Homework-1: Summary of The Google File System

Name: Kunal Baweja
UNI: kb2896
COMS E6998 Cloud Computing & Big Data (Fall 2016) September 25, 2016

Introduction

This article summarizes the various aspects, design principles and goals of *Google File System*[1] presented by Sanjay Ghemawat, Howard obioff and Shun-Tak Leung, as researchers at Google.

The *Google File System(GFS)*[1] is a scalable distributed file system designed for large, distributed and data intensive applications. It provides efficient performance on large number of clients and storage machines and also offers fault tolerance by constant monitoring, replication of crucial data, and fast and automatic recovery. In addition, *GFS*[1] is capable of running on inexpensive commodity hardware. It mininmizes the metadata storage for file system and separates data flow from control flow to maximize network bandwidth usage. The relaxed consistency model of *GFS*[1] simplifies file system design greatly by excluding factors such as symlinks etc in comparison to traditional file systems and also provides crucial guarantees in terms of data availability to applications using *GFS*[1].

The following sections summarize the *Google File System*[1] and it's key details with respect to file system design, interface and operation.

Design Considerations

The Google File System[1] is oriented towards supporting *big data applications* i.e applications which read, write and process large amounts of data. Such applications are generally driven by concept of distributed computing as crunching data in size of multiple gigabytes/terabytes is a time consuming and expensive effort. These distributed applications typically also require guarantees in terms of data availability, integrity, reduced network latency, file system consistency and efficient metadata management. *GFS*[1] tries to address these concerns by making following design considerations:

1. Component failures have been considered as a norm rather than an exception, because *GFS*[1] is designed to work over storage systems made of inexpensive hardware.

2. A modest number of huge files, of sizes in multiple gigabytes, is considered to be normal use case for storing and processing big chunks of data using *GFS*[1].

3. Most files are modified by appending new data, rather than overwriting existing data and once written, all files are read only in a sequential manner.

4. API designs for file system and applications intended to run over it have been optimised to support high read and write throughputs along with multiple clients concurrently appending to a file. This is supported with *atomicity* of writes guarantee provided by relaxed consistency model for *GFS*[1].

5. High sustained bandwidth is more important than low latency as it's assumed that most applications using *GFS*[1] give priority to processing large amounts of data, rather than quick response times.

File System Interface

Apart from the standard file system operations, *create, delete, open, close, read and write*, *Google File System*[1] also provides *snapshot* and *record append* operations.

- The *snapshot* operation facilitates users by creating a copy of a file or directory tree at low cost, sometimes even receursively if required by user.
- The *record append* operation allows multiple clients to append data concurrently to a file while guaranteeing atomicity of operations.

Architecture

A *GFS*[1] cluster consists of single *master* and multiple *chunkservers* which are accessed concurrently by multiple clients. Files are divided into fixed-size *chunks*. Each *chunk* is identified by an immutable and globally unique 64 bit *chunk handle* assigned by the master at the time of chunk creation. Chunkservers store chunks on local disks as Linux files and read or write chunk data specified by a chunk handle and byte range. For reliability, each chunk is replicated on multiple chunkservers and the metadata of the whole file system including information about chunks is maintained by the master node. Clients interact with the master node for metadata related operations but all data-bearing communication is directly handled by chunkservers, as delegated by the master.

The single master node, specifically handles all filesystem metadata operations and has minimized or no involvment in reading and writing data, thus preventing it from becoming a bottleneck in I/O operations. The client contacts the master node for chunkserver information for read/write operations. The master replies with the corresponding chunk handle and locations of the replicas, client caches this response using file name and chunk index as the key and then directly contacts relevant chunkservers in the future for data operations.

Metadata and Operation Log

Maintaining metadata in memory at master node and writing operation log on disk are very crucial for efficient functioning of *GFS*[1] and providing fault tolerance, respectively. *GFS*[1] stores the *file and chunk namepsaces, mapping from files to chunks and the locations of each chunk's replicas* in the memory of master node as these are to be sent as quick response to clients requesting addresses of chunkservers to read/write data. The *namespaces* and *file-chunk mapping* are also kept in *operation log files* on the master's disk and also replicated on remote machines, this is done to maintain consistency in case of failure of master node. The chunk location information is not stored persistently by the master, instead it queries this at startup time from each of the chunkservers or whenever a chunkserver joins the cluster.

