The Google File System (GFS) was developed in the late 1990s. It uses thousands of storage systems

built from inexpensive commodity components to provide petabytes of storage to a large user community

with diverse needs [136]. It is not surprising that a main concern of the GFS designers was to ensure

the reliability of a system exposed to hardware failures, system software errors, application errors, and

last but not least, human errors.

The system was designed after a careful analysis of the file characteristics and of the access models.

Some of the most important aspects of this analysis reflected in the GFS design are:

• Scalability and reliability are critical features of the system; they must be considered from the

beginning rather than at some stage of the design.

• The vast majority of files range in size from a few GB to hundreds of TB.

• The most common operation is to append to an existing file; random write operations to a file

are extremely infrequent.

• Sequential read operations are the norm.

• The users process the data in bulk and are less concerned with the response time.

• The consistency model should be relaxed to simplify the system implementation, but without placing

an additional burden on the application developers.

Several design decisions were made as a result of this analysis:

1. Segment a file in large chunks.

2. Implement an atomic file append operation allowing multiple applications operating concurrently

to append to the same file.

3. Build the cluster around a high-bandwidth rather than low-latency interconnection network. Separate

the flow of control from the data flow; schedule the high-bandwidth data flow by pipelining the data

transfer over TCP connections to reduce the response time. Exploit network topology by sending

data to the closest node in the network.

4. Eliminate caching at the client site. Caching increases the overhead for maintaining consistency

among cached copies at multiple client sites and it is not likely to improve performance.

5. Ensure consistency by channeling critical file operations through a master, a component of the

cluster that controls the entire system.

6. Minimize the involvement of the master in file access operations to avoid hot-spot contention and

to ensure scalability.

7. Support efficient checkpointing and fast recovery mechanisms.

8. Support an efficient garbage-collection mechanism.

GFS files are collections of fixed-size segments called chunks; at the time of file creation each chunk

is assigned a unique chunk handle. A chunk consists of 64 KB blocks and each block has a 32-bit

checksum. Chunks are stored on Linux files systems and are replicated on multiple sites; a user may

change the number of the replicas from the standard value of three to any desired value. The chunk size

is 64 MB; this choice is motivated by the desire to optimize performance for large files and to reduce

the amount of metadata maintained by the system.

A large chunk size increases the likelihood that multiple operations will be directed to the same

chunk; thus it reduces the number of requests to locate the chunk and, at the same time, it allows the

application to maintain a persistent network connection with the server where the chunk is located.

Space fragmentation occurs infrequently because the chunk for a small file and the last chunk of a large

file are only partially filled.

The architecture of a GFS cluster is illustrated in Figure 8.7. A master controls a large number of

chunk servers; it maintains metadata such as filenames, access control information, the location of all the

replicas for every chunk of each file, and the state of individual chunk servers. Some of the metadata is

stored in persistent storage (e.g., the operation log records the file namespace as well as the file-to-chunk

mapping).

The locations of the chunks are stored only in the control structure of the master’s memory and are

updated at system startup or when a new chunk server joins the cluster. This strategy allows the master

to have up-to-date information about the location of the chunks.

System reliability is a major concern, and the operation log maintains a historical record of metadata

changes, enabling the master to recover in case of a failure. As a result, such changes are atomic and are

not made visible to the clients until they have been recorded on multiple replicas on persistent storage.

To recover from a failure, the master replays the operation log. To minimize the recovery time, the

master periodically checkpoints its state and at recovery time replays only the log records after the last

checkpoint.

Each chunk server is a commodity Linux system; it receives instructions from the master and responds

with status information. To access a file, an application sends to the master the filename and the chunk

index, the offset in the file for the read or write operation; the master responds with the chunk

handle and the location of the chunk. Then the application communicates directly with the chunk server

to carry out the desired file operation.

The consistency model is very effective and scalable. Operations, such as file creation, are atomic and

are handled by the master. To ensure scalability, the master has minimal involvement in file mutations

and operations such as write or append that occur frequently. In such cases the master grants a lease

for a particular chunk to one of the chunk servers, called the primary; then, the primary creates a serial

order for the updates of that chunk.

When data for a write straddles the chunk boundary, two operations are carried out, one for each

chunk. The steps for a write request illustrate a process that buffers data and decouples the control

flow from the data flow for efficiency:

1. The client contacts the master, which assigns a lease to one of the chunk servers for a particular

chunk if no lease for that chunk exists; then the master replies with the ID of the primary as well as

secondary chunk servers holding replicas of the chunk. The client caches this information.

2. The client sends the data to all chunk servers holding replicas of the chunk; each one of the chunk

servers stores the data in an internal LRU buffer and then sends an acknowledgment to the client.

3. The client sends a write request to the primary once it has received the acknowledgments from

all chunk servers holding replicas of the chunk. The primary identifies mutations by consecutive

sequence numbers.

4. The primary sends the write requests to all secondaries.

5. Each secondary applies the mutations in the order of the sequence numbers and then sends an

acknowledgment to the primary.

6. Finally, after receiving the acknowledgments from all secondaries, the primary informs the client.

The system supports an efficient checkpointing procedure based on copy-on-write to construct system

snapshots. A lazy garbage collection strategy is used to reclaim the space after a file deletion. In the

first step the filename is changed to a hidden name and this operation is time stamped. The master

periodically scans the namespace and removes the metadata for the files with a hidden name older than

a few days; this mechanism gives a window of opportunity to a user who deleted files by mistake to

recover the files with little effort.

Periodically, chunk servers exchange with the master the list of chunks stored on each one of them;

the master supplies them with the identity of orphaned chunks whose metadata has been deleted, and

such chunks are then deleted, Even when control messages are lost, a chunk server will carry out the

housecleaning at the next heartbeat exchange with the master. Each chunk server maintains in core the

checksums for the locally stored chunks to guarantee data integrity.

CloudStore is an open-source C++ implementation of GFS that allows client access not only from

C++ but also from Java and Python.