Lecture 21: April 24

Lecturer: Prashant Shenoy *Scribe:*

21.1 Distributed File Systems (DFS)

A DFS is a file system where files are stored on multiple distributed machines.

21.1.1 Coda

Coda is a research FS designed at CMU. It was designed to support Mobile devices. It was **disconnection transparent** which means users would see the all files even if the system was temporarily disconnected. The FS proactively cache files that user is accessing so that in case of disconnection, user can still work on the local copy of the file. It may also happen that a user is disconnected and they try to access a file that is not cached in which case they will know that they are disconnected. The cache adapts so that only files that the user wants are cached. Some source control systems also work this way.

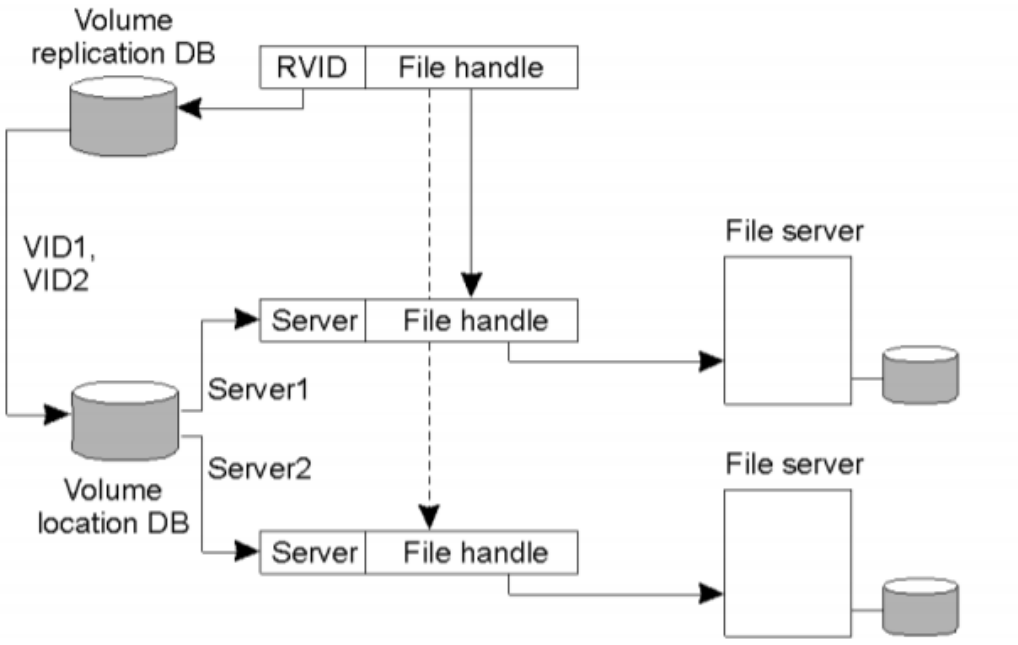


Figure 21.1: Coda architecture

Each directory can be replicated on multiple servers. Each file in Coda belongs to exactly one volume. A volume may be replicated across several servers as shown in Figure 21.1. Multiple logical (replicated)

volumes map to the same physical volume. Any file identifier is of length 96 bits which consists of 32 bit RVID and 64 bit file handle.

