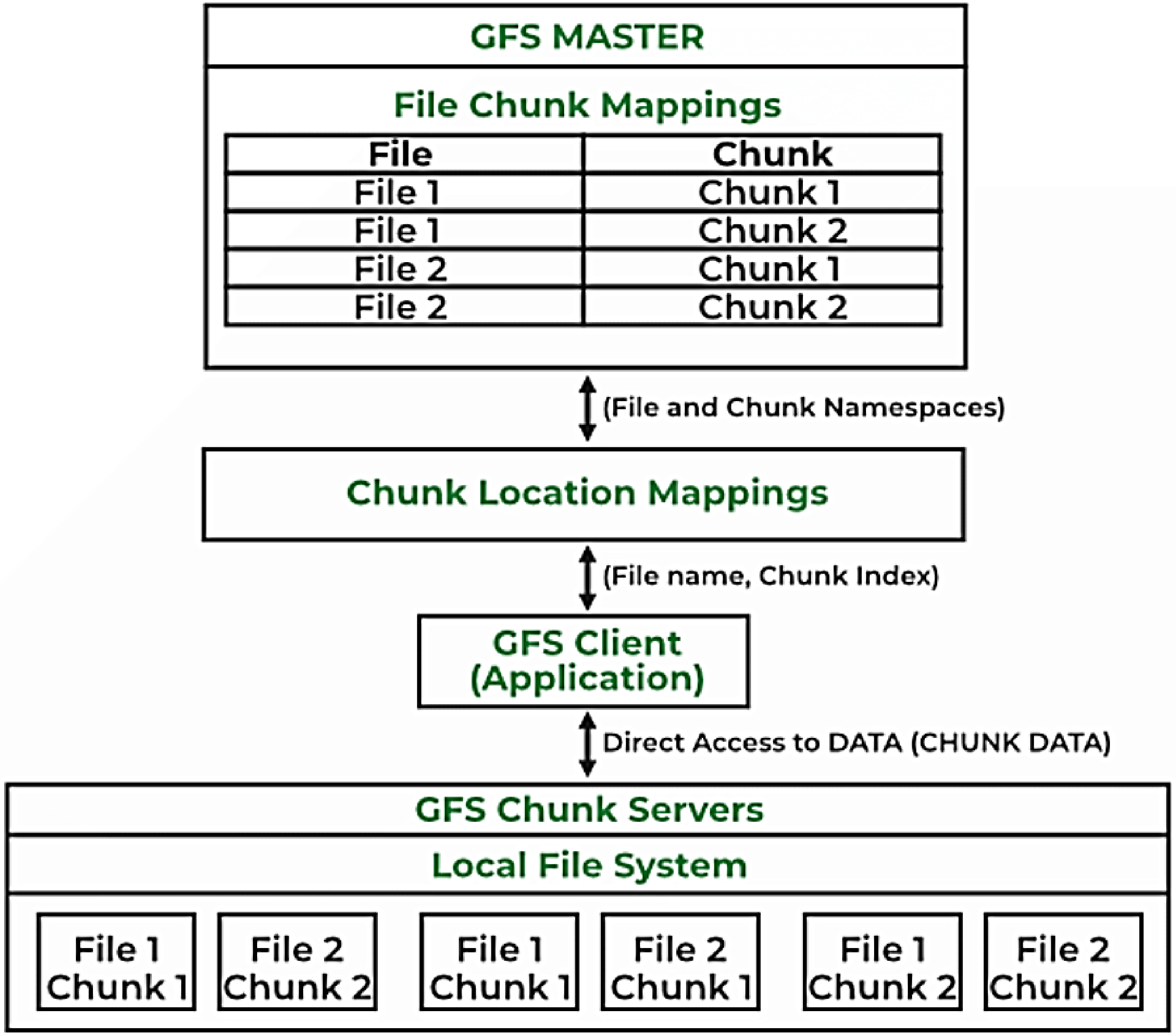
- ✓ Act as Coordinator
- ✓ Maintain Operation log
- ✓ Keep the track of Metadata
- ✓ Keep the track of location of chunks
- ✓ Does not handle file data

- ✓ Store 64MB file chunks
- ✓ Do not send chunk to master-server but directly to the client
- ✓ Chunk Replica (Primary and Secondary)

Reads

In GFS, There are three main entities: *Client*, *Master Server* and *Chunk Server*

GFS Architecture



GFS MASTER

File Chunk Mappings

File	Chunk
File 1	Chunk 1
File 1	Chunk 2
File 2	Chunk 1
File 2	Chunk 2

(File and Chunk Namespaces)

Chunk Location Mappings

(File name, Chunk Index)

GFS Client (Application)

Direct Access to DATA (CHUNK DATA)

