Google File System (GFS) Roll No: 2017CSZ8058

It is a **proprietary** distributed file system developed by Google to provide efficient, reliable access to data for large distributed data-intensive applications. The key assumptions that have guided the design of the file system are as follows:

1. The clusters are built from *commodity hardware*, that have a high chances of failure
2. The workload primarily consists of *large streaming reads* and small random reads.
3. The workloads have many large, *sequential writes* that append data to files.

The file-system design has been highly optimized for the common-case, while supporting the remaining cases, but not necessarily efficient at them.

The GFS cluster contains a **master node**, and multiple **chunkservers**, accessed by multiple **clients**. The master node contains the metadata associated with the file system and chunkservers contain the actual files stored in form of multiple chunks. The centralized nature affords simplicity to the design.

We will look at the design decisions that are guided by the assumptions listed above.

In order to accommodate for failure in chunkservers, redundancy is a fundamental aspect of the design, where each file is replicated among multiple chunkservers (default of 3), each selected based on their proximity to the GFS client writing the data. In order-to prevent bottleneck at the master, the interaction between the client and the master is limited to transacting the metadata, such as the location of the chunkserver containing the file. With the required information, the client communicates with the chunkserver and sends it the data, along with location of replica chunk servers.

Fetching the information of the first chunk of a read operation, typically takes more time due to pipeline of first interacting with the master and then communicating with the chunkserver. But reading multiple times is optimized, due to caching the existing chunk-location information and the master sending the information of chunks immediately after the current chunk, to have an overall high sustained bandwidth. Applications that require modifying an existing chunk, need to get the node that has the chunk-lease from the master, and communicate with it. Although supported, this operation typically takes more time than appending a new chunk to the file.

Hadoop File System (HDFS)

It is an open-source distributed file system, based on GFS, designed by Yahoo! and currently maintained under the Apache umbrella. It is designed to store large datasets very reliably and to stream those datasets at high bandwidth to user applications. It contains a NameNode (which serves akin to a master in GFS) and several DataNodes (similar to Chunk Servers), with multiple clients interacting with the NameNode and DataNodes to avail the desired functionality.

HDFS maintains a journaling system to track the changes made to the file system. For each transaction, the change is recorded in the journal, before being finalized to the HDFS client. The checkpointing system uses this journal to update the current checkpoint and clear the journal. Creating periodic checkpoints is a way to protect the file system metadata. The system can start from the most recent checkpoint and replay the changes in the journal, to get an up-to-date view of the file system.

The NameNode uses *handshake* signal from the DataNodes in order to understand their availability and the blocks that the DataNode currently holds. Although the NameNode persistently stores the file metadata, such as mapping from files to blocks, it does not persistently store the mapping from block to DataNode. Instead it relies on the handshake signal to determine this mapping, at the start of the server. This strategy is adopted because the DataNode has the ultimate say of whether it contains a block or not. It is of no use for the NameNode, to explicitly save this.

HDFS also contains a BackupNode, for immediate recovery in-case of failure of MasterNode. The BackupNode maintains an up-to-date image of the file system in it’s memory, in order to enable fast switching. The way it achieves this is by replaying the changes in the transaction log of the MasterNode to its own image in the memory.

GFS vs HDFS

Some of the major differences between the initial versions of GFS and HDFS are listed below.

Modifying a file

In GFS, modifying a chunk (or) introducing mutations in it, is done using chunk leases. The client communicates with the master and obtains the address of the primary node which has the lease over a particular chunk. Once the mutations are sent from the client to the different chunk servers, the primary determines the order to apply them on the file and sends this order to the secondary nodes, who then apply it on the replicas they hold.

In HDFS, you cannot alter bytes written to a file. You can only append blocks to the file. HDFS makes assumption that the workloads do not need to modify an existing block. GFS allows this, but does not optimise for it.

Concurrent in writing to file

HDFS implements a single-writer, multiple-reader model. A client which wishes to write to a file, first obtains a lease for the file from the NameNode. Once granted, no other client can get the lease, unless the lease-owner closes it, or it expires after a fixed duration of no-renewal requests. However, the lease does not prevent other clients from reading the file. There may be multiple-concurrent readers for a file.

Whereas in GFS, multiple clients are allowed to concurrently append to files. This is particularly important, in the case that results from multiple processes are being merged in the file. Atomicity with minimum overhead are achieved using a special primitive called *record append*. This operation ensures that data is appended atomically at least once. Although this may cause the different replicas ceasing to be byte-wise identical. But it is ensured that particular data is stored at the same offset of the chunk, in all replicas.

Checkpointing system

GFS uses an *operation log* to journal all the changes to the metadata in the master. Whenever the operation log, grows beyond a certain size, the master makes a new

checkpoint and truncates the log. Apart from keeping the log size small, it also helps in fast booting times in case of server-restart, as it has to load the latest checkpoint while applying the limited number of log records after that. In order to avoid the lag of building a checkpoint, the master build a checkpoint in a separate thread, considering only the changes before the start of checkpointing.

Whereas in HDFS, the checkpointing is done using a separate CheckpointNode. It combines the existing checkpoint and journal to create a new checkpoint. It is usually placed on a host, different from that of NameNode.

CheckSum

CheckSum is used for verifying correctness of stored chunks (or) blocks, as it is more efficient than checking equality between blocks stored in different servers (Not even possible in case of GFS, as even valid replicas may be byte-wise different, due to atomic appends). In HDFS every block is stored as two files, one containing the actual data and another containing the checksum of the block and other metadata such as generation time-stamp. While reading the block, the client computes the checksum and verifies it against the stored checksum, reporting corruption to NameNode in case of any discrepancies.

In GFS, each chunk is divided into smaller blocks (of size 64KB) and their 32-bit checksum is computed and stored in the memory. If a read-request comes, the chunkserver verifies that the checksum of the data in the chunk matches that in the memory. In case of error, the chunkserver reports it to the master node. Therefore the onus of managing the checksum lies with the chunkserver in case of GFS, in-contrast to HDFS which hands it over to the client.

Supported Operating Systems

GFS is designed specifically for Linux, while HDFS can use any file system in the DataNode, and hence can operation on a cluster with heterogeneous operating systems.

References

[1] Ghemawat, Sanjay, Howard Gobioff and Shun-Tak Leung. “The Google file system.” *SOSP* (2003).

[2] Shvachko, Konstantin, Hairong Kuang, Sanjay Radia and Robert Chansler. “The Hadoop Distributed File System.” *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (2010): 1-10.