CMPSCI 677 Distributed Operating Systems **Spring 2025**

Lecture 23: April 28

Lecturer: Prashant Shenoy *Scribe:* ***Sylvia Imanirakiza (2025), Chaitali Agarwal (2024)***

23.1 NFS (contd)

23.1.1 Recap

NFS has a weak consistency model. Whenever a client application user modifies a file, the changes get written to the cache at the client machine and later on the client can send the changes to the server. Meanwhile, if the server receives a request for the same file from some other user it will send stale content.

23.1.2 Client Caching: Delegation

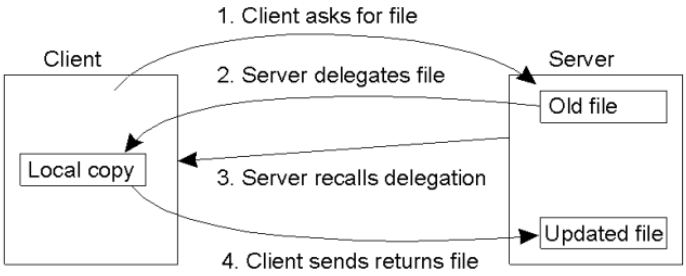

Figure 23.1: Delegation

NFS supports the concept of delegation as part of caching. The client receives a master copy of the file to which the client can make updates. Upon completion, the client can send the file back to the server. This is similar to the concept of upload/download model. Thus, the server is delegating the file to the client so that the client can have a local copy. If another client tries to access the file, the server recalls the delegation given to previous client. The previous client returns the file to the server and then the server uses the old model where multiple clients can access the file by read/write requests to the sever.

Question: When does the server decide to delegate the file?
Answer: Since this feature is stateful, it is only present in version 4. If the server is serving only one client then the server can delegate the file. Otherwise since the server is not the current owner of the file, the server can not delegate and thus has to use the old model. For example, files in the user's home directory can be delegated, whereas binaries of application programs can not be delegated as multiple users might access them.

Question: Is there a way to periodically update the server as in case of client failure the files may get lost?
Answer: It is possible for the client to flush the changes to the server in the background while it still holds the master copy.

23.1.3 RPC Failures

23.2 Coda Overview

23.2.1 DFS designed for mobile clients

- Nice model for mobile clients who are often disconnected

- Introduced in late 80-90s by CMU
- Use file cache to make disconnection transparent
- Designed for weakly connected devices which can be used at home, on the road, away from network connection
- Serves as a precursor to Cloud drives
- It supplements file cache with user preferences. E.g. always keep this file in the cache or Supplement with system learning user behavior.
- Uses replicated writes: Read once and write all. Writes are sent to all accessible replicas.

Question: 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 cache. So when you are disconnected, you're essentially just working with files caches, 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 the mobile device.

23.2.2 File Identifiers

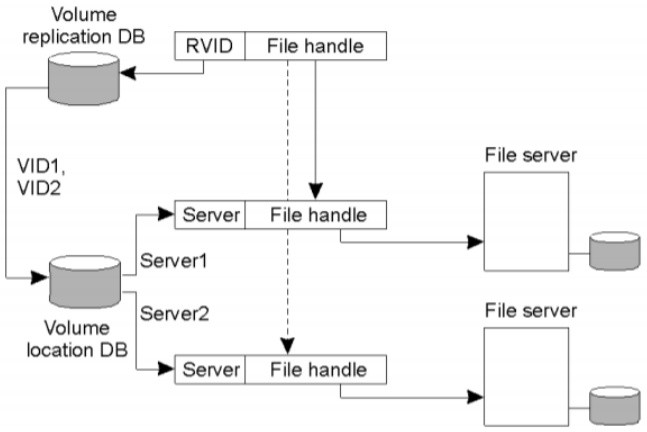

Figure 23.2: Coda architecture

- Each file in Coda belongs to exactly one volume. A volume could be a disk or a partition of a disk.