21.1.1.1 Issues with server replication

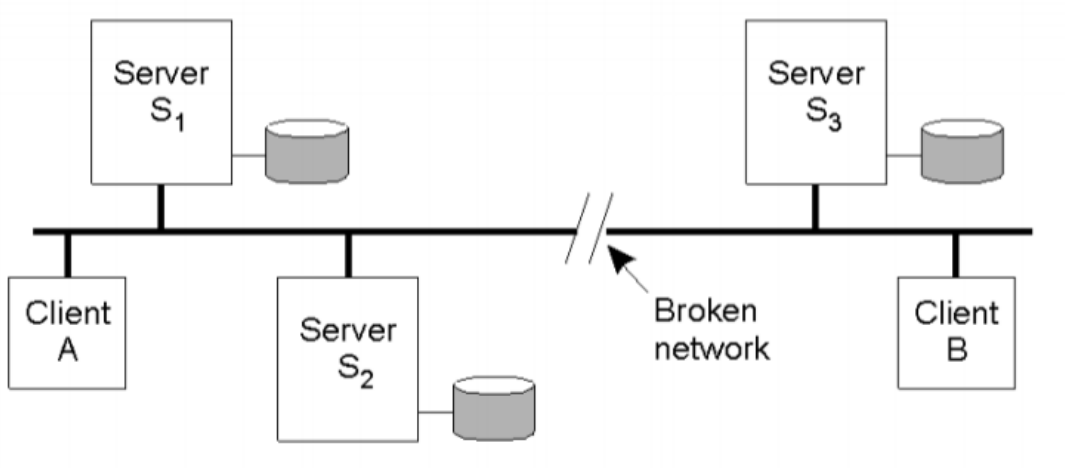


Figure 21.2: Server replication issues in Coda

Figure 21.2 shows what happens when there is a disconnection in the network. Each partition uses the local cached copies and application treat the local cached copies as actual files and all operations are performed on them. Later, when the network reconnects, the two parts resynchronize. Each file has a version vector which is a vector containing versions of the file on all other servers. This is incremented by each server when the file is updated locally. When all servers are synchronized all elements of the version vector will be the same. However, they will be different in case of disconnections. During reconnection, if a version vector of a server 1 is strictly greater than server 3, then server 1 has the latest copy. If server 1 and 2 has version vector [2,2,1] and server 3 has vector [1,1,2] then there is no strict ordering in which case there will be a write-write conflict and there needs to be a manual merge conflict.

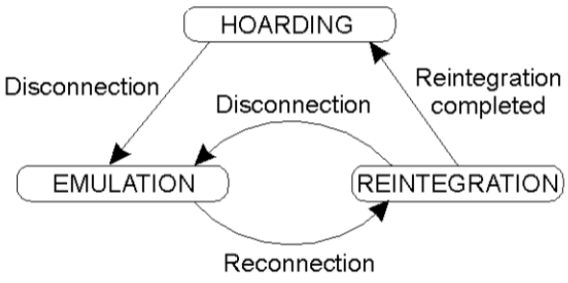


Figure 21.3: Disconnected operation in Coda

Figure 21.3 shows the state-transition diagram of a Coda client with respect to a volume. It has 3 states: HOARDING, EMULATION and REINTEGRATION. Whenever the client is connected to the network, it is in HOARDING mode. In this mode, the client does aggressive caching. Whenever the client is disconnected,

it makes a transition into the EMULATION mode. In this mode, all the file requests are serviced by the local cache and the version vectors get updated. The local cache file acts just like the actual file. On reconnection, the client goes to the REINTEGRATION mode. Resynchronization and version vector comparisons are made in this mode.

21.1.1.2 Transactional Semantics

Coda treats all operations as transactions which entails serializability, ACID properties etc. This is important since there are frequent disconnections and there could be conflicts on resynchronization.

21.1.1.3 Client Caching

Conda ensures cache consistency using callbacks which is a type of server-push consistency. Any update to a file by a client is sent to all other clients so the clients update their caches.

21.1.2 xFS (No Server FS)

This is a decentralized serverless file system. So there are no explicit servers and clients. Nodes can be both servers and clients. The resources are shared by all nodes. So instead of fetching data from a server, nodes have to find where a particular piece of data is located and then fetch it. An example of what the nodes look like is shown in Figure 21.4

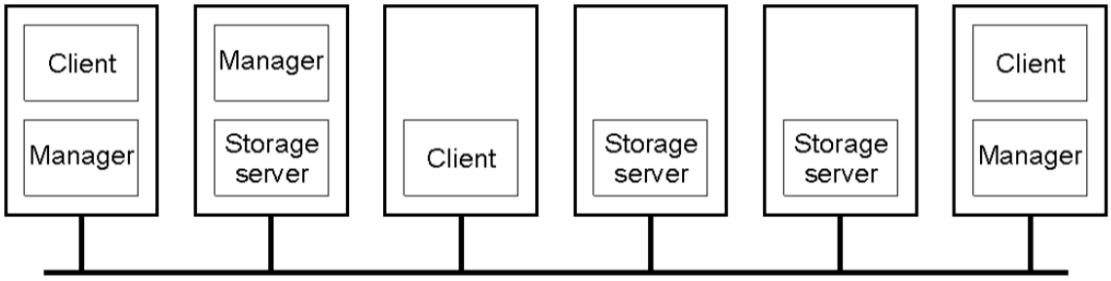


Figure 21.4: An example of nodes in xFS

xFS is built on two fundamental file systems: RAID and Log-structured File System as explained below.

