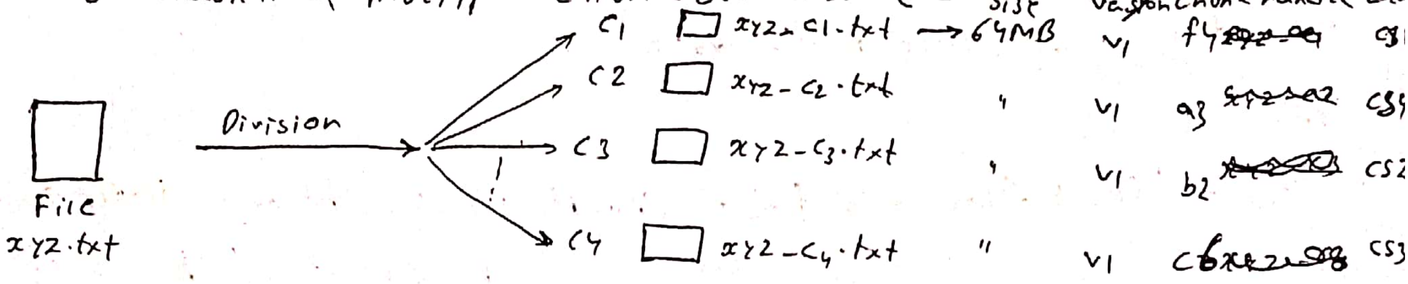
Google File System.

→ This was a system which was optimized for concurrent sequential writes & reads. Okay Performance for random r/w.
called record appends.

(CS → chunk server)
64 bit immutable & Unique

* Architecture:

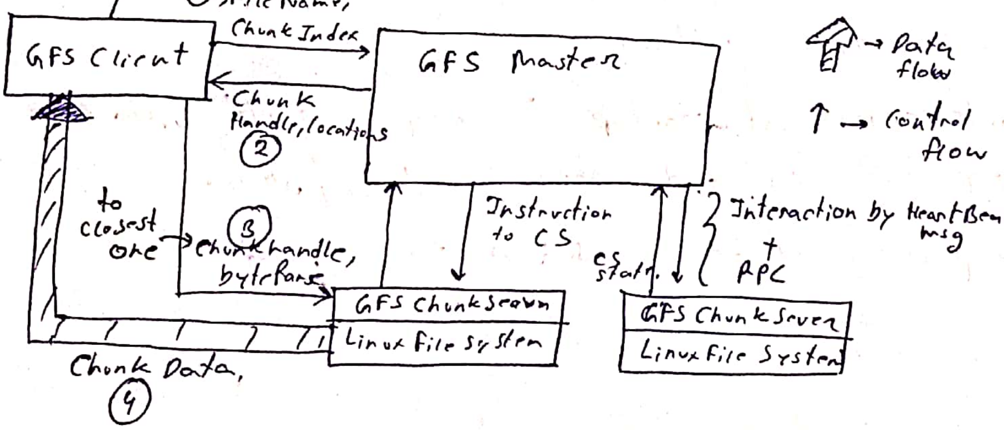
→ Single master & multiple chunk servers. (C → Chunk)



→ Each chunk is replicated 3 times (configurable) for reliability.

→ Each chunk has → i) Size = 64 MB
ii) Version #o
iii) Chunk Handle → 64bit unique & immutable id
iv) a chunk location.
v) Divided in 64KB blocks & each block has a CRC

* Reads: Can cache the chunk locations & asks CS directly



~~→ Neither~~
→ Client does not cache file data cuz:
i) Too large working set
ii) Stream through files

→ CS does not cache file data:
i) Chunks are stored in linux ~~fs~~ which has its buffer cache.

∴ No caching ⇒ No cache coherence issue.

* GFS Master:

→ The master has multiple non-volatile & volatile data:

- File Name → Chunk handle mapping (Non volatile)
- Chunk handle & its version # (Non volatile)
- Primary (Volatile)
- Lease time (volatile)

All are stored in RAM for fast Response.

- Chunk location (volatile) → Informed to ~~Chunk~~ Master by chunk server at startup.

But File Name & Chunk handle to CS mapping + version # → In persistent storage. [log + Checkpoint]
Replicated to remote machines.

→ Functions of master: Main → Failure Recovery

i) Name space & Locking Management
ii) Replica Placement
iii) Creation, Re-replication, Rebalancing.
iv) Garbage Collection.
v) ~~Stale~~ Stale Replica Detection.

→ Why is chunk size 64 MB?

Pros:

i) Reduces client's need to interact with master

Large file ⇒ I have to read & write from same file mare chance ⇒ Less Req to master.

ii) Can keep a persistent TCP connection & reuse it instead of multiple TCP conn. for smaller chunk sizes.

iii) Master keeps less meta data.

Cons:

i) Single / Small files ⇒ Less Chunks ⇒ Can become Hotspots.
⇓
Fixed by storing them with high replication factor.