- Volume may be replicated across several servers. Identifiers include volume ID and file handle.
- Multiple logical(replicated) volumes map to the same physical volume
- 96 bit file identifier = 32 bit RVID + 64 bit file handle
- Each write increments the version number. Similar to versions maintained by git.

23.2.3 Server Replication

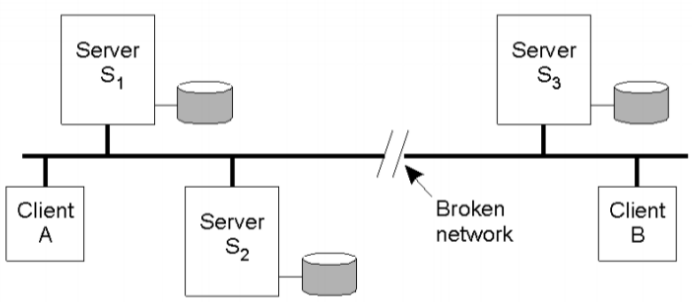

Figure 23.3: Server replication issues in Coda

Assume there are 3 servers and 2 clients connected over a network. In an ideal situation, the servers keep the copies of the files consistent. If there is a partition in the network, the servers no more have the same copy of the files. When network partition is fixed, the servers try to synchronize the files. If the files are different, there may not be any problems. Problem arises if the servers access the same files due to write-write conflicts.

- 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

Question: What is the size of the version vector?
Answer: The number of entries in the version vector is equal to the number of servers that have the copy of the file.

Question: If the file is being updated multiple times will the system keep incrementing the version?
Answer: It is possible. It will still give rise to the same kind of conflict.

Question: What does manual reconciliation mean?
Answer: It means that the user has to manually resolve the conflicts in the same way as the user is required to resolve merge conflicts in Git.

23.2.4 Disconnected Operation : Client disconnects from Server

- **Hoarding State:** Client is connected to the server and is actively downloading files into cache based on some prediction based on current usage of the user. The client tries to cache copies that the user is likely to access.
- **Emulation State:** When the client is disconnected from the server/internet and uses the cached copies.
- **Reintegration State:** Client is connected to the internet/server and merges its updates with the server.

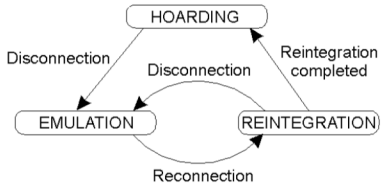

Figure 23.4: Disconnected operation in Coda

- The state-transition diagram of a Coda client with respect to a volume.
- Use hoarding to provide file access during disconnection.
  - Prefetch all files that may be accessed and cache(hoard) locally
  - if AVSG=0, go to emulation mode and reintegrate upon reconnection

23.2.5 Transactional Semantics

- Network 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

23.2.6 Client Caching

- Cache consistency maintained using callbacks

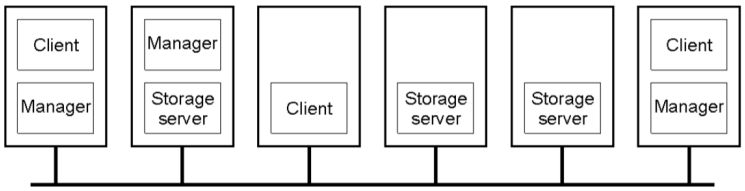

Figure 23.5: An example of nodes in xFS

23.3 xFS

23.3.1 Overview of 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
- All nodes participates in the File sharing

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.