21.1.2.1 RAID

RAID stands for Redundant Array of Independent Disks. In RAID based storage, files are striped across multiple disks. Disk failures are to be handled explicitly in case of a RAID based storage. Fault tolerance is built through redundancy.

Figure 21.5 shows how files are stored in RAID. d1,...d4 are disks. Each file is divided into blocks and stored in the disks in a round robin fashion. So if a disk fails, all parts stored on that disk are lost. It has an advantage that file can be read in parallel because data is stored on multiple disks and they can be read at

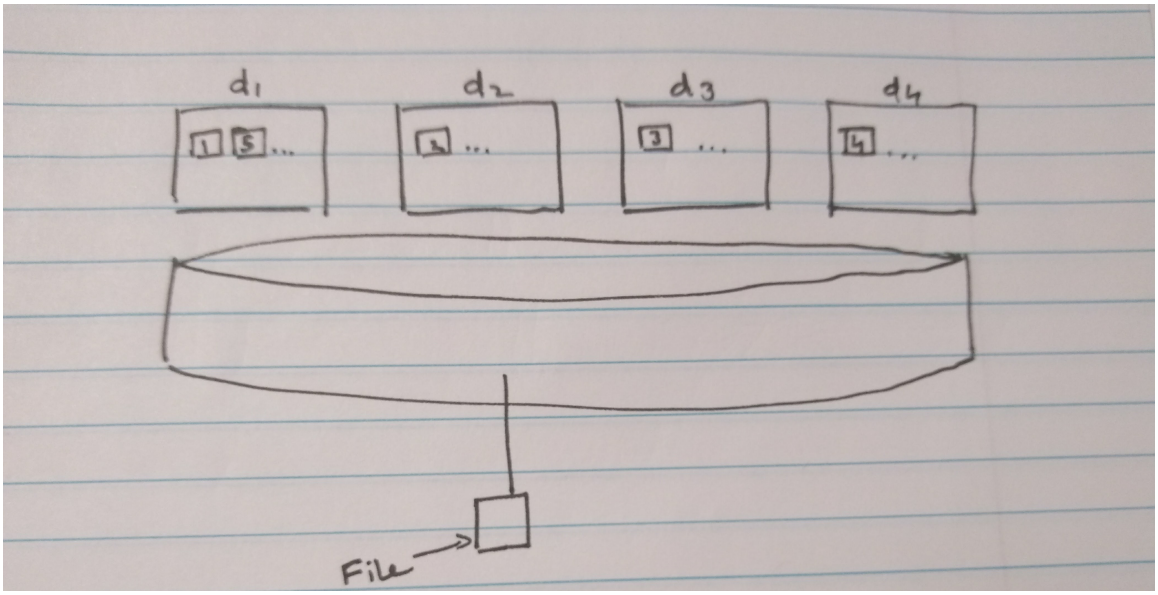


Figure 21.5: Striping in RAID

the same time. Secondly, storage is load balanced. If a file is popular and is requested more often, the load is evenly balanced across nodes. This also results in higher throughput.

A disadvantage of striping is failure of disks. The performance of this system depends on the reliability of disks. A typical disk lasts for 50,000 hours which is also knows as the Disk MTTF. As we add disks to the system, the MTTF drops as disk failures are independent.

Reliability of N disks = Reliability of 1 disk $\div N$

We implement some form of redundancy in the system to avoid disadvantages caused by disk failures. Depending on the type of redundancy the system can be classified into different groups:

21.1.2.2 RAID 1 (Mirroring)

From figure 21.6, we can see that in RAID 1 each disk is fully duplicated. Each logical write involves two physical writes. This scheme is not cost effective as it involves a 100% capacity overhead.

