CMPSCI 677 Distributed Operating Systems **Spring 2024**

Lecture 24: May 06

Lecturer: Prashant Shenoy *Scribe: Jeff Mao, Riya Singh(2023), Niti Mangwani(2024)*

24.1 NFS (contd)

24.1.1 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.

24.1.2 Client Caching: Delegation

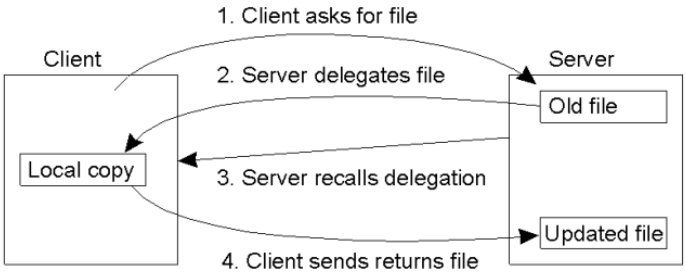

Figure 24.1: Delegation

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.

24.1.3 RPC Failures

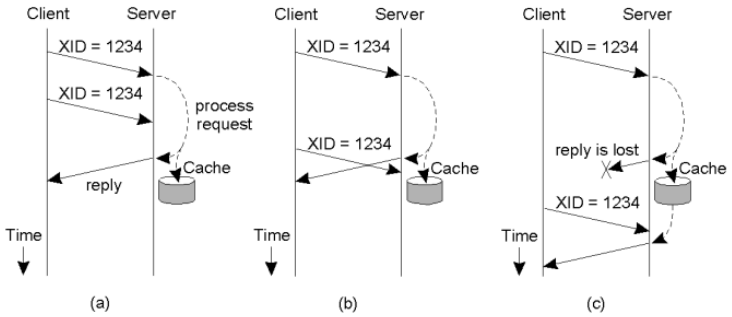

Figure 24.2: RPC Failures

For RPCs being used over TCP, TCP will take care of retransmissions between client and server. For RPCs over UDP, client and server can decide how to deal with lost requests and replies. Every RPC request is associated with an ID. Upon receiving an RPC request from the client, the server will maintain the request and response for that request in its cache. If the client resends the request and the reply was lost, the server will simply send the reply from the cache, instead of executing it again.

Question: What is the utility of UDP over TCP?
Answer: UDP is faster than TCP as there is three-way handshake in TCP. In LANs, where probability of loss is low, RPCs can be sent over UDP. Over WANs or noisy LANs TCP may be preferred.

Question: For how long can the reply be kept in the cache?
Answer: It depends on the application. Practically, after some unsuccessful tries within an hour, the client may assume that the server is down. So the replies can be cached for some hours.

Question: Is caching reply specific to some version of NFS?
Answer: It is not specific to some verison of NFS. In NFS v1, there was no concept of RPCs over TCP. Thus, this method was used for RPCs over UDP. Currently, with the advent of RPCs over TCP, this method is not needed to be used.

Question: Is the cache needed only so that the requests are idempotent?
Answer: Yes. For example, if requests are changing files, it might incur problems if they are not idempotent and if requests are needed to be idempotent the cache is required.

24.1.4 Security

Versions 1, 2 and 3 of NFS relied on a simple security model. Every request is sent with user ID and group ID. The server checks for the file permissions on the basis of user ID and server ID. This ensures only authenticated users can access the file. One drawback of this is that the channel between client and server, however, is not still secure. If an adversary intercepts the network traffic, the contents of a secure file may be exposed. In version 4, the concept of secure RPCs was introduced. Every RPC client stub sends the request to the security layer which encrypts the request before sending. Thus, file contents can not be read on the network.

