The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Paper Summary by Group 5:
Alok Thatikunta (111498776), Rahul Sihag (111462160), Sweta Kumari (111497926)
{athatikunta, rsihag, swkumari} @cs.stonybrook.edu

The paper presents the design of Google File System, an easily scalable, fault-tolerant and highly available distributed file system specialized for Google's workload. The Google File System contains thousands of inexpensive components and thus failures are common. The files stored are huge (100MB - a few GB) and most of the reads are large streaming reads or small random reads. And most of the writes are appends rather than overwrites.

GFS architecture is a three-layered distributed system. The cluster contains one master and multiple chunk servers. The master maintains all file system metadata (namespace, access-control information, file-chunks mapping and chunk locations) and coordinates between clients and chunk servers. The chunk servers store chunks of data on their local disks. Each chunk is replicated on multiple chunk servers for reliability and allows high throughput when serving multiple clients. The master server uses Heartbeat message to monitor the chunk servers periodically. GFS uses record appends to ensure atomicity as multiple clients are allowed to append data to the same file concurrently and snapshots to create variants and checkpoints of large data sets.

Clients communicate with the master to get metadata and all the data transfer is between the chunk servers and clients. Clients do not cache chunks as it is useless for streaming reads and random reads. This is a major difference from traditional distributed file systems such as NFS and SMB. But the client does cache metadata and Linux handles caching of chunks in chunk servers RAM.

GFS separates the data flow and control flow to maximize the throughput. The garbage collection is done lazily to minimize the cost as it can be deferred to times of low load. Shadow copies of the master server provide greater availability for read requests. Chunk size is default set to 64 MB as it helps in fewer interactions with the master, reduces the network load and reduces the amount of metadata the master needs to store. To improve system throughput and fault-tolerance it employs mechanisms such as snapshot, namespace locking service etc. The consistency model is relaxed and needs cooperation from the applications. For example, an application is responsible for preventing or detecting undefined areas, typically by including a checksum in each appended record. To ensure that all replicas of a chunk update it in the same order (e.g. for concurrent appends), the master gives a lease on the chunk to one replica known

as primary replica. The primary then assigns sequence numbers to updates to the chunk. This minimizes the load on master as it lets replica serialize the updates to a chunk.

The paper provides a detailed discussion of the architecture of GFS and how it is designed to provide efficient reads and writes for Google's workloads. Also, the assumption of single components failing anytime is practical and valuable and building a file cluster keeping these assumptions is the biggest contribution.

But the paper offers little to compare the performance of GFS with other files systems (NFS, AFS etc.) when used with a similar workload. The paper does not justify the selection of 64MB chunk size and there could be a potential issue with large chunks such as hotspots (overly popular chunks), wasted bandwidth and internal fragmentation. The papers also fail to discuss the inefficiency of random seeks besides mentioning that small reads transfer a significant portion of read data. In addition, the system uses a single master that can have many limitations like lack of scalability to a large amount of metadata, lack of scalability to high request rate and failover time. These problems are addressed in a more fundamental way by the successor of GFS, Colossus.