CMPSCI 677 Distributed & Operating Systems **Spring 2018**

Lecture 20: April 11

Lecturer: Prashant Shenoy *Scribe: **Ajinkya Indulkar***

20.1 Locating and Accessing data

With hierarchical proxy caching, the latency is high in case of cache miss. A different architecture is shown in Figure 20.1. The clients are white boxes and the green circles are caches. If there is a cache hit, it works normally but in case of a cache miss, a cache has to get the data from another cache. But how to know which cache has that data? The answer is that each cache keeps a table that has a list of all data the cache has locally and also a table that tells what data the nearby proxies have. In case of a request we first check the local cache and if it is not found locally, the proxy gets the data from a nearby proxy. The system then should ensure that the tables are up to date. For example, a client requests page A, and it is a local miss. It checks its table that says that the page A is at node X. So it then fetches page A from node X. If the page is not available nearby, the page is fetch from the server as shown in the Figure 20.1 on the right. There can be at most 2 hits as shown in the figure. And there can be at most 2 misses because it might happen that the table is not up to date on a node and when it tries to fetch data from another node, it might not have it, in which case it will be fetched from the server.

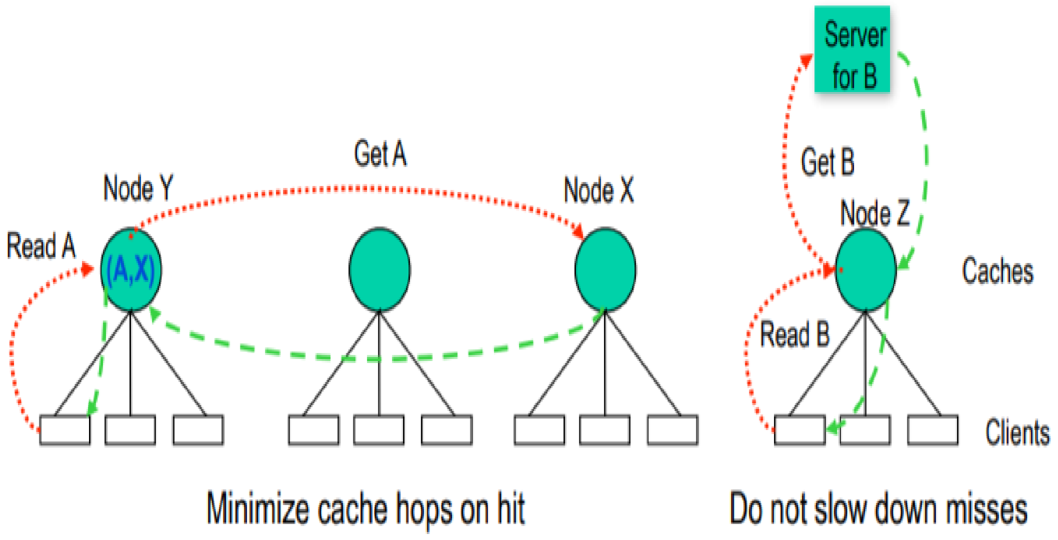

Figure 20.1: Alternate architecture

20.2 Content Distribution Network (CDN)

A CDN provides caching service to websites that have a lot of traffic. So when a client makes a request some content may be served from a CDN. When some content on the server changes, it has to be propagated to the proxies. CDNs cache static content for a website. They don't cache dynamic content. Additionally when a user makes a request it the CDN has to ensure that the nearest proxy handles the request so the latency is minimal. Akamai does this using DNS based location. Figure 20.2 shows the architecture of a typical CDN. In this case, we assume the images are served by the CDN and the HTML pages are served by the server.