Question: Client can send user ID and group ID, but how does the server know if it is authentic?
Answer: As long as the server trusts the OS on the client the server knows the user ID and group ID are authentic. However, if the OS is corrupted/hacked the server can not trust the client

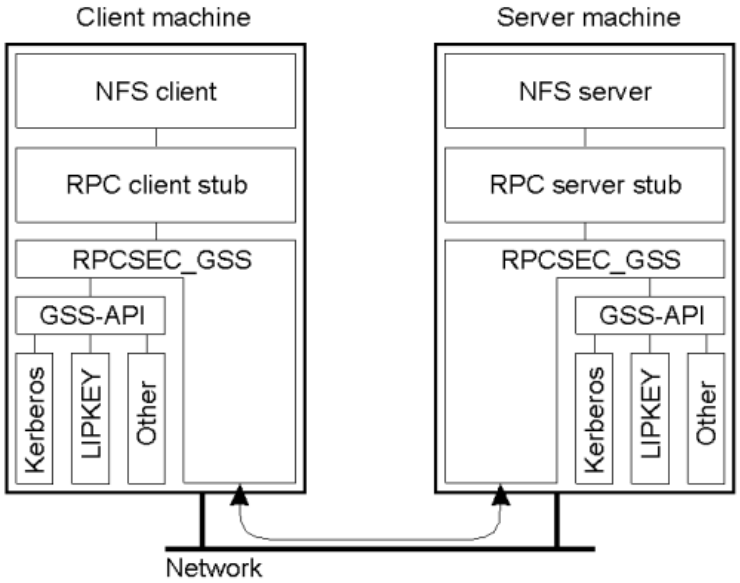

Figure 24.3: Secure RPCs

24.1.5 Replica Servers

There may be multiple servers serving different set of files. Version 4 allows the files to be replicated. Client can make request for accessing files from any of the replicas. NFS provides implementation of maintaining consistency between the replicated servers.

24.2 Coda Overview

24.2.1 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

24.2.2 Coda supplements file cache with user preferences

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

24.2.3 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.

24.2.4 File Identifiers

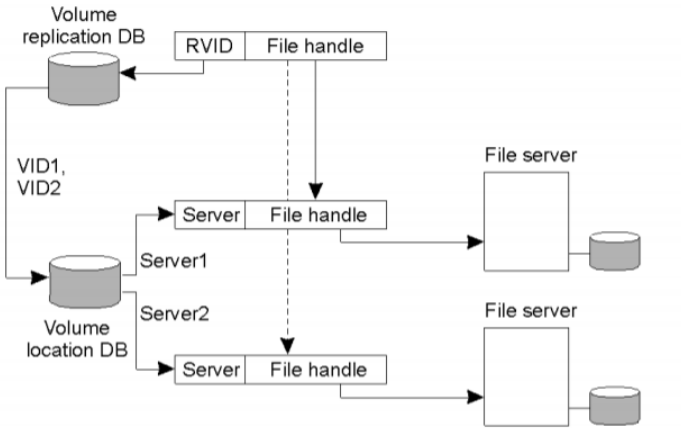

Figure 24.4: Coda architecture

- 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

24.2.5 Server Replication

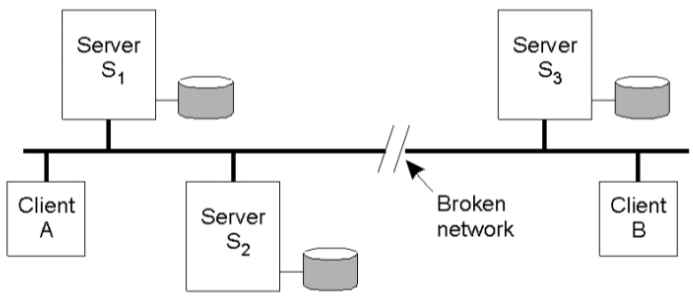

Figure 24.5: Server replication issues in Coda

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.

24.2.6 Disconnected Operation

Hoarding state means the client is connected to the server and is actively downloading files into cache based on some prediction based on current usage of the user. Upon disconnecting the client is in emulation state. Upon reconnecting to the server, the client is in reintegration state. The clients merge its updates with server's updates.