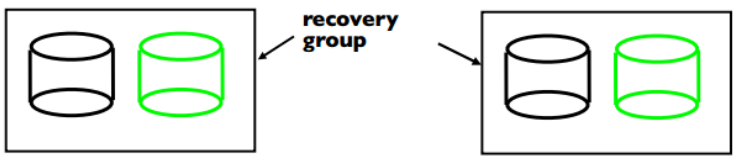


Figure 21.6: RAID 1

21.1.2.3 RAID 4

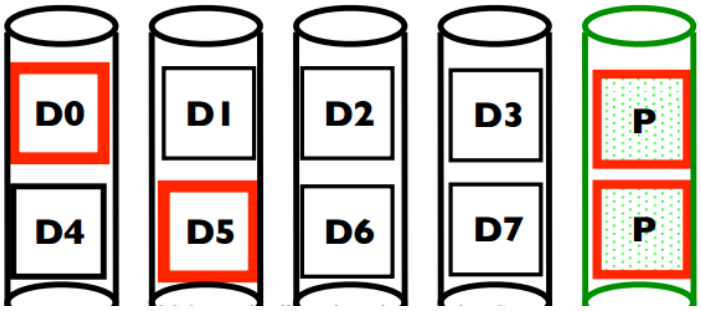


Figure 21.7: RAID 4

This method uses parity property to construct `ECC` (Error Correcting Codes) as shown in Figure 21.7. First a parity block is constructed from the existing blocks. Suppose the blocks D_0, D_1, D_2 and D_3 are striped across 4 disks. A fifth block (parity block) is constructed as:

$$P = D_0 \oplus D_1 \oplus D_2 \oplus D_3 \tag{21.1}$$

If any disk fails, then the corresponding block can be reconstructed using parity. For example:

$$D_0 = D_1 \oplus D_2 \oplus D_3 \oplus P \tag{21.2}$$

This error correcting scheme is one fault tolerant. Only one disk failure can be handled using RAID 4. The size of parity group should be tuned so as there is low chance of more than 1 disk failing in a single parity group.

21.1.2.4 RAID 5

One of the main drawbacks of RAID 4 is that all parity blocks are stored on the same disk. Also, there are k + 1 I/O operations on each small write, where k is size of the parity block. Moreover, load on the parity disk is sum of load on other disks in the parity block. This will saturate the parity disk and slow down entire system.

In order to overcome this issue, RAID 5 uses distributed parity as shown in Figure 21.8. The parity blocks are distributed in an interleaved fashion.

Note: All RAID solutions have some write performance impact. There is no read performance impact.

RAID implementations are mostly on hardware level. Hardware RAID implementation are much faster than software RAID implementations.

21.1.2.5 Log-Structured FS (LFS)

Purpose of LFS is to write data as a structured log. We keep appending blocks on disks as we do in a log file. This results in a very high write throughput.

In practice, we use a memory cache to increase efficiency as I/O operations on disks takes time. However, due to this cache, the traffic seen by the disks are predominantly writes, and few reads (caused by cache misses). Hence, we can optimize disks for write traffic. This is the idea behind LFS.

