CMPSCI 677 Distributed Operating Systems **Spring 2022**

Lecture 23: April 25

Lecturer: Prashant Shenoy *Scribe:* ***Jeff Mao***

In this lecture, continue the topic "Distributed File Systems", Case Study: Coda File System and xFS

23.1 Coda Overview

23.1.1 DFS designed for mobile clients

- Nice model for mobile clients who are often disconnected

- Use file cache to make disconnection transparent
- At home, on the road, away from network connection

23.1.2 Code supplements file cache with user preferences

- E.g., always keep this file in the cache
- Supplement with system learning user behavior

23.1.3 How to keep cached copies on disjoint hosts consistent?

- In mobile environment, "simultaneous" writes can be separated by hours/days/weeks

Question: What is coda using a remote access model or an upload download model?

Answer: It's a little bit of both. When you're connected, your changes can be uploaded or sent to the server immediately, but you always have a cash. So when you are disconnected, you're essentially just working with whatever files subcaste, in which case you it looks like an upload download model. So the answer is it actually depends on whether you're the state of.

23.1.4 File Identifiers

- Each file in Coda belongs to exactly one volume
  - Volume may be replicated across several servers
  - Multiple logical(replicated) volumes map to the same physical volume
  - 96 bit file identifier = 32 bit RVID + 64 bit file handle

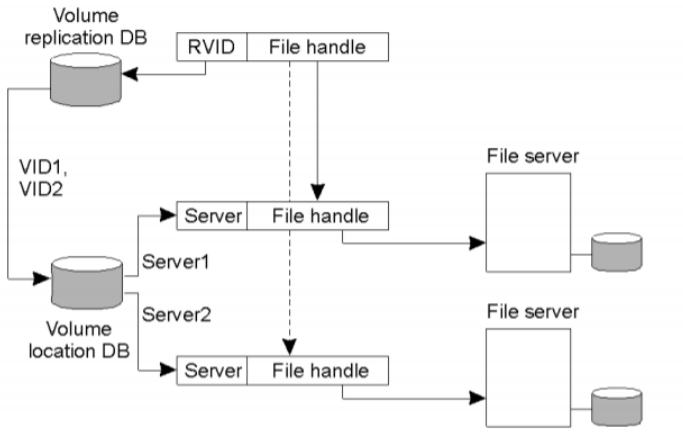

Figure 23.1: Coda architecture

23.1.5 Server Replication

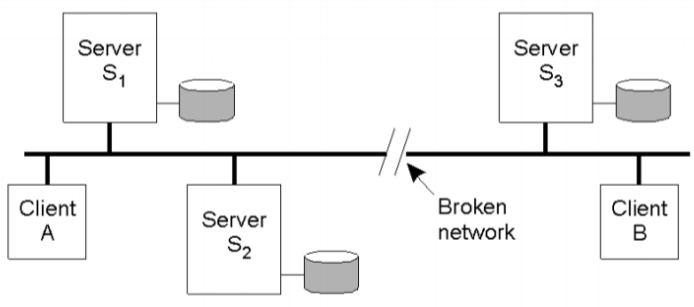

Figure 23.2: Server replication issues in Coda

- Use replicated writes: read-once write-all
  - Writes are sent to all AVSG(all accessible replicas)
- How to handle network partitions?
  - Use optimistic strategy for replication
  - Detect conflicts using a Coda version vector
  - Example: [2, 2, 1] and [1, 1, 2] is a conflict =>manual reconciliation

23.1.6 Disconnected Operation

- The state-transition diagram of a Coda client with respect to a volume.
- Use hoarding to provide file access during disconnection.

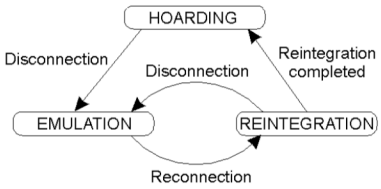

Figure 23.3: Disconnected operation in Coda

- Prefetch all files that may be accessed and cache(hoard) locally
- if AVSG=0, go to emulation mode and reintegrate upon reconnection

23.1.7 Transactional Semantics

- Notwork partition: part of network isolated from rest
  - Allow conflicting operations on replicas across file partitions
  - Reconcile upon reconnection
  - Transactional semantics => operations must be serializable
    - Ensure that operations were serializable after thay have executed
  - Conflict => force manual reconciliation

23.1.8 Client Caching

- Cache consistency maintained using callbacks

23.2 xFS

23.2.1 Overview of xFS

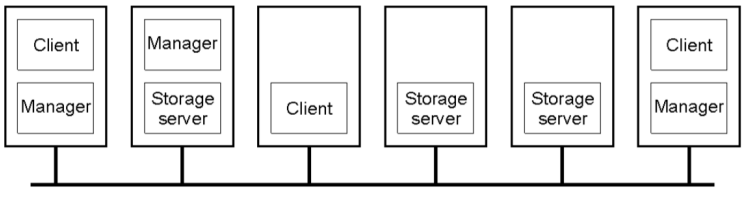

Figure 23.4: An example of nodes in xFS

- Key Idea: fully distributed file system [serverless file system]
  - Remove the bottleneck of a centralized system

- xFS: x in "xFS" => no server

- Designed for high-speed LAN environments

XFS combines two main concepts ; RAID - Redundant Array of Inexpensive Disks) and Log Structured File Systems (LFS). It uses a concept of Network Stripping and RAID over a network wherein, a file is partitioned into blocks and provided to different servers. These blocks are then made as a Software RAID file by computing a parity for each block which resides on a different machine.