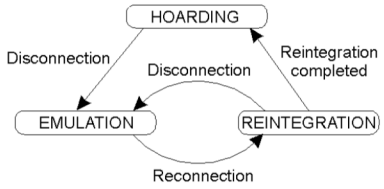

Figure 24.6: Disconnected operation in Coda

- 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

24.2.7 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

24.2.8 Client Caching

- Cache consistency maintained using callbacks

24.3 xFS

24.3.1 Overview of xFS

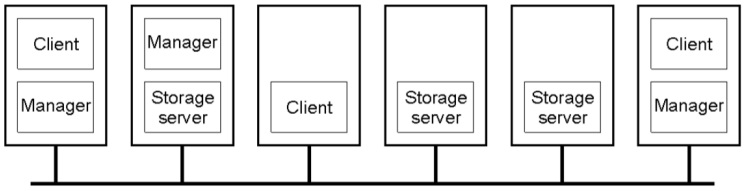

Figure 24.7: An example of nodes in xFS

- 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 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.

24.3.2 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.

24.3.3 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!

24.4 RAID

24.4.1 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

24.4.1.1 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.

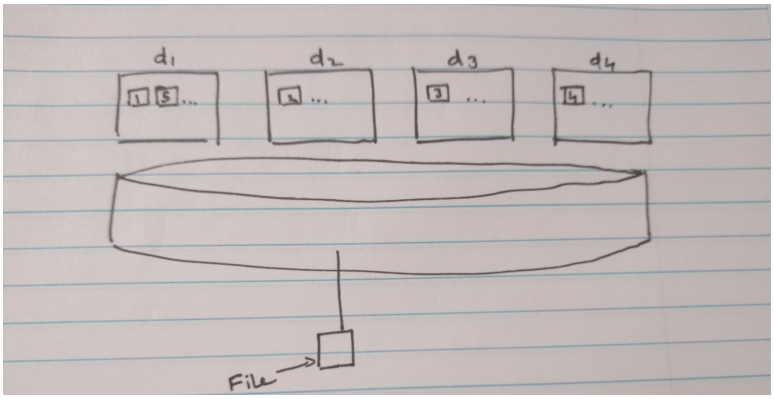

Figure 24.8: Striping in RAID

Figure 24.8 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:

24.4.1.2 RAID 1 (Mirroring)

From figure 24.9, 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.

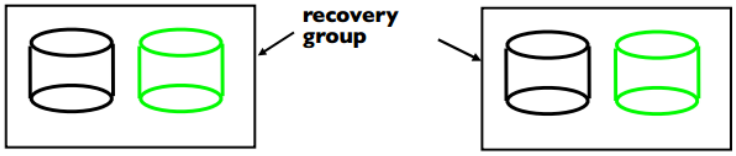

Figure 24.9: RAID 1

24.4.1.3 RAID 4

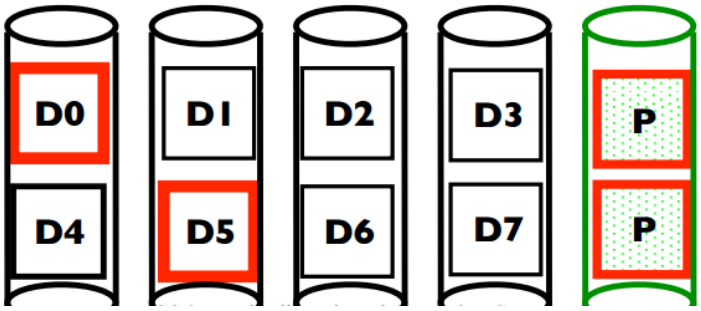

Figure 24.10: RAID 4

This method uses parity property to construct `ECC` (Error Correcting Codes) as shown in Figure 24.10. 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:

$$P = D_0 \oplus D_1 \oplus D_2 \oplus D_3 \tag{24.1}$$

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

$$D_0 = D_1 \oplus D_2 \oplus D_3 \oplus P \tag{24.2}$$

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.

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.