→ Problem with single master:

i) Limited by memory. 64MB ~~size~~ chunk needs 64B namespace.
↳ Can be fixed by more storage.

ii) No auto recovery ⇒ Solved in Raft.

→ How does master know chunk location?

→ It polls the chunk servers at startup. It keeps itself upto date by Heart Beat msg with CS and when CS leave / join.

→ This eliminates the problem of having CS & master synced

→ How does master recover from failure?

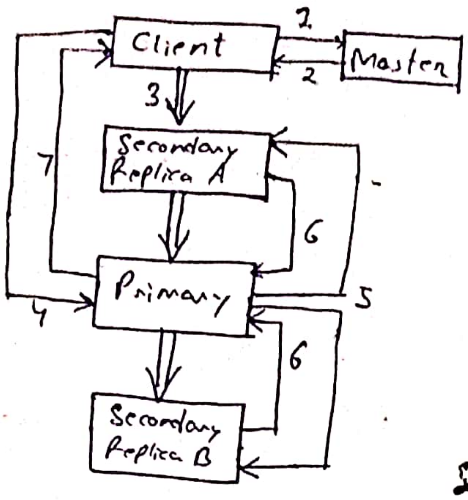
→ It has a operation log & does checkpoints.

Operation log: Since Every item is uniquely identified by some ids. whatever operation the master is asked to do, it first logs the operation and then starts doing it. This is so cuz if it fails in between it knows what operation to do. When logs grow beyond a certain size → It checkpoints the state.

→ whenever a master restarts, it starts with state from last checkpoint and performs the log operations subsequently. The logs & checkpoint data are also ~~stored~~ replicated & stored in other remote machines.

* Writes → Means record appends.

→ The master leases a particular replica of a chunk for 60s to be Primary. If for whatever reason, ~~mast~~ ~~Primary~~ master needs to wait for lease expiry before making someone else the primary



1) C asks M for file's last chunk.

2) If M sees chunk has no primary (or lease expired):
   a) If no chunk servers with latest version # → Error.
   b) Pick Primary & Secondary from those with latest version #
   c) increment version # → write to disk (log on)
   d) tell P and secondaries who they are and new version #
   e) replicas write new version # to disk.

2) ~~Client sends~~ M tells C the P&S

3) Client sends it data to all replicas & P [data flows from client to its closest then to its next closest ex: Client → Sec Rep A → P → Sec Rep B This is done to avoid network bottlenecks.]

4) C ~~sends data~~ tells P to append.

5 → P checks that lease has not expired and chunk has space, picks a offset and writes to that offset. (else write is done on new chunk same by all secondaries.)

5) P tells all S to write at the same offset

6) P waits for all secondaries to reply ok if even one response is error or something. The whole 4-6 is retried

7) P tells C ok or error.

8) If error, it is retried multiple times.

* GFS does not guarantee that all ~~records~~ replicas are identical byte to byte but it guarantees that each ~~record~~ record is written atleast once [due to retry]

* All replicas write stuff at same offset.

* In successful record appends ⇒ The regions are defined ⇒ consistent.

* In ~~unsuccessful~~ Interleaving " " ⇒ " " " undefined but consistent

* Guarantees By GFS: Mutation = ~~Write~~ Record Append. ⑨

→ A file region is consistent ⇒ all replica sees same data irrespective of which replica it is read from.

→ A file region is defined ⇒ If after a mutation, clients can see what ~~it~~ was written i.e

* Interleav regions.

AAA
BBB → AABB instead of AAABBB

AAAA → AAAA

→ Concurrent successful mutations may leave interleaving ~~a~~ fragments. The region is undefined but consistent.

⇒ Failed mutation is inconsistent ⇒ undefined region.

⇒ GFS may insert padding on record duplicates in inconsistent region.

→ After seq of successful mutation, region is guaranteed to be defined

i) applying mutation to all chunk replicas in same order.

ii) Using chunk version # to ensure no stale chunk replica was used

→ So by atomic record appends → ~~In case~~ Undefined & Inconsistent case is removed.

Then How does application check for duplicate / padding regions/records?

① Applications use checksums and/or a unique Id for each record

Valid checksum ⇒ Record valid, Unique Id ⇒ No record is read twice

* Snapshot :

→ ~~Sna~~ Snapshot operation makes a copy of a file or a directory tree structure almost instantaneously, ~~which~~

→ Used to create copies or checkpoints.

How is it implemented: Main idea is immediate copy of directory and lazy copy of chunk.

i) Master receives snapshot request.

ii) Revokes any outstanding request on chunk it is about to snapshot

↳ This means if anyone wants access to the file req goes through master which allows it to be copied.

(iii) After leases are revoked / expired, the master logs the operation to disk.

(iv) Applies the log record to its in memory ~~store~~ state by duplicating the metadata for source ~~and~~ file or directory tree.

