GFS : scalable distributed FS

- Created for Big Storage Optional: read design overview from GFS paper
- Main motive : deliver huge aggregate performance to a large no. of clients while being fault tolerant.
  - Performance comes from splitting data across multiple servers & being able to read in // (sharding)
  - Automatic Fault tolerance comes from replication of data (with replication of data comes a lot of complexity like consistency of data across servers. To get consistency, you need to pay with decrease in performance)

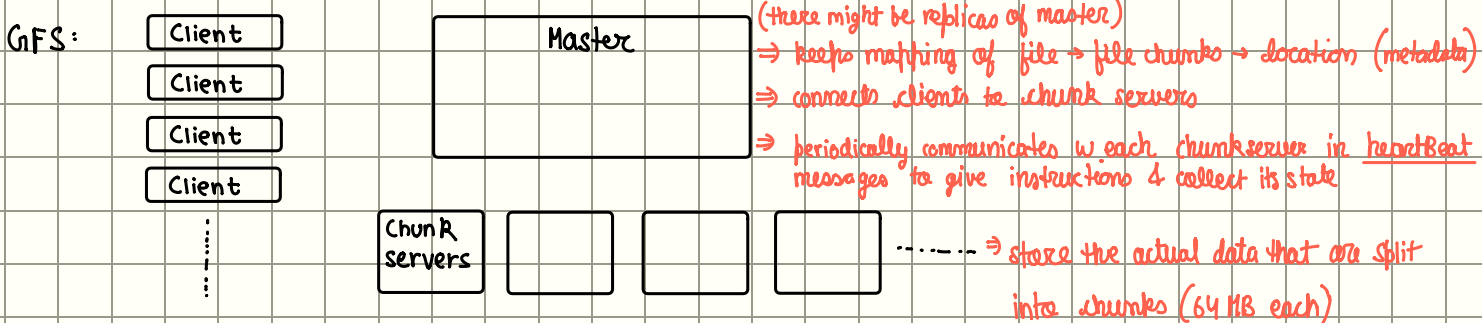
    File → split into chunks

    chunks → have replicas

    there is one primary replica & rest secondary replicas
    ↓ chosen by master

- GFS initially was designed to run at a single datacenter & for internal use.
- Designed for big sequential access. Focus was also throughput rather than latency
- GFS also did not primarily focus on high data consistency

> Consistency in GFS was **relaxed** and more **application-driven**:
>
> - GFS guaranteed **atomicity of record appends** (an append would either succeed once, or be retried, possibly creating duplicate records). This was important for MapReduce workloads where duplicate records are tolerable.
> - **Strong consistency** was guaranteed only under certain operations (e.g., after a successful file namespace mutation like create, delete, rename, you could trust the result).
> - But **concurrent writes** to the same file region by multiple clients could leave data undefined (though metadata remained consistent).
>
> So the tradeoff was:
>
> - Metadata consistency = **strong** (via master)
> - File data consistency = **relaxed** (applications had to tolerate duplicates/undefined regions)
>
> 👉 In short: **High consistency was not the main focus. Fault tolerance and throughput were.** GFS deliberately exposed a weaker consistency model to simplify the system and match its primary workload (large-scale data processing like MapReduce).



What exactly does master store?

⇒ Two main tables
① Map filename → array of chunk ids
② Map chunk id →
a) list of chunk servers that hold replica of that chunk
b) version number of chunk
c) primary chunk location
d) lease expiry time (a chunk is set primary only for a certain leasetime)