24.4.1.4 RAID 5

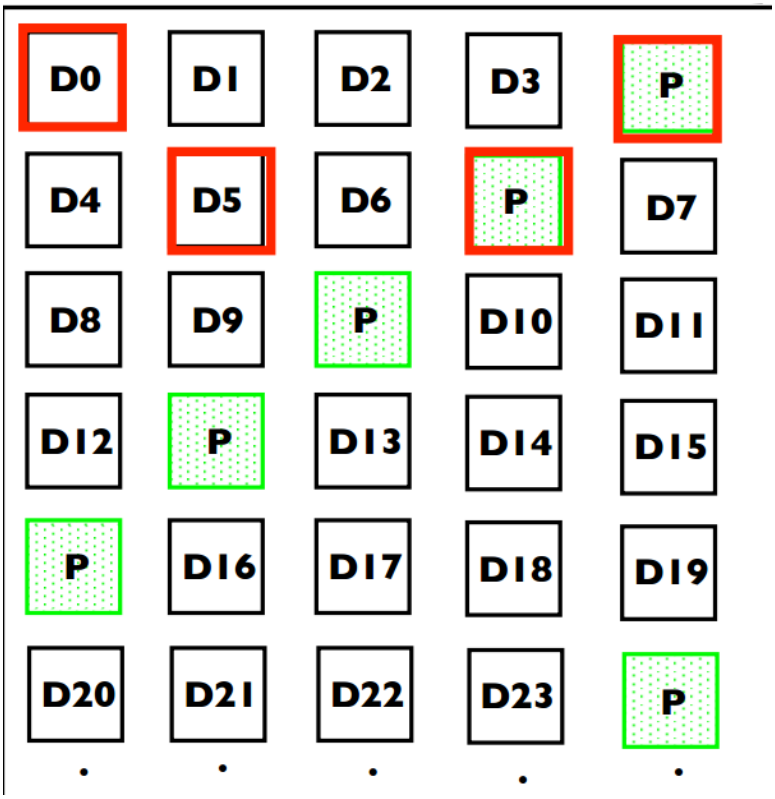

Figure 24.11: RAID 5

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 24.11. 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.

24.4.2 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

24.4.3 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

24.4.4 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

24.5 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.

24.5.1 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.

24.6 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.

24.7 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

24.8 Distributed Objects

In case of remote objects, code on a client machine wants to invoke an object's method on a server machine. A common way to achieve this, is as follows. Clients have a stub called proxy with an interface matching the remote object. An invocation of a proxy's method is passed across the network to the 'skeleton' on the server. That skeleton invokes the method on the remote object and returns the marshalled response. This can be recognized from earlier in the course as an RMI or RPC call.

Distributed objects are similar, but the distributed objects are themselves partitioned or replicated across different machines. Distributed objects use RPC. Middleware systems have been developed to support distributed objects.

24.9 Enterprise Java Beans

Enterprise Java is used to write multi-tier applications where the app server is actually written in Java. It also gives some additional functionality like the concept of a bean. A bean is a special type of an object. As a middleware, we will essentially have our objects written as a bean of some sort. It also provides other services like RMI, JNDI, JDBC(used to connect to Databases), JMS(Java Messaging Service). EJBs support more functionality that makes it easier to write web applications.

EJB are fundamentally object oriented, with two components, the interface and the implementation. The EJB class encodes the business logic of the application. EJB helps in persisting state of objects to the disk and retrieve when needed.

24.9.1 Four Types of EJBs

- **Stateless session beans** - They are essentially objects which do not have any state at all, they might just expose code.
- **Stateful session beans** - By default, the memory state is transient and if we kill the application, the object is gone. We can automatically persist the state of the object using these. Two important attributes: 1. Session 2. Session state is stored on the server side.
- **Entity beans** - They look more like standard Java objects. The object has state which is persisted on disk
- **Message-driven beans** - They are designed for messaging and the messages can persist.

24.10 CORBA : Common Object Request Broker Architecture

At the core of CORBA is the Object Request Broker (ORB) [also called messaging bus] is a intermediate communication channel that allows communication between objects.