In log structured File systems, data is sequentially written in the form of a log. The motivation for LFS would be the large memory caches used by the OS. Larger, the size of cache, more the number of cache hits due to reads, better will be the payoff due to the cache. The disk would be accessed only if there is a cache miss. Due to the this locality of access, mostly write requests would trickle to the disk. Hence, the disk traffic comes predominantly from write. In traditional hard drive disks, a disk head read or writes data . Hence, to read a block, a seeks needs to be done ie move the head to the right track on the disk.

How to optimize a file system which sees mostly write traffic ?

The basic insight is to reduce the time spent on seek and waiting for the required block to spin by. Every read/write request incurs a seek time and a rotational latency overhead. In general , random access layout is assumed for all blocks in the disk wherein the next block is present in an arbitrary location. This would require a seek time.

To eliminate this, a sequential form of writing facilitated by LFS can be used. The main idea of LFS is that we try to write all the blocks sequentially one after the other. Thus LFS essentially buffers the writes and writes them in contiguous blocks into segments in a log like fashion. This will dramatically improve the performance. Any new modification would be appended at the end of the current log and hence, overwriting is not allowed. Any LFS requires a garbage collection mechanism to de-fragment and clean holes in the log.

Hence, XFS ensures 1. fault tolerance - due to RAID, 2. Parallelism - due to blocks being sent to multiple nodes. 3. High Performance - due to Log structured organization.

In SSD's, the above mentioned optimization to log structures doesn't give any benefits since there are no moving parts and hence, no seek.

23.2.2 xFS Summary

- Distributes data storage across disks using software RAID and log-based network striping
  -RAID = Redundant Array of Independent Disks

- Dynamically distribute control processing across all servers on a per-file granularity
  - Utilizes serverless management scheme.

- Eliminates central server caching using cooperative caching
  - Harvest portions of client memory as a large, global file cache.

23.2.3 Array Reliability

- Reliability of N disks = Reliability of I Disk N

  50, 000 Hours 70 disks = 700 hours

  Disk system MTTF:Drops from 6 years to I month!

- Arrays(without redundancy) too unreliable to be useful!

23.3 RAID

23.3.1 RAID Overview

- Basic idea: files are ”striped” across multiple disks
- Redundancy yields high data availability
  - Availability: service still provided to user, even if some components failed
- Disks will still fail
- Contents reconstructed from data redundantly stored in the array
  - Capacity penalty to store redundant info
  - Bandwidth penalty to update redundant info

23.3.1.1 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.

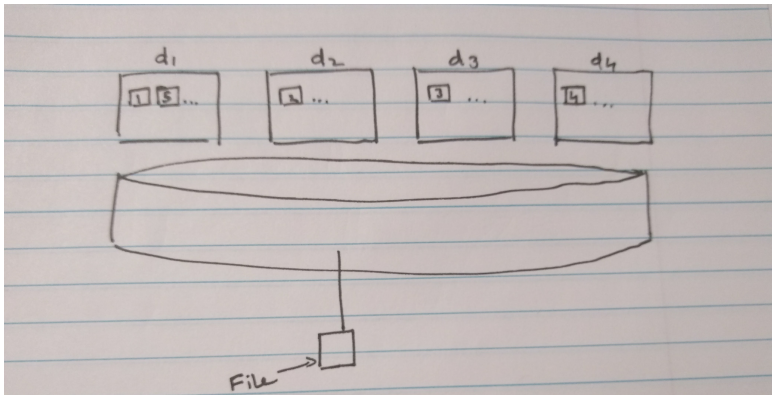

Figure 23.5: Striping in RAID

Figure 23.5 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

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:

23.3.1.2 RAID 1 (Mirroring)

From figure 23.6, 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.

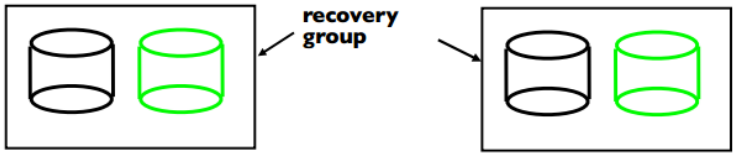

Figure 23.6: RAID 1

23.3.1.3 RAID 4

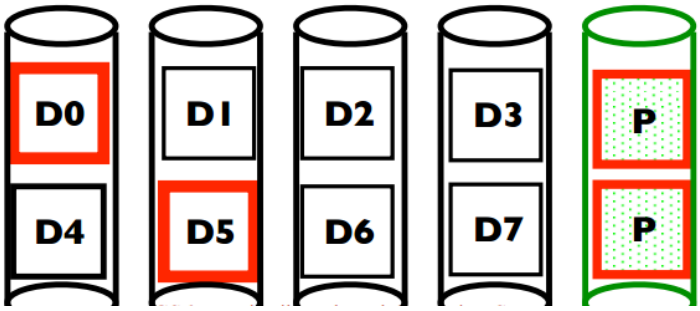

Figure 23.7: RAID 4

This method uses parity property to construct `ECC` (Error Correcting Codes) as shown in Figure 23.7. 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:

$$P = D_0 \oplus D_1 \oplus D_2 \oplus D_3 \tag{23.1}$$

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

$$D_0 = D_1 \oplus D_2 \oplus D_3 \oplus P \tag{23.2}$$

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.