(v) The newly created snapshot files point to same chunk.

→ Now Client wants to write to chunk C after snapshot.
i) Client asks master for primary & S.
two tree struct point to same chunk.
ii) Master sees that C has ref ~~cont~~ count more than 1. It defers replying to client req & instead picks a new chunk handle C'.
iii) Asks chunk servers to that has a current replica of C to create a new chunk called C'. By doing this we assure that data is copied on same chunkserver as the original so no network overhead.

iv) Client replies info about C' and all writes are now done to C'.

* Master operations:

i) Namespace Management and Locking
→ Multiple operation are allowed to be active
→ Locks are used for serialization.

→ GFS logically represents its ~~lookup~~ namespace as lookup table mapping full path names to meta data. [Path compression done to save space]
→ There is no i-node thing like Linux.
only on last item
→ Each directory / file as a read / write lock.

Ex:
A directory called /home/user can have a file /home/user/foo. To create the file, we put read locks on /home → /home/user and write lock on /home/user/foo. ① The read lock on user shows that something in its children has write so it can't be deleted.
② There can be multiple writes in /home/user → /whatever
R
W.

ii) Replica Placement.

Two purpose : ① Maximise data reliability & availability

② Maximise Network utilization.

→ Replicated across multiple racks.

→ Trade off: reads can be faster, writes slower
↳ Not a problem.

→ The ~~ma~~ machines are allocated IPs such that closer machine has closer IP.

iii) ~~R.~~ Creation, Re-balancing & Re-replication:

→ When master creates a chunk, it chooses where to place the empty replicas. The factors are

① Place on below avg disk space utilization.

② Minimize # of recent creation on each chunk server.
Recent creation ⇒ ~~follo~~ followed by writes there.

③ Spread across racks.

→ ~~Ref~~ Master re-replicates as soon as # of replicas fall below user specified goal. The factors are:

① ~~be~~ How far is it from replication goal.

② Re-replicate chunks for live files as opposed to recently deleted files.

③ Minimize Impact ~~on~~ of failure on running application. We boost priority of any chunk that is blocking user progress.

→ Rebalancing is done periodically. It examines the current replica distribution and moves replicas for better disk space & load balancing.

iv) Garbage Collection:

→ When a file is deleted, ~~immediately~~

① Master logs the deletion immediately.

② The file is renamed to a hidden name that includes the deletion times.

③ During master's regular scan of file system, it removes any such hidden files if the have existed for 3 days (configurable)

④ Until then, file can be read under new ~~file~~ special name.

⑤ When ~~fil~~ the hidden file is removed from namespace, its in-memory meta data is erased. This effectively severs its ~~relation~~ links to all its chunks.

→ In similar scan of chunk namespace, the master identifies orphaned chunks (unreachable from any file). and erases the meta data for those.

→ In heart beat msg regularly exchanged with master, each chunk server reports a subset of ~~its~~ chunks it has and the master ~~approves~~ replies with the identity of all chunks that are no longer present in master's meta data. The chunk server is free to delete its replicas.

v) Stale Replica Detection:

→ Master keeps track of latest version number & version # is increment during writes [See ~~section~~ writes part].

→ ~~Ch~~ Replicas may miss mutation if it is down.

→ During scanning of chunkserver, it can match the version numbers ~~to~~ of a replica and ~~last~~ latest ~~to~~ and marked as stale which can be re-replicated & deleted [Prev Parts]

* Fault Tolerance:

i) High availability
- Fast recovery (discussed) ⇒ Fast startup incase of master / chunkserver failure.
- Chunk Replication (discussed) ⇒ For stale chunks, creation of new chunks have multiple replicas.
- Master Replication

→ There are replicas of master replicas with full copy of master's state. Paper's design requires human intervention to switch to one of the replicas after a master failure.

ii) Data Integrity: Done by checksums! → can't compare replicas since no guarante of identical byte by byte.

→ Each chunk is broken into 64 kB blocks & each block has a checksum. They are kept in memory & logged persistently.

→ Read Path:

i) verify the checksum of record with the existing one.

ii) On mismatch. ⇒ Returns error & reports corruption to master.

iii) Then the client reads from a different replica. & master ~~clones a~~ does re replication [discussed)

→ write path: Checksum is CPU extensive. So optimized for record appends.

Append Case: when append data server does not verify for the block. It incrementally updates. the checksum for the block. like old checksum + Check sum (Append data) = New Checksum. Cuz If last block was corrupt the new will be corrupt too.

→ Overwrite Case: If you overwrite, Read & verify for first & last blocks that are touched, write the new data. Compute new checksums. why? → If first & last are corrupt then the new data might cover up old corruption [~~old~~ Like corruption can be hidden] [Paper]

→ During idle time, ~~master~~ chunk server can scan and verify the checksums if corrupted reported to master and re replicated. (discussed)