Google File System (GFS) runs on inexpensive hardware, provide fault tolerance and high aggregate performance to a large number of clients. It provides the same services such as performance, scalability, reliability and availability like the previous file systems. While designing GFS the authors explored different points based on their experience. Hardware component failure are the norm rather than exception. With respect to current standard file are huge (like multiGB to multi TB are common). Most files are updated by appending data, random writes are practically rare. Once written, files are read only often sequentially. File system and application which are supposed to use the file system through client are co designed to take the max advantage. The authors have also introduces an atomic append operation to the file so multiple client can append data without extra synchronization.

The design was done based on some assumptions. The file size would be large 100 MB or more (Multi GB or TB). Storing small files also possible but does not need any optimization. Two kinds of read (Large streaming read and small reandom read) were assumed. Write sequentially, multiple concurrent clients append data to the same file, high sustainable bandwidth rather than low latency was the expected behavior at the time of design.

GFS provides familiar file system interface but not POSIX compatible. It supports file operations like create, delete, open, close, read, write. Some advanced operation like snapshot (copy on write) and record append is also supported.

GFS clients communicate with single master and data is provided from different chunkservers to the GFS clients based on locality. Chunk file size is fixed (64 MB). Chunk is identified by chunk handle (64 bits). Chunkservers store files on local disk as linux file. For reliability each chunk is replicated on multiple chunkservers (default 3).

GFS master maintains file system metadata like namespace, access control, chunk mapping and locations. Master send periodically heartbeat messages with chunkservers to send instruction and monitor state.

GFS client library is linked to each application those are using GFS. It communicates with GFS master for metadata operation (control plane) and with chunkserver for read/write data (data plane). It was designed to have single master though shado master incase of hardware failure. This single master assumption provide global knowledge about chunk placement and replication decisions. Single master does not create any bottleneck because, clients only communicate with master to get the chunk placement information after that it fetches/writes data directly from/to chunkservers.

By design chunk size is only 64MB stored as plain linux file on chunk servers, extracted only when needed. This lazy space allocations avoid wasting space. The advantages of having such large chunk size are reduced client interaction with GFS master, reduced size of metadata stored on master, reduced network overhead by keeping persistent TCP connections to the chunkservers for an extended period of time. The disadvantage is the case when small files create hotspot on chunkservers if many clients access the same file.

Master keeps 3 types of metadata file and chunk namespace, mapping from files to chunks and location of chunk replicas. All metadata is kept in master's memory to facilitate period scanning of chunk garbage collection, re-replication and chunk migration to balance load and disk space usage across chunk servers.

The operation log file keeps the order of concurrent operations, metadata namespace, chunk mapping. It is kept on master's local disk and replicated on remote machines. Masters does not persist record of chunk locations rather polls chunk servers at startup. GFS master checkpoint it's state (B-tree) whenever the log grows beyond a certain size.

File namespace mutation are atomic, data is written at a application specified file offset and concurrent append to the record is done to an offset of GFS choosing. A file region is consistent if all the clients always see the same data regardless of replicas. A region is defined after a file mutation if it is consistent and clients see what the change done entirely. In case of failure at any replicas the client try the write again as a result there might be some duplicate data. Therefore the GFS does not guarantee against duplicate but write would be done at least once.

Each mutation (append/write) is carried out at all the chunks replicas. Master grants leases to maintain a consistent mutation order. Master grants a chunk lease for 60s to the primary replica which selects a serial order for all mutations to the chunk. All replicas follow this order (global mutation order is first defined by the master and within a lease by serial numbers assigned by the primary replica).

The write control and data flow is as follows, client ask to master for primary and secondary replicas. Master replies with replicas identity (client cache it with timeout), Client push data to all the chunk servers and the chunkservers store data in a LRU buffer until data is used or timed out. Once all replicas acknowledged receiving data, client send a write request to the primary replica. Primary replica then forward the write request to secondary replicas that applies mutations in the same order as primary. Then the secondary replicas acknowledged the primary that they have completed the operation. Then the primary report back to the client. Any errors done on this period is also report back to the client. In case of error there might be some inconsistency in primary or subset of secondary replicas. GFS client code handles this kind of situation by retrying the failed mutation.

Data flow decoupled from control flow to use the network efficiently. Data is pushed linearly to a chain of chunkservers through TCP pipeline connection. Once a chunkserver receives data it starts forwarding to the next one. Each machine forward data to the closest machine in the network topology that has not received yet.

While appending data, client specify only the data the GFS choose the offset when appending the data to the file. The max size of append record is 16 MB. Client push data to all replicas last chunk of the file to write. Client send it request to the primary replica, primary replica check whether the data size exceeds the maximum size (64MB), if so it pads the remaining part, instruct secondary to do the same and replies the client to try with the next chunk. If the record data fits in current chunk then it append the data and ask secondary to do the same. Any future record would be assigned a higher offset or a different chunk. In google workloads, such file serve as multiple producer/single consumer queues or contain merged result from many different clients.