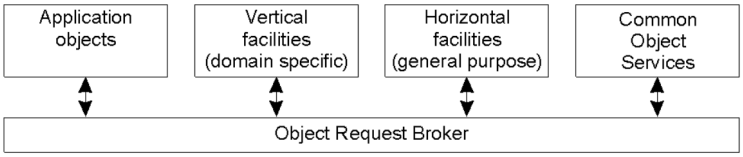

Figure 24.12: CORBA

The four boxes are the four components and they communicate using RPCs handled by the ORB. Many functionalities are already provided in CORBA as a service. They provide a dozen different services from concurrency to licensing in complex distributed systems. The advantage is that they can help reduce the code needed to develop complex distributed systems. However, in trying to provide every service you'd ever need, CORBA became very heavy weight. It does a lot of overhead to write even a small application. By becoming very heavy-weight, it became very difficult to learn and simple distributed applications would require deploying such a heavy weight system. Even though it did not become a commercial success, some stripped down versions of CORBA actually got used. Example - messaging service in Linux desktop manager called gnome. EJB are widely used.

The stub in the object model of CORBA is called ORB. It uses Interface Definition Language (IDL) to use an interface and compiler to generate code (like protobufs). Proxy is used to specify the objects and services. Object adapter provides portability between languages. Thus CORBA, is language independent. It also allows dynamic invocation of interfaces. At runtime, CORBA can fetch interfaces to put in the stub which can then be used. CORBA provided even more flexibility having the option of invoking RPCs as any type including synchronous, one-way, or deferred synchronous. CORBA was one of the first distributed middleware systems. Modern middleware systems take many ideas from it.

24.10.1 Event and Notification Services

This is done using an event channel. It allows us to implement an application using the publisher-subscriber model. Publishers post events to the event channel, and consumers/subscribers ask for events that they subscribe to from the event channel. Publisher subscriber works with any combination of push and pull. In

CORBA, it is a push-push model where data is pushed from publisher to event channel. The event channel will see the list of consumers subscribed and whenever there is a match, it will push to the consumer. Event channel can be thought of as a buffer. In pull-pull model, event channel polls data from the publisher and similarly, consumer pulls data from the event channel.

Two ways to implement the event channel: 1. Push based model. 2. Pull based model.

We can have hybrid models as well where we can have combination of push and pull models.

24.10.2 Messaging - Async method Invocation

To implement all kinds of RPCs, CORBA has callback model. It will have a callback interface where we can register a callback function. Whenever a reply to the async RPC comes back, we are notified and we can get our reply. We can also use the polling method where we keep polling periodically to see if the reply has come back.

24.10.3 Messaging - Polling based model

In this, the consumer keeps polling the event channel to check if there is any data that aligns with the subscription.

24.11 DCOM : Distributed Component Object Model

DCOM is Microsoft's middleware which has now evolved into .NET. DCOM will only run on Windows servers or desktops. COM is a simple RMI based framework running local to a machine that allowed communication within a machine. It is mainly used for communication between Microsoft applications. Object Linking and Embedding (OLE) was added to allow Microsoft office applications to communicate with one another via embedding and document linking. The ActiveX layer facilitates exposing these services as web applications by allowing us to embed things in web documents. Microsoft picked up this whole thing and made it distributed called DCOM. .NET has a language independent runtime, but ActiveX only works with Internet Explorer.

The architecture of DCOM is fundamentally the same as the distributed objects. The stub is essentially the COM layer. At its core, it is an RMI based system on objects. There is a type library which is similar to CORBA's interface library. We also have a SCM (Service Control Manager) which keeps track of what all objects are in the system and where are they running. DCOM is not as heavy as CORBA.

The objects in DCOM can also be made persistent even though they are transient by default. It is done using the notion of a Moniker. Moniker is the name of a persistent object that allows us to reconstruct that object after we shut down the server application.