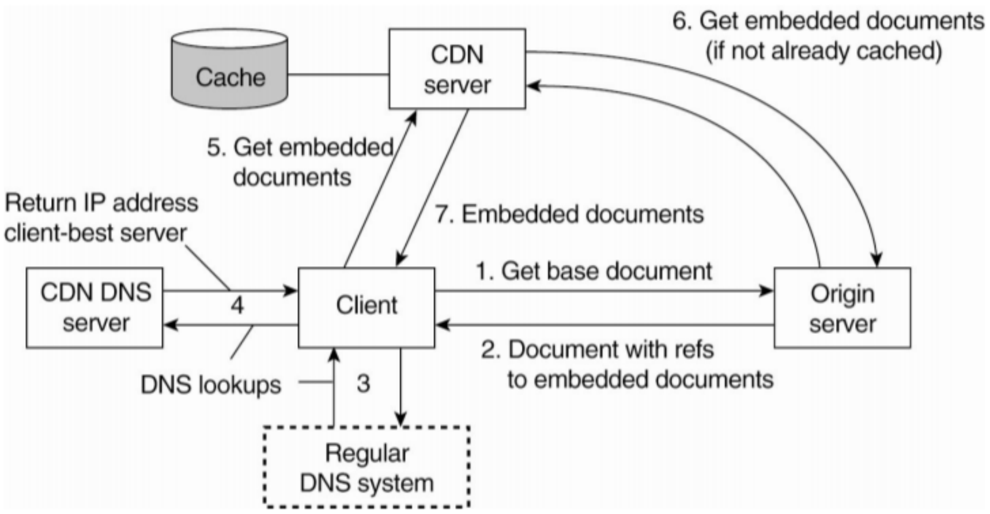
Cache
CDN server
6. Get embedded documents (if not already cached)
5. Get embedded documents
Return IP address client-best server
7. Embedded documents
CDN DNS server
4
Client
1. Get base document
Origin server
2. Document with refs to embedded documents
DNS lookups
3
Regular DNS system

Figure 20.2: CDN architecture

20.3 Distributed File Systems (DFS)

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

20.3.1 Coda

Coda is a research FS designed at CMU. It was designed for Mobile devices. It was **disconnection transparent** which means users would see the all files even if the system was temporarily disconnected. In case of disconnection, there is a local copy of the files. 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.

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 20.3. 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.

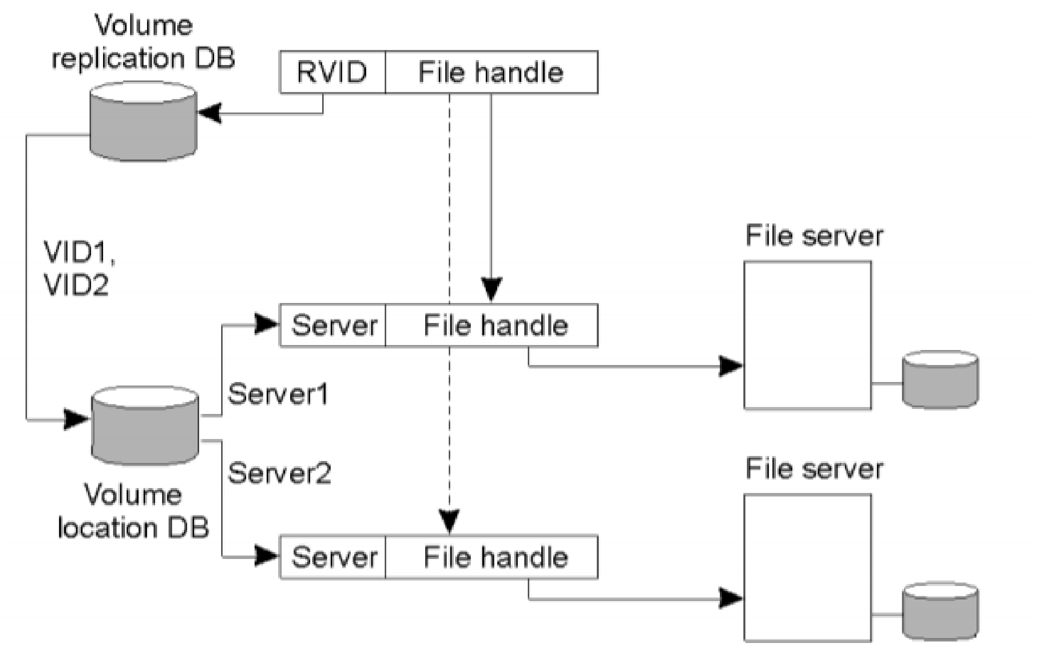

Figure 20.3: Coda architecture

20.3.1.1 Issues with server replication

Figure 20.4 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. 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 has version vector [2,2,1] and server 3 has vector [1,2,2] 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 20.5 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.

20.3.1.2 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.

