[File System] UNIX VS. GFS

- In this article, I want to discuss two classic papers of File System, which are almost 30 years apart. I feel it is very interesting to understand and review their differences.
    - Dennis M. Ritchie , Ken Thompson, The UNIX time-sharing system, Communications of the ACM, v.17 n.7, p.365-375, July 1974
    - Sanjay Ghemawat , Howard Gobioff , Shun-Tak Leung, The Google file system, Proceedings of the nineteenth ACM symposium on Operating systems principles, October 19-22, 2003, Bolton Landing, NY, USA
- I will compare them through the following items.
    - Different Applied Target (Motivation)
    - Different Manifestation of Design Principles
    - Design Challenges
        1. Consistency
        2. Scalable Performance
        3. Reliability
        4. Availability
        5. Fault Tolerance
        6. Transparency
- The blog version: http://hawxchen.blogspot.com/2016/11/file-system-unix-vs-gfs.html

1. Different Applied Target (Motivation):

- GFS: For large distributed data intensive applications and web services such as YouTube, Google Drive, and Google Docs, they need a scalable and distributed file system to store large or even large amount of data shared between users.
- UNIX FS (File System): For highly interactive applications such as editor, shell, and graphical user desktop, they require a file system to minimize response time between user and kernel.

2. Different manifestation of design principles

	GFS	UNIX FS
Components	• Constructed from many common and inexpensive components that might be often malfunctioned. • Component failures are viewed as the norm instead of the exception.	• It does not assume to use those kinds of often malfunctioned components like GFS. • Component failures are seen as exceptions.
Read	• In **Large streaming reads**, individual read might read hundreds of KBs, usually 1 MB or more. Consecutive operations from the same client usually read through a contiguous region of a file. • **Small random reads**, each read might read a few KBs at some arbitrary offset.	• The size in each read is much fewer than GFS. Generally, It might only read data in tens of bytes. • In UNIX FS, it mainly uses random reads at arbitrary offsets.

Write	• GFS majorly uses large, sequential writes to end of files (Append). • Writes in small size at arbitrary offset in a file are still provided without improving efficiency.	• The size in each write is much fewer than GFS. Generally, It might only write data in tens of bytes. • It mainly uses random writes at arbitrary offsets.
Concurrent Write to a same file.	• In GFS multiple clients might concurrently **append** to the same file. • GFS use two techniques to maintain atomicity with minimum overhead o Producer-consumer queue o Many-way merging. • If the region is **defined** and **consistent**, then no writes will be lost and **each client** can see the same result from mutations.	• UNIX FS creators thought that they have no opportunity to let one independent process own a large file so that they did not design user-visible locks. • Without user-visible locks, if users have the same privilege to a file, users can write the file in any time and at any offset. Unlike GFS, although UNIX users can see the same file (**consistency**), they might lose some writes (**undefined state**) due to overwrites to the same file regions.
Bandwidth VS. Latency	• For GFS, high steady bandwidth is more important than low latency.	• Due to the focus on interactive applications, UNIX FS is devoted to achieving low latency.
File Size	• Typically, each file is 100 MB or larger in size. • Multi-GB files are ordinary.	• The Maximum single file in the UNIX FS is 1MB.
Block Size	• In GFS, it uses the chunk to store data, the unit of a chunk is 64MB. • Although in raw layer file system, GFS still use blocks, the unit of each block size might be higher than 512 bytes.	• Its block size is 512 bytes

3. Design Challenges:

1. Consistency
    - Although UNIX FS keeps consistency in directories and concurrent writes to a file, GFS can be better through the **defined state:** consistency and all updates from writes and append are known to clients.
    - However, GFS is still possible to have an **undefined but consistent** state.
2. Scalable Performance:
    - To achieve performance, GFS separates control and data plane.
        1. Master server is responsible for control plane. If clients want to know which one Chunkserver they can request the replication from, then they can ask the Master server.

    2. Chunkserver is responsible for data plane. After clients asks Master server to get the exact chunkservers, they can request replications, then clients can transfer data between themselves and chunkservers.
  - To achieve scalability, GFS manages different levels of replications such as Master server, primary replica, and secondary replicas throughout chunkservers so that enormous amount clients can request same content from different chunkservers.
  - Because of so many replications in different chunkservers, GFS uses the **pipeline** technique to achieve high performance of updates.
    1. To completely utilize each machine's networking bandwidth, each chunkserver, masterserver, and client transfer the data as soon as possible, rather the sliced data into multiple receivers.
    2. To keep away from high-latency connections and network bottleneck, according to the network location, each machine just copies the data to the nearest machine, which does not receive the same data.
  - UNIX FS has little scalability because, in the 1970s, networking was not popular like now. They did not have enough motivation to achieve scalable performance like GFS in 2003.
3. Reliability
    - Unlike UNIX FS, which has no self-made visible backup for files, GFS has chunk replications in many chunkservers. When one replication failed, clients can request the replications from other chunkservers.
    - In GFS's internal management, it does not only replicate **data chunks** but also maintains **replication for management chunk of Master** server in different chunkservers. Therefore, if the central manager, Master server, crashed, then client can request information from the replications of Master server so that GFS achieves high reliability.
4. Availability
    - According to GFS assumption, server failures are the norm so that creators made GFS achieve high availability through the techniques Fast Recovery, Chunk and Master Replication.
    - For Fast Recovery, masters and chunckserver are regularly shut down to restore their state and restart their services so that they can prepare well for any abnormal termination in the future.
    - For Chunk Replication, because chunk replications are in chunkservers of different racks, clients can acquire the file namespace's different parts from servers, even though some of chunk replication might be malfunctioned.
    - Master Replication is not only helpful to reliability but also availability. GFS has shadow masters falling behind primary master servers with management information. Although shadow servers only provide Read Access, they still can provide some requested information to clients.
5. Fault tolerance
    - GFS achieves fault tolerance through three techniques: Master and Chunk replication, operation log, and data integrity.
    - For Master and Chunk replication, client does not have to worry about any failure from Master or Chunkserver because GFS will restore state and restart service through replications of different servers.

- For Operation Log, GFS will save operations after the last checkpoint. When servers were malfunctioned for any reason, to restart these servers, GFS will read the operation log to load a checkpoint and replay each operation in the log.
- For Data Integrity, GFS uses checksum to verify the integrity of each chunk. GFS also enhances checksum mechanisms through partial updates of the tail in each checksum because appending is the major file operation.

6. Transparency
    - For fault tolerance and load balance, GFS can achieve transparency through the location independent namespace stored in Master server.