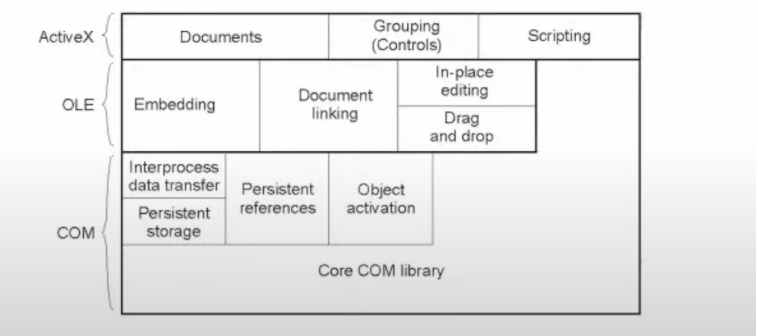

Figure 24.13: DCOM

24.12 Distributed Coordination Middleware

In this case, we have a very loose coupling between how communication works. The idea is that we want to separate our computations from coordination. Distributed applications can be classified based on whats happening in time and space dimension. Applications can either be coupled or decoupled in space and time.

1. ⟨ coupled in space and time ⟩ Direct
2. ⟨ coupled in space but not time ⟩ Mailbox - receiver is known but receiver state does not matter.
3. ⟨ coupled in time but not space ⟩ Meeting Oriented - unknown who will show up to meeting.
4. ⟨ decoupled in space and time ⟩ Generative Communication - components can communicate with another without knowing who might read it or when it would be read. It uses a pub-sub model.

Question: If the applications are coupled in time, how does the clock synchronization work?
Answer: Coupled in time does not necessitate that clocks have to be synchronized. It just means that both parties need to be active at the same time. It could be two people or two processes.

24.12.1 Jini Case Study

Jini is a Java based middleware that uses distributed coordination. It facilitates service discovery. We do not know what entities are present in the system so this notion of discovery allows us to discover what services are available and so on. These are also called zero-configuration services because we do not need to pre configure anything as services as discovered on the fly. It uses a event notification system that is pub-sub based. Jini uses a bulletin board architecture. Services advertise on the bulletin board and machines can access the services it requires through the board. It is decoupled in time and space.

In Jini, bulletin board is called JavaSpace or tuple space. JavaSpace is basically a database. Each tuple is a Java object. We have reads and writes into a shared database. One can request callbacks in this model as well.

Question : How is this different from a Publish-Subscribe Model?
Answer : In Pub-Sub, subscriptions are done beforehand. In addition, messages posted in the bulletin-board model can be stored for a long period of time.

Question : Where is this JavaSpace running?
Answer : JavaSpace is our middleware service which has to run on some server or some set of servers. So essentially, our middleware is running somewhere else and we have to read and write from that.

Jini utilizes a pub-sub architecture with both pull based as well as notification based discovery. Here the messages are persisted on disk unlike Event channel where messages don't persist.

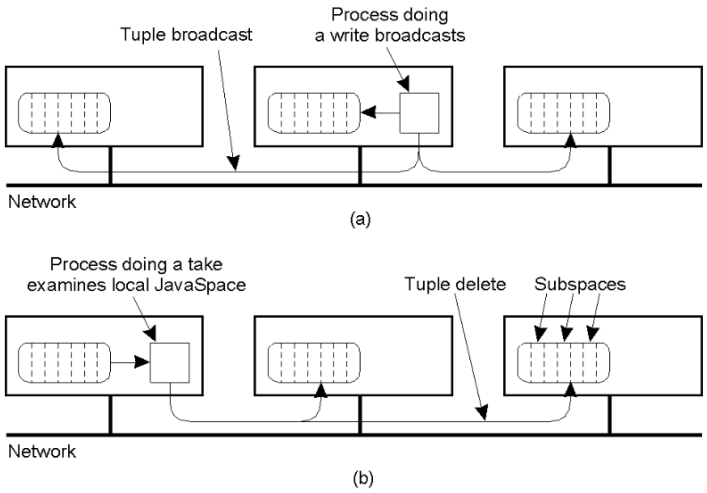

Figure 24.14: Jini Processes

Here, JavaSpace is distributed across multiple machines. The boxes are JavaSpaces with tuples in them. JavaSpaces can either be fully replicated or distributed. In case of replicated JavaSpaces, writes need to be broadcasted to all replicas whereas reads are local. On the other hand, in distributed bulletin board, each board has a subset of nodes, while writes are local and reads need to be done on each and every board.

