Eric Johnson
efj2106

It is hard to believe that the Google File System (GFS) which has been so instrumental in shaping how large scale commodity hardware file management systems are implemented today originated from a research project nearly 15 years old. When reading this paper the dating of the commodity hardware brings gravity to its age – the original implementation was dual 1.4 GHZ PIIIs!! We are talking pre-P4 hyper-threading (yet this design is still widely implemented today with 6 to 12 core Xeon server CPUs. The design implemented by Ghemawat, Gobioff, and Leung had a large impact on not only Google's scalability but on the future design of Hadoop and many other open source scalable applications that on their own now represent billion dollar industries. For this reason, I will go through some of what I believe were the strongest technical achievements in the paper and why the GFS system design interests me so much.

GFS differentiated itself by providing a design that had reliable performance, scalability, reliability, and availability. By designing a system that openly embraced commodity hardware components (which delivered lower cost but were more susceptible to component failure) the system make component failure a norm rather than an exception. By designing constant monitoring, error detection, fault tolerance, and automatic recover the system was prepare for such component failure and almost expected it. The system was also designed to support large multi-gigabyte files (particularly large for the time) that would be mutated by appending new data rather than overwriting existing data – something they coined the phrase **atomic append** for which is an atomic append operation so that multiple clients can append concurrently to a file without extra synchronization between them. All while doing this high sustained bandwidth would be more important than low latency (processing data in bulk at a high rate rather than higher response time for an individual read/write).

In order to implement such a design they created the concept of a single **master** and multiple **chunkservers** that could be accessed by multiple clients. (This is very similar to the HDFS implementation with a single **master** and multiple **slave** nodes). Files would be divided into fixed-size **chunks** which each would have a 64-bit chunk handle that was assigned by the master at the time of its creation. For reliability each chunk was replicated (default 3 replicas = same used in Hadoop HDFS file system). The master would maintain all system metadata, system-wide activities like chunk lease and migration between servers – communicating through a heart-beat to its chunkservers. Implementing a large chunk size meant it would reduce the clients need to interact with the master because reads/writes on the same chunk require only on initial request to the master for the chunk location.

Three basic meta-data drive the system: the file and chunk namespaces, the mapping from the files to the chunks, and the locations of each chunk's replicas. Aside from the unique use of triplicate duplicates distributed across servers (allowing both increased readability and more reliable redundancy) the system also implemented a more efficient garbage collection system. After a file is deleted the GFS does not immediately reclaim the available physical storage - instead it writes the file as a hidden file and allows it to be unhidden should the deletion need to be reversed. This implementation allows for easy garbage collection at a specified time in the future – an approach that is simple and reliable in a large-scale system where hardware

component failure is likely. It creates a uniform reliable way to clean up replicas as well, instead of aggressively deleting the data and its replicas all at once when requested.

Aside from the clever deletion method the whole process of chunk re-replication and rebalancing help make the GFS system so unique and efficient. The chunk replicas are created for three reasons: chunk creation, re-replication, and rebalancing. All while doing this the system strives to put replicas on chunkservers with below-average disk utilization and while balancing the load of the overall system. As soon as the chunk replicas falls below the required amount (default) the master will instruct a chunkserver to replicate its replica chunk to another location. The whole system will rebalance itself periodically – moving replicas for better disk space and load balancing. Because the replicas exist on multiple chunkservers on different racks it ensure both high availability to multiple users as well as data integrity and fast recoverability. By making sure the chunkservers independently verify the integrity of its own copy by maintaining a checksum it removes a single point of failure in the master node by making sure all replicas remain consistent independently.

If I were to take away certain aspects of the GFS design that make it so widely successful and a basis for the design of future file systems like HDFS I would have to pin it on the ability to leverage commodity hardware (more prone to failure yet cheaper) while simultaneously delivering a system that would allow multiple users to atomically append large files by multiple users. The redundancy and use of things like chunks and chunkservers was able to deliver this widespread large client load read/writes while also ensuring data integrity and restorability. By creating a system that expected component failure and actively managed and repaired itself (creating additional replicas when needed) the system could scale effectively while avoiding catastrophic losses that could be caused by a loss of a single drive, rack, or even blade. Having a distributed system with storage across multiple racks in different locations also ensured data integrity in the event of a loss.

The system was quite remarkable and I hope to do further research into its influences on HDFS and other modern distributed file system design.