23.3.2 RAID : 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.

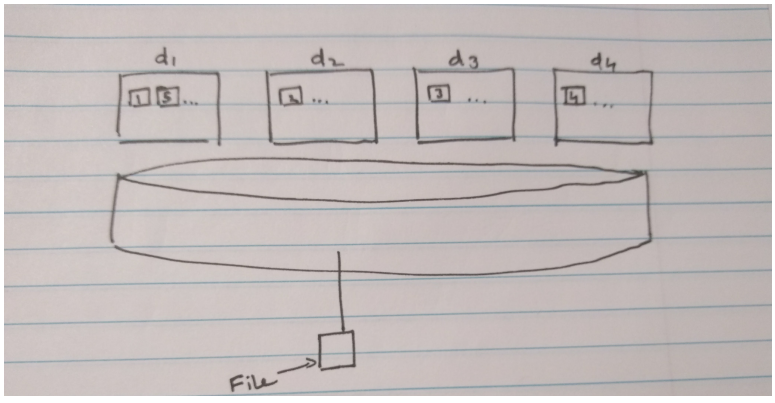

Figure 23.6: Striping in RAID

Figure 23.6 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.

MTTF : Mean time to failure. It is about 5-6 years for a disk.

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.

$$\text{Reliability of N disks} = \text{Reliability of 1 disk} \div N \tag{23.1}$$

$$\text{Probability of failure of a system} = (\text{1-p})^n \tag{23.2}$$

Consider a case where there are 70 disks in the system.

Reliability of system = 50, 000 Hours $\div 70 disks = 700 hours$

Disk system MTTF: Drops from 6 years to 1 month!

23.3.2.1 Advantages

- Load balanced across multiple disks
- Parallelizes the access to each disk and hence high throughput.

23.3.2.2 Disadvantages

- If a single disk fails, 1/N of the data of each file will be lost, without redundancy.
- The performance of this system depends on the reliability of disks.

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:

23.3.2.3 RAID 0

Doesn't have any redundancy. Only striping. Each files in stripped into multiple parts and stores on a separate disk.

23.3.2.4 RAID 1 (Mirroring)

From figure 23.7, 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.

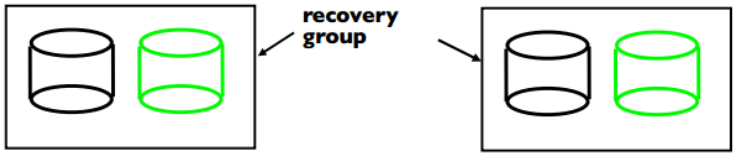

Figure 23.7: RAID 1

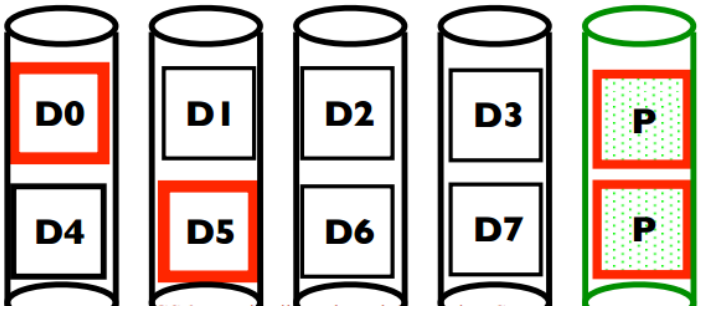

Figure 23.8: RAID 4

23.3.2.5 RAID 4

This method uses parity property to construct `ECC` (Error Correcting Codes) as shown in Figure 23.8. All parity blocks are stored on the same disk. 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:

$$P = D_0 \oplus D_1 \oplus D_2 \oplus D_3 \tag{23.3}$$

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

$$D_0 = D_1 \oplus D_2 \oplus D_3 \oplus P \tag{23.4}$$

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.

Smaller writes are expensive, as the corresponding parity would need to be changed and it would require reading the other members of the parity group. This involves N+2 operations; N block reads and 2 block writes (Read-modify-write cycle).

Writes seen by the parity disk is N times the writes seen by the other disks. Therefore, the parity disk will quickly get overloaded if you have a write-intensive system.

Question: Where is the information about which files are on which disk?
Answer: The hardware controller serves the request internally to identify which blocks are stored on which disk.

Question: In RAID, hardware controller keeps a track of data blocks and parity, what happens if controller fails?
Answer: There will be problems in accessing the disk. That may be a point of failure. In case of Software RAID this issue will not occur.