23.3.1.4 RAID 5

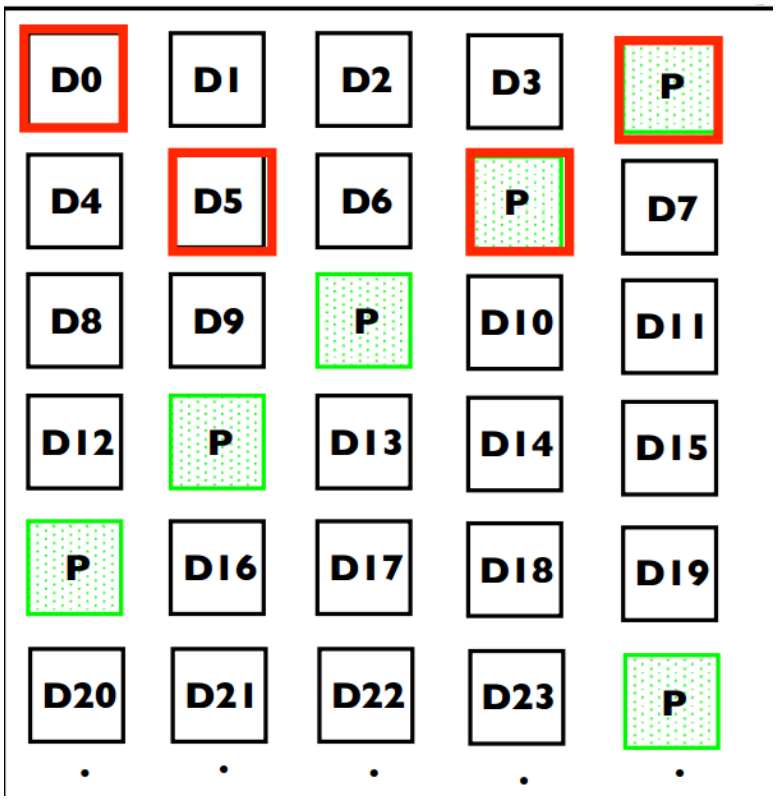

Figure 23.8: RAID 5

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 23.8. 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.

23.3.2 xFS uses software RAID

- Two limitations

- Overhead of parity management hurts performance for small writes
  - Ok, if overwriting all N-1 data blocks
  - Otherwise, must read old parity+data blocks to calculate new parity
  - Small writes are common in UNIX-like systems
- Very expensive since hardware RAIDS add special hardware to compute parity

23.3.3 Log-structured FS

- Provide fast writes, simple revovery, flexible file location method
- Key Idea: buffer writes in memory and commit to disk in large, contiguous, fixed-size log segments
  - Complicates reads, since data can be anywhere
  - Use per-file inodes that move to the end of the log to handle reads
  - Uses in-memory imap to track mobile inodes
    - Periodically checkpoints imap to disk
    - Enables ”roll forward” failure recovery
  - Drawback: must clean ”holes” created by new writes

23.3.4 Combine LFS with Software RAID

Log written sequentially are chopped into blocks which a parity groups. Each parity group becomes a server on a different machine in a RAID fashion

23.4 HDFS - Hadoop Distributed File system

It is designed for high throughput - very large datasets. It optimizes the data for batch processing rather than interactive processing. HDFS has a simple coherency model in which it assumes a WORM (Write Once Read Many) model. In WORM, file do not change and changes are append-only.

23.4.1 Architecture

There are 2 kinds of nodes in HDFS ; Data and Meta-data nodes. Data nodes store the data whereas, meta-data keeps track of where the data is stored. Average block size in a file system is 4 KB. In HDFS, due to large datasets, block size is 64 MB.Replication of data prevents disk failures. Default replication factor in HDFS is 3.

23.5 GFS - Google File System

Master node acts as a meta-data server. It uses a file system tree to locate the chunks (GFS terminology for blocks). Each chunk is replicated on 3 nodes. Each chunk is stored as a file in Linux file system.

23.6 Object Storage Systems

- Use handles(e.g., HTTP) rather than files names
  - Location transparent and location independence
  - Separation of data from metadata
- No block storage: objects of varying sizes
- Uses
  - Archival storage
    - can use internal data de-duplication
  - Cloud Storage: Amazon S3 service
    - uses HTTP to put and get objects and delete
    - Bucket: objects belong to bucket/partitions name space