Project Title : Distributed GFS

Project is based on the paper Google File System.

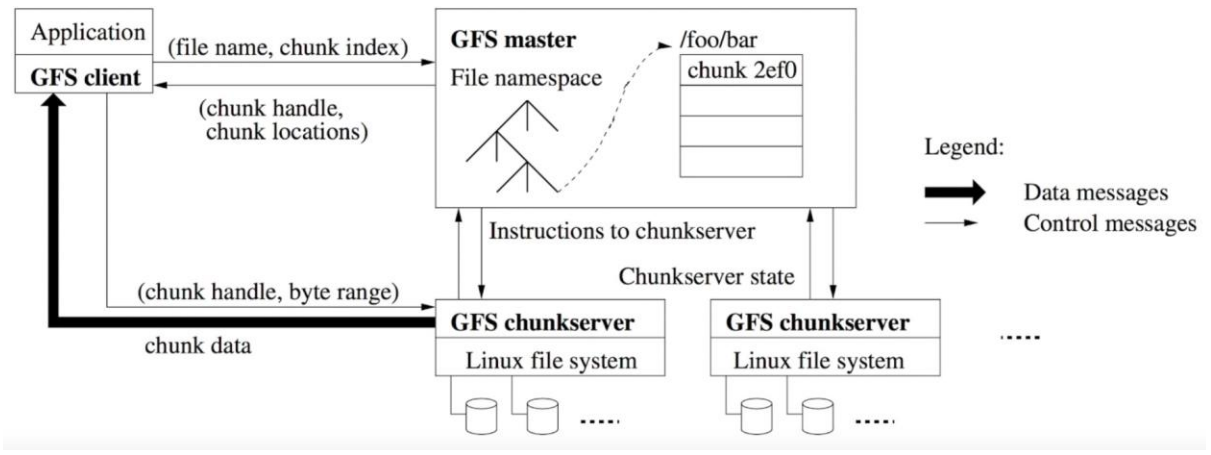
Traditional file systems face limitations in handling large-scale data storage and fail to provide adequate fault tolerance mechanisms. As data volumes continue to grow exponentially, there is a pressing need for distributed file systems capable of efficiently managing massive datasets while ensuring data reliability and availability. The project aims to address these challenges by developing a GFS that utilizes chunking and replication techniques to enhance fault tolerance and scalability.

Google File System from its' Research Paper

INTRODUCTION:

Google File System is a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.
Google has designed and implemented a scalable distributed file system for their large distributed data intensive applications. They named it Google File System, GFS. GoogleFile System is designed by Sanjay Ghemawat, Howard Gobioff and Shun-Tak Leung of Google in 2002-03.GFS provides fault tolerance, while running on inexpensive commodity hardware and also serving large number of clients with high aggregate performance. Even though the GFS shares many similar goals with previous distributed file systems, the design has been driven by Google's unique workload and environment.Google had to rethink the file system to serve their "very large scale" applications, using in expensive commodity hardware.

OUR IMPLEMENTATION:
Having been inspired by the paper I will implement the google file system.
The figure below depicts the overall layout of GFS.



Key Components:

1)Master Server:

It stores the metadata of entire file chunk wise. It is the backbone behind the entire system. Client contacts Master Server while uploading and downloading any file.
It redirects the client to chunk servers appropriately.

HeartBeat:

Running on a different thread at a time interval of some seconds ,the master server will check whether all the chunk-servers are active or not. If a chunk server fails, then when the master server catches it, it will trigger a functionality which will send all the file chunks that the failed chunk server had from their replicas stored in other chunk servers to other appropriate chunk servers which will maintain the condition that there will always be two replicas in every case of every chunk of a file.

***2)*Backup Master Server:**

It comes into play whenever the Master server goes down. Its
basic working is similar to Master server.

3)Client:

It is basically the end user trying to upload and download the document without being worried about the complex operations. A particular client can put a lease on certain file so as to restrict the file from being accessed by other clients.No other client can put a lease on that file for that particular time. Client can later un-lease the file as well. If a client does not un-lease the file then it is un-leased by default after 60 seconds.

4)Chunkserver:

Four chunk servers would be running on four different terminals.
The file is stored in format is chunks in round robin manner in chunk server whenever it is uploaded for first time. Then every time a client uploads a file, it is stored in forms of chunks based on load balancing. The replicas of the chunks is maintained in load balanced manner.
A list is maintained for every chunk servers whenever any chunk server goes down master tells the appropriate chunk servers to redistribute the chunks from that particular chunk server to the another chunk server to maintain the constraint that there will be every time one replica of a particular chunk of the file. It means every chunk of a file will be stored on a primary chunk-server and a secondary chunk server at all circumstances.

Why Big Storage is hard?

1. To improve **performance**, large systems require **sharding**
2. Sharding leads to faults
3. To improve fault tolerance we need make replications
4. Replications leads to inconsistencies
5. To bring consistency, we often require clever design where the clients and servers have to do more work, this leads to low performance

Goal for the project

To understand:

1. Working of Distributed File Systems while building a simple fault tolerant GFS.
2. Working of RPCs.

Design Goals & Assumptions

Writing to a file is more challenging than **reading**, because all chunk replicas must be in sync. This means that they must apply the same updates in the same order. GFS achieves this by designating one replica of a chunk as the primary. It determines update order and notifies the secondaries to follow its lead. Large updates are broken into chunk-wise updates. File creation proceeds similarly, but does not need an update order,

Language Specifications

Due to time constraints, simplicity was favoured over a complex and more apt design, as in the actual GFS. So, I decided to go with Python and RPyC instead of using something like C++ and gRPC which would've been better for learning.

WBS

- Adding Remote Procedure Calls with rpyc
- Adding replication factor
- Adding Create, Read, Append, Delete & List operations