Question: Won't the cost of accessing files increase since all disks are being accessed?
Answer: There are two ways to access a file, either block by block or accessing the whole file. If a request is made to access the whole file, in the above figure, eight requests to different disks would have been made, making it parallel. If the entire file would have been saved on the same disk, it would have resulted in eight requests being made to the single disk, which makes it sequential. Thus, accessing multiple disks does not necessarily make it expensive.

Question: Can you read D0 and the parity,P and xor it to get the middle blocks?
Answer: This would be an a good optimization idea. But there is still additional overhead given there is still a read-modify-write cycle.

23.3.2.6 RAID 5

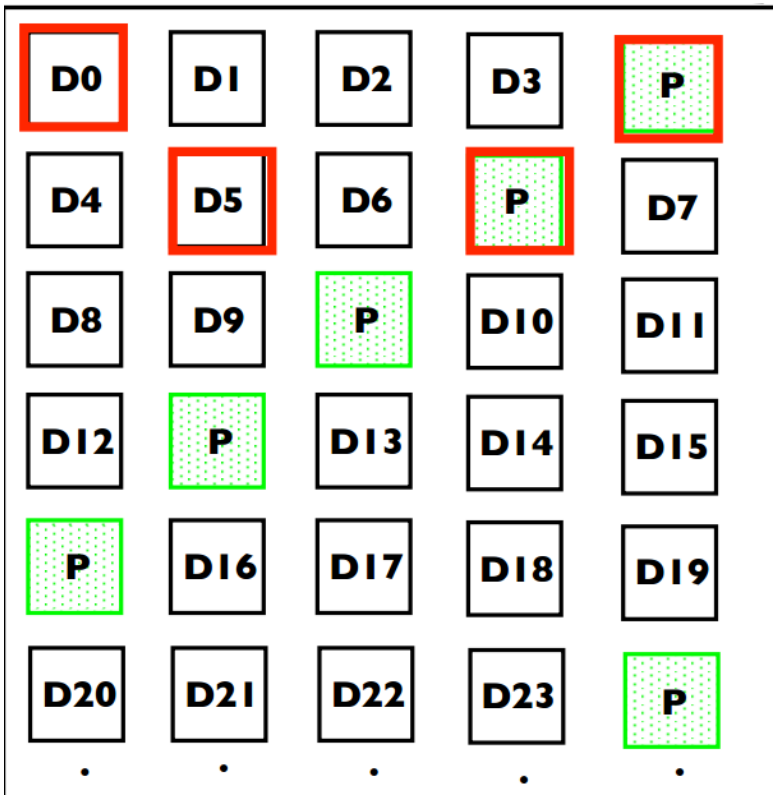

Figure 23.9: RAID 5

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 23.9. The parity blocks are distributed in an interleaved fashion.

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

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

software RAID implementations.

Question: Can a file not be stored in the same disk as its parity?
Answer: The parity and the file stripe wouldn't be in the same parity group. It can still handle 1 disk failure.

23.3.2.7 RAID Summary

- 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

23.3.3 LFS: Log File Structure

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 i.e. 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.

Question: Is there an overhead to maintain lookup as block of the files need to be tracked?
Answer: There is higher overhead to maintain the lookup. For every write, the data gets appended, so it

is meant to be for high write workloads. Metadata of the files is also written to the log. In case of lookups, the metadata has to be accessed. Hence there is high overhead.

Question: Can the writes be cached?
Answer: Reads are directly cached. Writes are cached in batches i.e. a batch of writes are written as an append-only log.

Question: Is LFS one server?
Answer: LFS are traditionally designed as single disk system. Here, they are combined with xFS. The logs are stripped across machines.

23.3.3.1 Log-structured FS Summary

- Provide fast writes, simple recovery, 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 xFS Summary

- Distributes data storage across disks using software RAID and log-based network striping.
- 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.

23.3.5 xFS uses software RAID and LFS

xFS uses software RAID because the disks are not on one machine. Striping occurs across a network and uses software RAID to update the parity.