GFS can take snapshot. It can take copy of file or directory tree by using standard copy-on-write technique. The workflow is as follows, master receives request for a snapshot, then it revokes any outstanding lease on the chunks of the target files, then it wait leases to be revoked or expired, the logs the snapshot operation to disk. Master duplicate the metadata for source file or directory tree. Newly created snapshot files point to the same chunk as source files. Then first time a client wants to write to a chunk C, it sends a request to the master to get the current lease holder, master finds the reference count for Chunk C is greater than one. Then it defers replying to the client and create a new chunk handle c" and ask each the chunkserver containing c to create to clone c" (local disk cloning without using network).

Locks over region of directory namespace ensure serialization and allow multiple operations on master. Each absolute file name or directory has an associated read/write lock. Each master operation acquires lock before it runs. To make operation on a particular file of a directory the master take read lock on all the path directory except the last file where it take write lock. The read lock ensures not to rename/delete any parent directory

name while doing the file mutation operation. Write lock on the file serialize attempts to create a file with the same name. Locks are acquired in a consistent total order to prevent deadlock. First ordered by level in the namespace tree and then lexicographically ordered within the same level.

Replica placement serves 2 purposes. It maximize data reliability and availability. It also maximizes network bandwidth utilization. GFS spreads chunk replicas across racks to ensure chunk survivability, to exploit aggregate read bandwidth of multiple racks. While doing write operation it has to flow through multiple racks.

Chunk replicas created for 3 reasons chunk creation, chunk re-replication and chunk rebalancing. Chunk creation depends on various factors like placing new chunk replicas on chunk servers with below average disk space utilization, limit the number of recent creating in each chunk server, spread replicas of a chunk across racks. Re-replication is done as soon as the number of available replicas falls below a user specified goal. Re-replication of chunk is prioritized based on several factors like higher priority to chunk that has lost 2 replicas than chunk that lost 1 replica, chunk of live files preferred over recently deleted files, boost the priority of any chunk that is blocking client progress. The re-replication placement is similar as for creation.

Master re-balances replicas periodically for better disk space and load balancing. Master gradually fills up a new chunkserver rather than instantly swaps it with new chunks. Re-balancing placement is like "creation". Master also decide which existing replica to remove, better to remove from the chunkservers with bellow average free space to equalize disk space usage.

GFS does not reclaim the physical storage immediately as soon as file is deleted rather it does lazily on regular cycle of garbage collection at both the file and chunks level. The execution cycle is as follows Master logs the deletion, file renamed to hidden name with deletion timestamp. Then while scanning of filesystem namespace master removes any such hidden files exist for more than 3 days. When the hidden file is removed from namespace it's in memory metadata is erased. Master does same kind of scanning on chunk namespace to identify orphan chunks and erase the metadata. In the regular heartbeat message shared by the master the chunkserver share the set of chunks it has, master replies with the identity of the chunk not present in it's metadata. Chunkserver is free to delete all those stale chunks.

If a chunkserver fails and misses mutation chunk replica may become stale. To deal with this kind of situation the chunk replicas keep a version number. Whenever a chunk is granted lease, master increases the version number and inform all the replicas. Chunk version number is persisted on both the master and chunk server. After failure on a chunk server, master detects the stale replica when the chunkserver restart and reports back with it's set of replicas and version number. Master removes the stale replica on it's regular garbage collection. Master also include the version number when it replies to a client request with the chunkserver locaiton that is holding the lease. It also include the version number at the time of instructing a chunkserver to read the chunk from another server in cloning operation.

As hardware failure is a norm in GFS, the system is highly available based on two simple strategy fast recovery and replication. Master and chunkservers are designed to restore their state in seconds no matter how those go down (hardware failure or killing process). GFS client and other servers experience minor hiccup on outstanding request, then reconnect to the restarted server and retry operation. Chunk replicated on multiple servers on different racks. Different parts of a file namespace can have different replication level. When a chunkserver goes offline master clone existing replicas to the other chunkservers and also detect stale replicas by doing checksum verification.

Master operation log and checkpoints are replicated on multiple machine for reliability. If the master machine fails, monitoring process outside GFS starts a new master machine from replication log. Clients only use the canonical name of master machine with DNS alias which can be changed. Shadow master provides read only access to the file system even when the master is down.

Each chunkserver use checksumming to detect corrupted chunk. Chunks broken into 64 KB blocks with associated 32 bits checksum. Checksums are kept in memory and stored persistently with logging. While performing the read operation, chunkserver verifies the checksum of data blocks that overlap the range before returning any data. If a block is corrupted the chunkserver returns error to the client and also inform master about the mismatch. Then the requester read from other chunk server and the master clone chunk from another replica and instruct that chunkserver to delete its corrupted replica. Before doing the write operation the master calculate the checksum of first and last block that overlap the write range, after that it calculate and store the new checksum. Mismatch handling is same as read operation. For append operation checksum handling is optimized. No checksum verification for the last block rather incrementally update the checksum for the last partial checksum block. If a new block is created as for append operation, checksum is computed separately.

During the idle period chunkservers scan and verify the contents of inactive chunks to find out any corrupted replica and adjust chunk replica ratio throughout the servers.

GFS servers generate logs sequentially and asynchronously about significant events and all RPC requests/replies. RPC logs include exact request and response sent on the wire without data. Interaction history can be reconstructed from this log by matching request response and collating RPC records on different machines. Logs can be used for load testing and performance analysis. Most recent events of log are kept in memory and used for online monitoring.