System Interactions

The *Google File System*[1] employs a series of system interaction within it's components i.e. master node, chunkservers and master replicas to form the actual working functionalities of the file system. These interactions are summarized below:

1. **Leases and Mutation Order**: To maintain a consistent mutation order over chunks and their replicas, master node adopts a leasing approach for allocating chunks to file system. First the master grants a chunk lease to one of the replicas, called *primary*. The primary picks a serial order for all mutations (content or metadata changes) to be applied to the chunk and communicates to replica chunkservers. All *replicas* follow this order when applying mutations and thus the file system global mutation order is maintained with consistency.

2. **Replica Placement**: *GFS*[1] has a well defined 'chunk replica placement' policy that serves two purposes: maximizes data reliability and availability and maximizes bandwidth utilization.

3. **Chunk Replica Creation**: Chunk replicas are created for three reasons: *chunk creation, re-replication and rebalancing*.

   - *Chunk creation* refers to the creation of a new chunk when the filesystem gets data to be written in form of request for a chunk location to the master node.

   - *Re-replication* is the situation where master re-replicates a chunk as soon as the number of available replicas falls below the user-specified number. This could happend due to a number of reasons such as, chunkserver failure, corruption of replica data on chunkserver or disk failure over chunkserver(s).

   - *Rebalancing* of replicas is done periodically by master to optimize disk space utilization and load balancing.

4. **Garbage Collection**: This refers to claiming back the free space after a file is deleted. *GFS*[1] claims back the physical storage space in a lazy fashion and not immediately after the data is deleted. This approach maskes the system more simpler and reliable. Upon deletion of a file, master logs it in operation log and instead of actual deletion the file is renamed, along with deletion timestamp. The file is deleted finally, after a grace period of 72 hours and this is when *GFS*[1] actually claims back the physical storage.

5. **Stale replica detection**: The master also periodically scans the chunkservers for any stale replicas which may have occurred due to failure of chukserver in applying mutations to a chunk. Upon finding such stale replicas, master removes these stale replicas via garbage collection and if needed creates more copies of chunk replicas.

Fault Tolerance and Diagnosis

As *GFS*[1] is designed to run over inexpensive commodity hardware comprising a large number of storage machines/disks and to handle large amounts of data, it is critical that the system is incorporated with fault tolerance mechanisms. To achieve this *GFS*[1] has set some goals and strategies to achive those, as discussed below.

1. High Availability - The *GFS*[1] should be available at all times, despite possible component failures. This is achieved by:
   - **Fast Recovery**: *GFS*[1] doesn't distinguishes between normal and abnormal termination of master or chunkserver, hence enabling them to restore their state and start within a few seconds irrespective of the fact, how they might have stopped.
   - **Chunk replication**: To deal with situations like disk failures, network failure or server failure etc, data is replicated over multiple chunkservers, so in case one or more chunkservers become unavailable, there are replicas available to take up the request of data on their behalf.
   - **Master replication**: The master state is replicated for reliability. Its operation log and checkpoints are replicated on multiple machines. If a master fails, 'shadow' masters take over the control, which provide read-only access to the filesystem until master node takes back control. In case of permanent failure, the external monitoring system associated with *GFS*[1] creates a new master node for the cluster.
2. **Data Integrity**: Each chunkserver uses checksumming to detect corruption of stored data. To confirm integrity of data across multiple replicas, the chunkservers obtain checksum(e.g., MD5) which is compared against the checksum values of othe chunkservers containing the replicas of the same chunk of data to detect anamolies. This mechanism prevents against data integrity failures due to component failures or read-write failures.
3. **Diagnostic Tools**: *GFS*[1] servers generate diagnostic logs that record significant events and prove helpful in problem isolation, debugging and performance analysis. The performance effect of this logging is minimal whereas benefits are huge, as controlling and observing a cluster of thousands of nodes is not an easy task without proper logging mechanisms.

References

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, (New York, NY, USA), pp. 29–43, ACM, 2003.