- 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

23.3.6 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

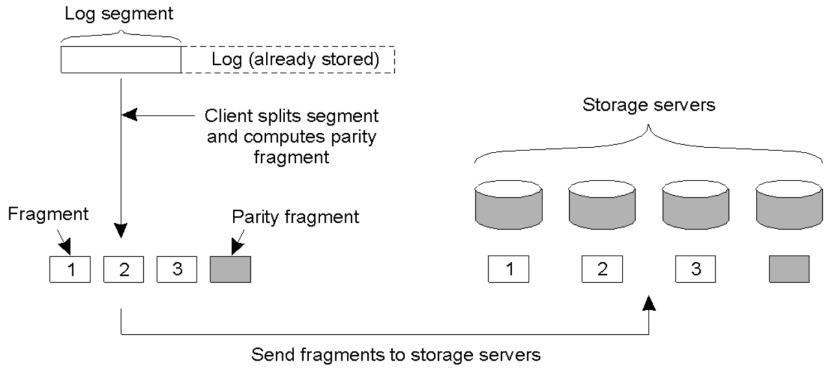

Figure 23.10: Combining LFS with Software RAID

Question: In RAID 5, if one disk has failed, how do you know where the parts of the file are stored? **Answer:** RAID is operating below the file system. It doesn't know anything about files, it only knows about blocks. The file system is the one that knows the metadata that keeps track where the data blocks are stored.

23.4 HDFS - Hadoop Distributed File system

- It is designed for high throughput - very large datasets. Optimized for read only applications.
- HDFS is designed with specific goals:
  - Fault Tolerant: It has to have fault tolerance built in.
  - Streaming data access: It is designed for batch processing rather than interactive processing.
  - Large datasets: scale to hundreds of nodes.
  - Simple coherency model: 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.
  - Move computation to the data when possible.

23.4.1 Architecture

There are 2 separate 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.

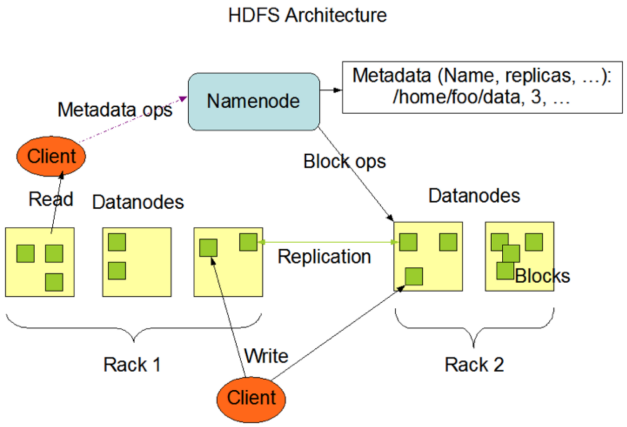

Figure 23.11: HDFS Architecture

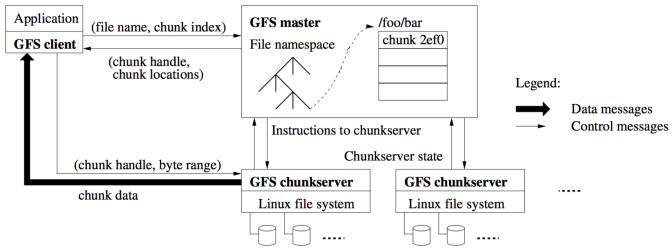

Figure 23.12: GFS Architecture

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 (similar to HDFS and GFS)
- 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

Question What does "Location transparent and location independence" imply?
Answer: Think of a handle as a large numeric ID, looking at the handle you will not know where the object is stored. This explains the location transparency. For the location independence, Internally the system can decide to move the object somewhere else and the handle name remains the same.

In Amazon's case, they replicate the buckets across data centers. They assume entire data centers can fail. They keep the objects replicated geographically and the object handle will not tell you the location and internally will figure out the closest available location and get that data for the user.