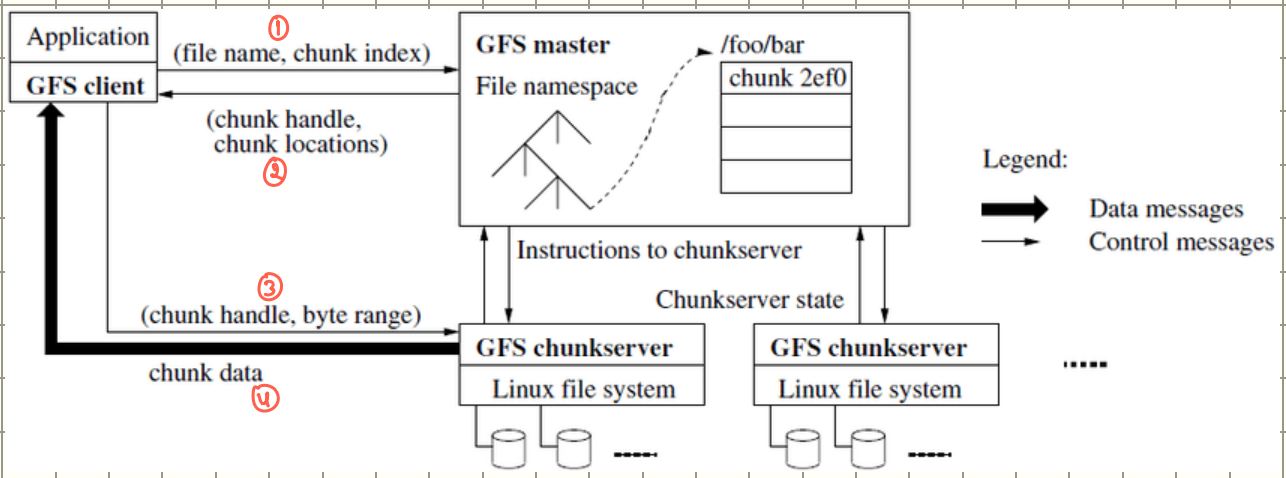
watch lecture3 : GFS 1:01:00 to know reason for lease

#Both these tables are stored both in disk as well as RAM inside master.
→ every time smth is changed in RAM, log is written to disk & disk table is updated
↓
change in ram will happen if any change in above two tables

What happens when we read data from GFS

once written, files are only read, & often only sequentially

1) Client : sends filename, offset where it wants to read from to master
2) Master : looks up filename in table, gets array of chunk ids, does offset / 64 to find which chunk to send.
: looks up that chunk id in chunk table & finds list of chunk servers that have replicas of that chunk
: Returns the list to client
3) Client : Caches the list for later use, takes lil load off of the master
: Has choice to choose (usually chooses the one closest to client in network)
: Sends read RPC req to chosen chunk server, gives it the offset
4) Chunk Server : Has chunks stored as linux files, searches for file & the client then reads the desired range of bytes from the file located in chunk server.



What if client wants to read a range of bytes that spans across chunks?

⇒ When client requests metadata for a range that spans multiple chunks, master returns information of immediately following chunks to sidestep future lookups & reduces cost. The client issues sequential reads to appropriate replicas for each subsequent chunk, if a needed mapping is missing or expired, it contacts master again.

What happens when we write data to GFS (lets just consider file appends)

most files are mutated by appending rather than overwriting. Random writes within files are practically non-existant

In reading, client can read from any up-to-date replica. In case of writing, for consistency purposes, writing happens to primary chunk & is cascaded to replicas later.

1) Client: I want to append to "filename", gimme last chunk
2) Master: If primary isn't assigned for that chunk
   : master finds set of chunk servers with most up-to-date copy of chunks.
   : up to date means version # of chunk on chunk server = ver # in master metadata table
   : master picks a primary
   : increments version #
   : tells primary & secondary servers the new version # & gives lease time to primary
   : tells client the primary & secondary servers

3) Client: Sends copy of data to be appended to all the chunk servers (in a clever way for clever reasons) →
   : The data is stored to a temp location, not yet appended to chunks.
   : After all chunk servers have data, client tells primary to append.

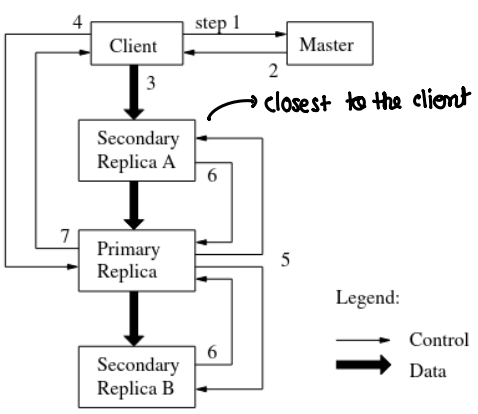


Figure 2: Write Control and Data Flow

4) Primary: Primary receives these sorts of requests from lots of diff clients concurrently
   : It picks some order to serve the requests
   : Finds EOF & picks offset = EOF & Appends the data to offset
   : Tells secondaries to append the data to offset = EOF

5) Secondary: May or may not append (in case of failures)
   : If appended, says yes to primary

6) Primary: If receive yes from all secondary, reply SUCCESS to client