



Fault Tolerance, High Availability, and Data Integrity

Let's learn how GFS implements fault tolerance, high availability, and data integrity.

We'll cover the following



- Fault tolerance
- High availability through Chunk replication
- Data integrity through checksum

Fault tolerance#

To make the system fault-tolerant and available, GFS makes use of two simple strategies:

1. **Fast recovery** in case of component failures.
2. **Replication** for high availability.


Let's first see how GFS recovers from master or replica failure:

- **On master failure:** The Master being a single point of failure, can make the entire system unavailable in a short time. To handle this, all operations applied on master are saved in an **operation log**. This log is checkpointed and replicated on multiple remote machines, so that on recovery, a master may load the checkpoint into memory, replay any subsequent operations from the operation log, and be available again in a short amount of time. GFS relies on an external monitoring

infrastructure to detect the master failure and switch the traffic to the backup master server.



- **Shadow masters** are replicas of master and provide read-only access to the file system even when the primary is down. All shadow masters keep themselves updated by applying the same sequence of updates exactly as the primary master does by reading its operation log. Shadow masters may lag the primary slightly, but they enhance read availability for files that are not being actively changed or applications that do not mind getting slightly stale metadata. Since file contents are read from the ChunkServers, applications do not observe stale file contents.
- **On primary replica failure:** If an active primary replica fails (or there is a network partition), the master detects this failure (as there will be no heartbeat), and waits for the current lease to expire (in case the primary replica is still serving traffic from clients directly), and then assigns the lease to a new node. When the old primary replica recovers, the master will detect it as 'stale' by checking the version number of the chunks. The master node will pick new nodes to replace the stale node and garbage-collect it before it can join the group again.
- **On secondary replica failure:** If there is a replica failure, all client operations will start failing on it. When this happens, the client retries a few times; if all of the retries fail, it reports failure to the master. This can leave the secondary replica inconsistent because it misses some mutations. As described above, stale nodes will be replaced by new nodes picked by the master, and eventually garbage-collected.

 **Note:** Stale replicas might be exposed to clients. It depends on the application programmer to deal with these stale reads. GFS does not guarantee strong consistency on chunk reads.



High availability through Chunk replication#

As discussed earlier, each chunk is replicated on multiple ChunkServers on different racks. Users can specify different replication levels for different parts of the file namespace. The default is three. The master clones the existing replicas to keep each chunk fully replicated as ChunkServers go offline or when the master detects corrupted replicas through checksum verification.

A chunk is lost irreversibly only if all its replicas are lost before GFS can react. Even in this case, the data becomes unavailable, not corrupted, which means applications receive clear errors rather than corrupt data.

Data integrity through checksum#

Checksumming is used by each ChunkServer to detect the corruption of stored data. The chunk is broken down into 64 KB blocks. Each has a corresponding 32-bit checksum. Like other metadata, checksums are kept in memory and stored persistently with logging, separate from user data.

1. **For reads**, the ChunkServer verifies the checksum of data blocks that overlap the read range before returning any data to the requester, whether a client or another ChunkServer. Therefore, ChunkServers will not propagate corruptions to other machines. If a block does not match the recorded checksum, the ChunkServer returns an error to the requestor and reports the mismatch to the master. In response, the requestor will read from other replicas, and the master will clone the chunk from another replica. After a valid new replica is in place

the chunk from another replica. After a valid new replica is in place,



the master instructs the ChunkServer that reported the mismatch to delete its replica.

2. **For writes**, ChunkServer verifies the checksum of first and last data blocks that overlap the write range before performing the write. Then, it computes and records the new checksums. For a corrupted block, the ChunkServer returns an error to the requestor and reports the mismatch to the master.
3. **For appends**, checksum computation is optimized as there is no checksum verification on the last block; instead, just incrementally update the checksum for the last partial block and compute new checksums for any brand-new blocks filed by the append. This way, if the last partial block is already corrupted (and GFS fails to detect it now), the new checksum value will not match the stored data, and the corruption will be detected as usual when the block is next read.

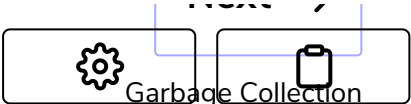
During idle periods, ChunkServers can scan and verify the contents of inactive chunks (prevents an inactive but corrupted chunk replica from fooling the master into thinking that it has enough valid replicas of a chunk).

Checksumming has little effect on read performance for the following reasons:

- Since most of the reads span at least a few blocks, GFS needs to read and checksum only a relatively small amount of extra data for verification. GFS client code further reduces this overhead by trying to align reads at checksum block boundaries.
- Checksum lookups and comparisons on the ChunkServer are done without any I/O.
- Checksum calculation can often be overlapped with I/Os.



GFS Consistency Model and Snapshot...



Mark as Completed



Report an Issue