COMPSCI 590S Systems for Data Science **Fall 2016**

Lecture 12: File Systems and GFS: the Google File System

Lecturer: Emery Berger *Scribe(s): Udit Saxena, Freddie Sanchez*

12.1 Review of File Systems

We reviewed the general concepts of file systems.

Traditional file systems treat everything like a file and have to do with reading and writing on files on file systems. They have a hierarchical structure and have a directory tree as the file system layout.

Most traditional file systems have a POSIX abstraction layer which functions like an API layer. The functions allowed using this API layer include:

- creat()
- open()
- close()
- read()
- write()

Some common examples include: ext 3/4, NTFS, FAT32, HFS+, ZFS, ReiserFS, XFS.

A "Named Pipe" is where a file points to a pipeline, and writing to the file would push to the pipeline process.

File Systems also allow us to mount filesystems anywhere in the directory tree. eg a remote file system can be mounted on a local system in a separate directory, which appears local. Each access to the directory is treated and transmitted as network messages along the network. Each place where the files are mounted are called mount-points.

12.1.1 Need for File Systems

- Different Use cases: Assign and access small chunks as compared to larger chunks - can help avoid internal fragmentation as compared to utilizing spatial locality while assigning and accessing larger chunks/blocks of memory.
- Different Hardware: Different issues crop up with different design goals for the file system. Fault Tolerance can dictate the nature of file systems.

Some possible issues that can occur with file systems is when a write is in flight, where a file is actively being modified, if something were to go wrong, power goes out, or in the case of USB?s it is ejected, the file can have an inconsistent state. E.g.: ext3 is not fault tolerant, but FAT32 (in USBs) asks user to notify that the drive is going to be ejected. This helps avoid file inconsistencies due to unforeseen and unstable

interruptions to the operations of the file system. The data within a file can be scrambled, and even the Metadata can be affected. Inconsistent states within a file are hard to recover.

Other systems like NTFS, HFS+, ext4 have fault tolerance built in, such as creating "journals" which write down the events of how a file was modified. This journal exists on disk. After a failure, the file system looks at the journal during reboot, and checks for matching "START-END" in the journal/log. If there are unmatched pairs, it decides that the file is in an inconsistent state. Via consistent "check pointing" (saving the current history of modifications done to a file), "rollback" to a previous stage if an inconsistent state is detected is possible. This makes it possible to return to a state where the file was consistent.

A checkpoint only tracks the writes to a file. "COW" stands for copy on write. It create a new copy of a file until it writes, it makes a replica and then makes the modifications.

Restrictions like file name limitations and file encoding also vary across file systems - e.g. : using 32 bits in FAT32 was not really sufficient for file sizes larger than 4Gb, and large directories were not possible. NTFS avoided these limitations by encoding files in 64 bits.

File System Locking also differs across various file systems to allow for different strategies of atomicity of file modification and protection from multiple users. e.g. MSDOS was built with only one user in mind.

12.2 Google File System

GFS is not like a traditional file system, rather it appears to talk to a file system that talks to the disk. There is a level of indirection above the file system.

12.2.1 Structure of GFS

A master-worker paradigm exists here. The Master deals with the filenames and metadata, and there are individual chunk servers that actually store data. The workers are the Chunk Servers. Within the chunk servers there are chunks with a minimum size of 64 MB, each of which are referenced with a GUID. The reason for a big chunk (block) size is that it allows a huge reach, supports big files, and all metadata is kept in RAM allowing faster access. The file system writes to the chunk. All chunks are replicated, and don't need the fault tolerance implemented by other file systems, such as ext4.

The master does not store information of what chunks are in which servers. It is responsible for replication, keeping track of metadata, keeping an operation log, which contains the history of critical data and checkpoints and snapshots. In case the master fails, there is a shadow in the background for recovery.

The master and the chunk server interact using leases and heartbeat signals, and the consistency of files is maintained by using mutation orders in logs across the chunk servers. Data integrity is maintained by using checksums across file chunks.

Usually when a file is being read repeatedly, the file system can cache the file in order to keep it in memory. But GFS does not cache files. In GFS, we make the assumption that we never are going to reread anything, so there is no need to cache it. This is due to the nature of workloads for which GFS is built. Most workloads in GFS are streaming data continuously, so GFS was built that most files will only need to be read once, and modifications or rereads of files will not occur very often.

12.2.2 Record Append

Adding to the end of a file. GFS does atomic adds, where everything being added is guaranteed to be added at least once. It is possible that a record can be duplicated multiple times, but this is a result of relaxing of the syntax to access a system, and by doing this, multiple users can access the files at once. The order in which a record is appended in GFS is preserved as one of the replicas writes first, storing the order in which it wrote. This order is then passed to other replicas, thereby preserving order.