GFS Chunk Servers

Local File System

File 1 Chunk 1

File 2 Chunk 2

File 1 Chunk 1

File 2 Chunk 1

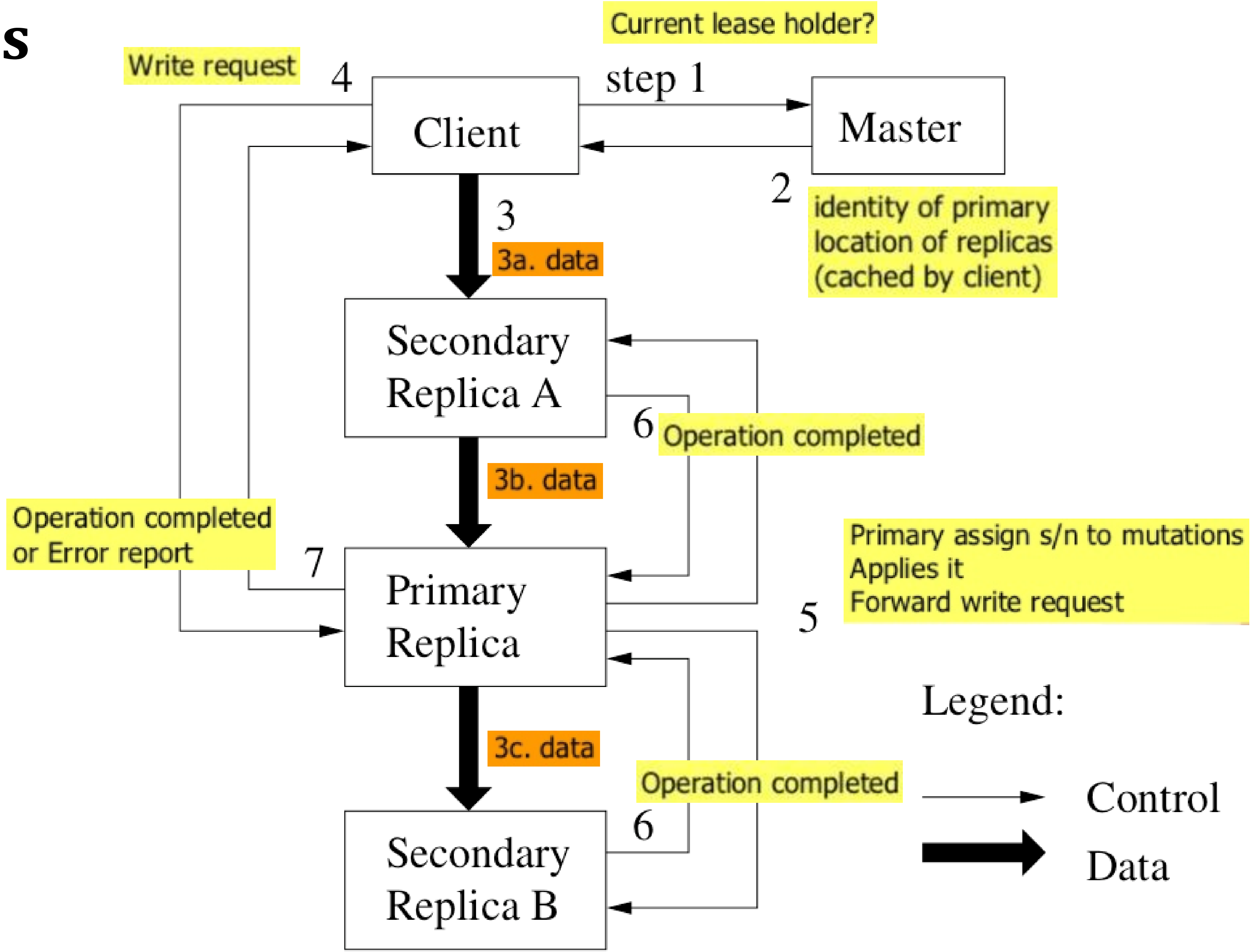
File 1 Chunk 2

File 2 Chunk 2

System Interactions

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



System Interactions

1. Client contact to master for all Chunk-servers.
2. Master grants new lease on chunk, increase chunk version no's including replica’s.
3. Client pushes data to all servers.
4. Once data is asked, client sends write request to primary. Primary decides serialization order for all incoming modifications and applies them to chunk.
5. After finishing modification, primary forwards write request and serialize order to secondaries.
6. All secondaries reply back to the primary once they finish the modifications.
7. Primary replay back to client, either with success or error
   - If write success at primary but fails at any of secondaries then we have in inconsistent state. So error returned to client.
   - If error, client can retry step-3 to step-7

Major limitations of the existing Google File System

- **Single master bottleneck:** GFS relied on a single master node for metadata management, creating a scalability bottleneck as data volume and access requests grew. Imagine a single librarian managing a massive library.
- **Limited metadata scalability:** The centralized metadata storage on the master node couldn’t scale efficiently.
- **High latency for real-time applications:** GFS was optimized for batch processing, leading to higher latency for real-time applications like Search and Gmail. Think of searching for a specific book in a large library with one librarian — it can take time.
- **Static data distribution:** GFS had a predefined data chunk size (64 MB) and replication strategy, lacking flexibility for diverse workloads and storage options like flash memory. Imagine using only one type of box for all your belongings — some things might not fit well.

Colossus File System(extended version of GFS)

- Solution of “Major limitations of the existing GFS” ---> Colossus File System

- The need for Colossus arose from Google’s rapid growth and data demands.

- GFS were becoming inadequate in handling the ever-increasing volume of data generated by Google’s core services.

- Colossus offered a solution with its scalability, reliability, and efficiency. Its unique features allowed Google to scale data storage and access seamlessly, ensuring smooth operation and fast performance for its services.