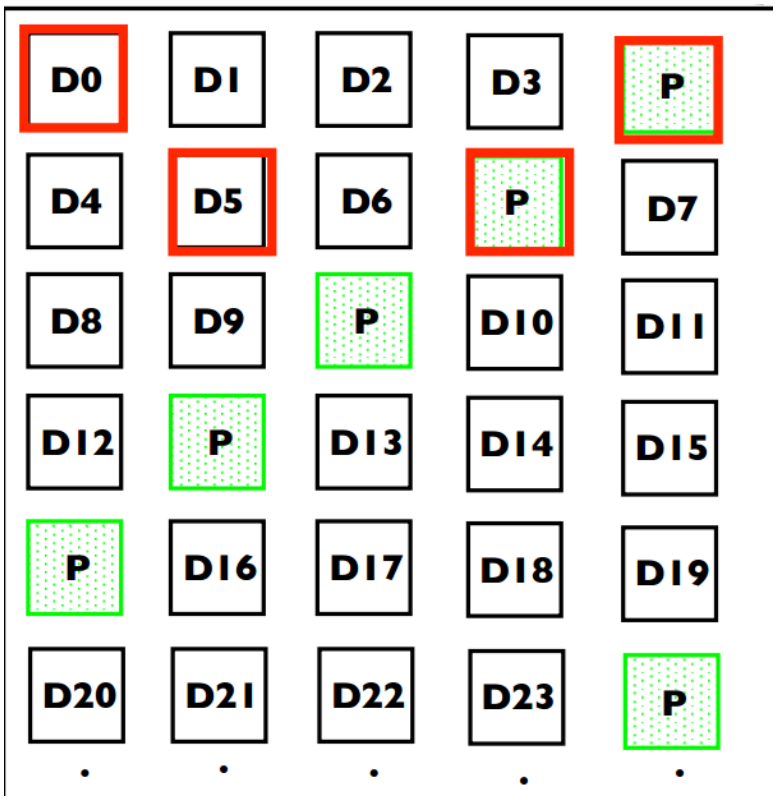


Figure 21.8: RAID 5

In LFS, we keep on writing sequentially. This minimizes seek time for a magnetic disk, hence increases the throughput. If a read comes after a write, we have to seek the block to be read which decreases the performance. Therefore, LFS works well if the I/O operations are mostly writes.

This advantage is only for magnetic disks and is not for SSDs, as there are no moving parts. SSDs provides random access.

In LFS, when we update a block, we write a new block, and mark the previous block as garbage. So a disadvantage of LFS is that we have to do garbage collection, which requires seek overheads. Also, it makes meta-data management complicated, as we have to update file Inode every time we update a block.

21.1.2.6 xFS and RAID

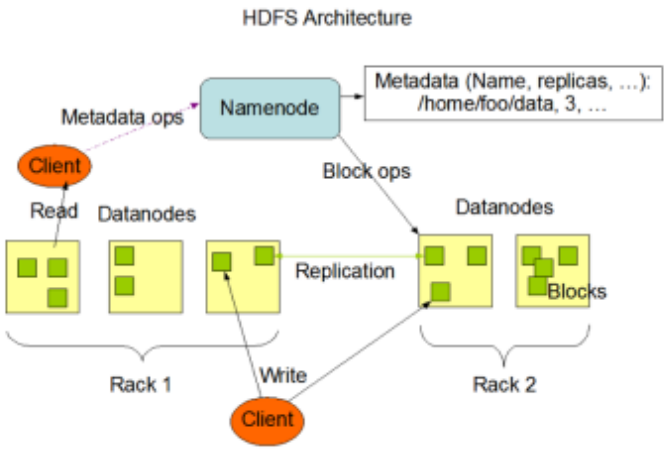
xFS uses software RAID by having multiple disks across multiple machines over a network. Small writes are expensive since updating a single block results in updating the parity which requires reading of all the other data blocks.

Moreover, xFS writes data using LFS. xFS implements striping which gives parallelism, RAID which gives fault-tolerance, and LFS wich provides high write throughput.

21.1.3 HDFS

Hadoop Distributed File System (HDFS) is used for processing very large datasets. HDFS files system has following goals:

- Fault-tolerant - System needs fault tolerant disks to store the large datasets.
- Streaming Data access - FS should provide batch processing rather than interactive usage.
- Size of the datasets are very large.
- Typically datasets are going to see WORM workloads. Which mean Write Once Read Many. This is because datasets are created only once, and is not changed afterwards, and only read many times.
- As the size of datasets are too large, we prefer moving computation instead of data.



HDFS uses large block size of around 64MB (Typical FS has 4KB block). Normally, files are much larger than block size, and large block size helps in reducing the seek overhead.

HDFS does not use RAID, but simply replicates every file 3 times (default value) and stored on random machines. This is a 2-fault tolerant system.

In fig **??**, we have meta data nodes which keep information about the datanodes.

21.1.4 GFS

Google File Systems is similar to HDFS. Each file is replicated on 3 different machines. A metadata nodes contains index of physical machines that stores a file.

File chunks are stored on standard linux files system. These have large block sizes which are tracked by GFS.

21.1.5 S3

There are storage systems which store a objects instead of a file. In object files system, we have key (hash of object) against which the object is mapped.

S3 also doesn't uses RAID. It replicates objects across data center. Moreover, there is no naming scheme like a FS. Storage is flat. And we are required to use HTTP to query required objects.