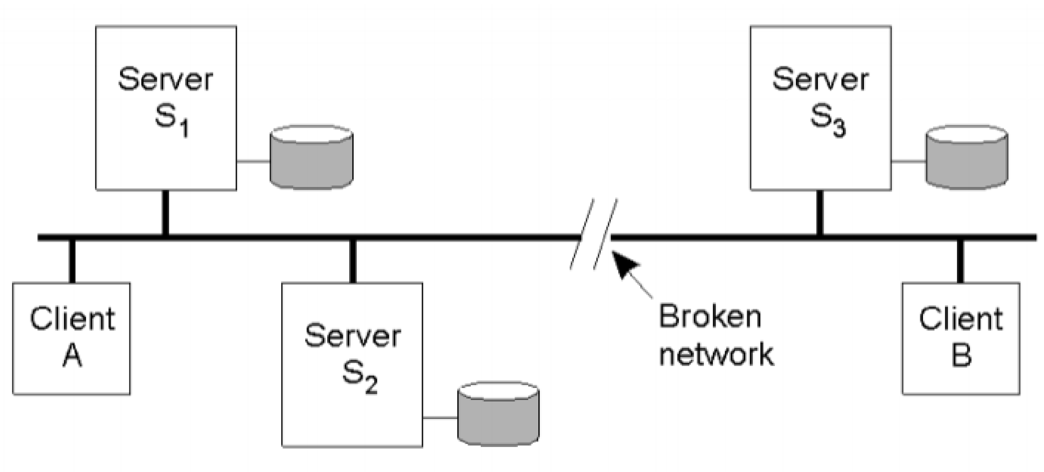

Figure 20.4: Server replication issues in Coda

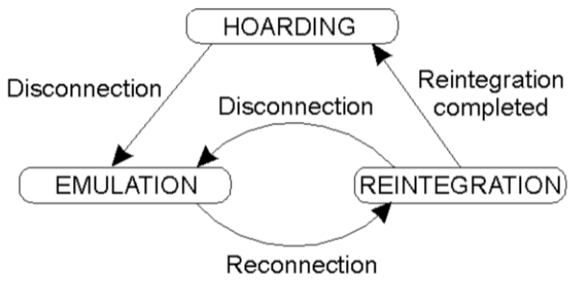

Figure 20.5: Disconnected operation in Coda

20.3.1.3 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.

20.3.2 xFS

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 20.6

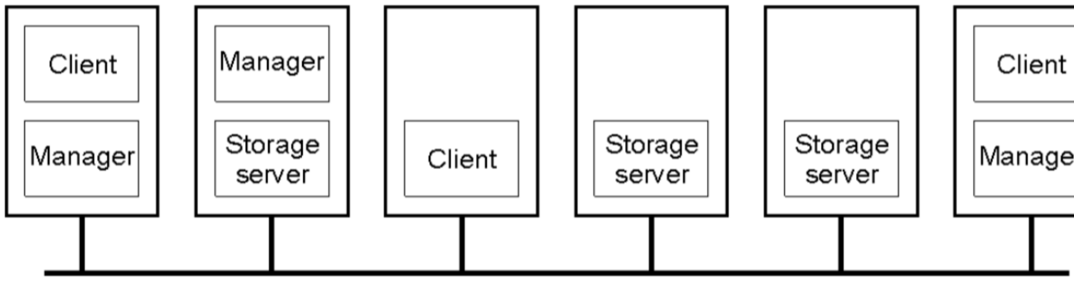

Figure 20.6: An example of nodes in xFS

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

20.3.3 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 20.7 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.

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 can also have redundancy in the system. Depending on the type of redundancy the system can be classified into different groups:

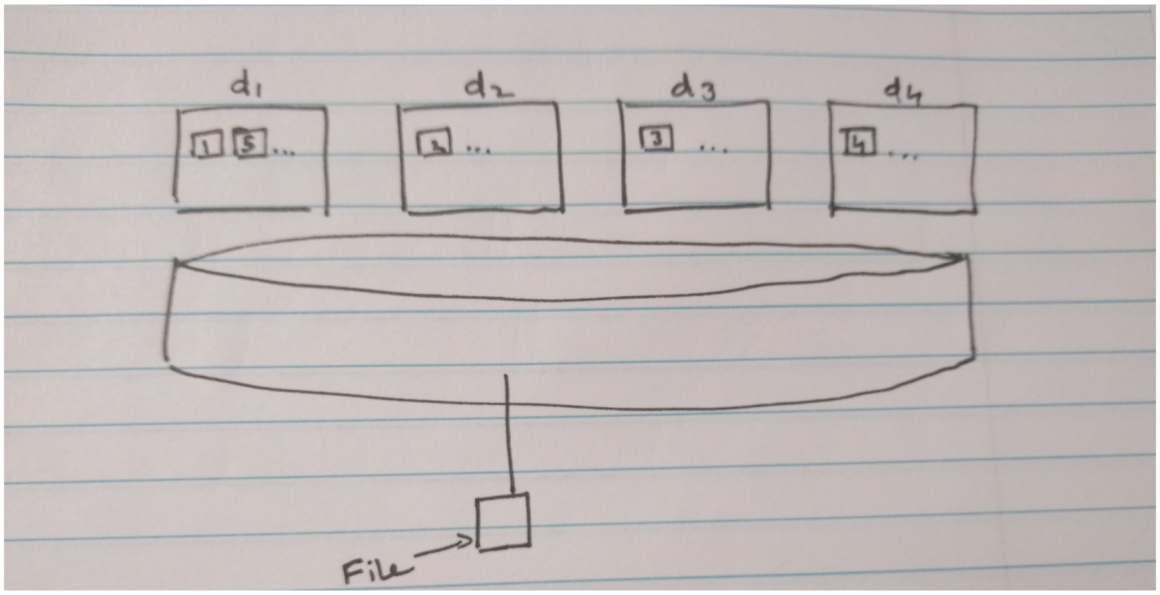

Figure 20.7: Striping in RAID

20.3.3.1 RAID 1 (Mirroring)

In this method, 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.

20.3.3.2 RAID 4

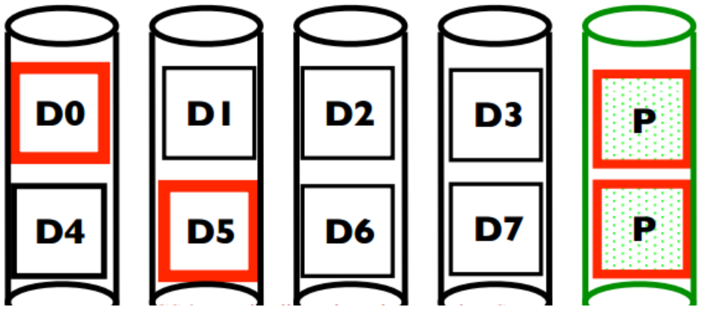

Figure 20.8: RAID 4

This method uses parity property to construct `ECC` (Error Correcting Codes) as shown in Figure 20.8. 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:

$$P = D_0 \oplus D_1 \oplus D_2 \oplus D_3 \tag{20.1}$$

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

$$D_0 = D_1 \oplus D_2 \oplus D_3 \oplus P \tag{20.2}$$

20.3.3.3 RAID 5

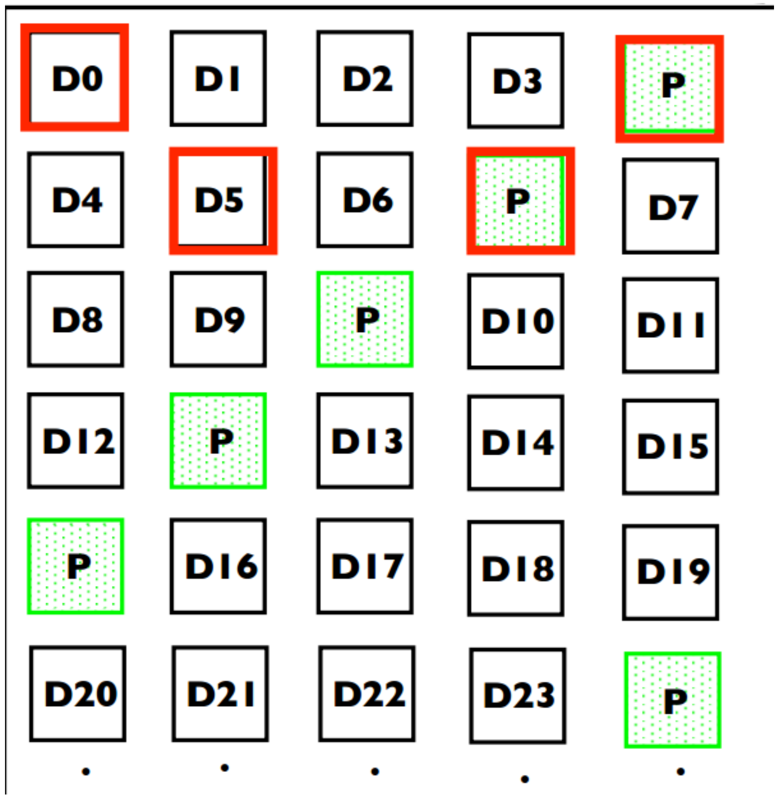

Figure 20.9: RAID 5

One of the main drawbacks of RAID 4 is that all parity blocks are stored on the same disk. Also the parity block has to be updated on each small write. In order to overcome this issue, RAID 5 uses distributed parity as shown in Figure 20.9. The parity blocks are distributed in an interleaved fashion.

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

20.3.4 xFS and RAID

xFS uses 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.