Question : Whatever we are posting in the JavaSpace, is it a Java object and how is it posted (copied/sent)?
Answer : Tuples are not Java objects, they are data objects like a key and a value. So essentially, we are publishing data or events instead of objects which is different from sending/ receiving objects.

Question: Why can there be multiple copies of data on the bulletin board?
Answer: That was just shown as a logical view. In reality, if the board is replicated, we write it to all the boards. The logical view just shows a union of all replicas

Question: Since messages are not intended for any one process, how does this process(in the example) who read C know that it has to delete C from the database?
Answer: It depends on the number of recipients of a message. If its just one recipient, then the first time a recipient comes and looks for C, we can delete C. If many recipient, then we wont remove it.

24.13 Distributed Middleware Systems

These are middleware systems that are designed for large scale data processing.

24.13.1 Big Data Applications

This is mostly covered in CS532 systems for data science. So we will only partially cover this. We have to parallelized data processing using distributed systems as its a large amount of data.

Distributed data processing is different types of middleware that are designed for processing large amounts of data. The basic idea is that we will use multiple machines of a cluster and parallelize our application for data processing. Each machine will read and process some part of the data.

24.13.2 MapReduce Programming Model

MapReduce is a framework for processing parallelizable problems across large datasets using a large number of computers (nodes), collectively referred to as a cluster (if all nodes are on the same local network and use similar hardware) or a grid (if the nodes are shared across geographically and administratively distributed systems, and use more heterogenous hardware). Processing can occur on data stored either in a filesystem (unstructured) or in a database (structured). MapReduce can take advantage of the locality of data, processing it near the place it is stored in order to minimize communication overhead.

MapReduce is a two-stage process to process a large dataset - Map phase and Reduce phase.

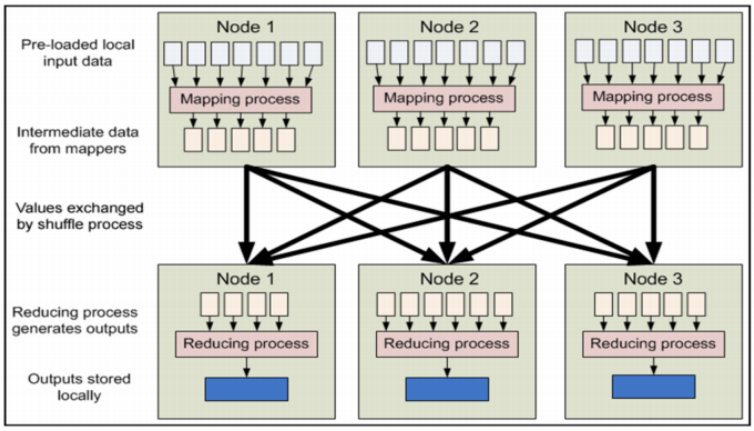

Figure 24.15: Map Reduce example

Map Step : Each worker node applies the "map()" function to the local data, and writes the output to a temporary storage. A master node ensures that only one copy of redundant input data is processed.

Shuffle Step : Worker nodes redistribute data based on the output keys (produced by the "map()" function), such that all data belonging to one key is located on the same worker node.

Reduce Step : Worker nodes now process each group of output data, per key, in parallel.

Example1: Let's say we want to do word counting on a large set of documents. A chunk of data is given to each machine which counts the words within that chunk (map phase). Now we need to sum these up. This is done in reduce phase - 1 node is assigned to each word. It receives count from machines for that word and sums them up.

Example2: Let's say we have huge dataset of number (or may be words in a document) which need to be sorted (or frequency of words). Suppose we have multiple machines. Then each machine could take a chunk

of data and sort it. Later, all these sorted chunks chould be merge together similar to merge sort algorithm. However, this requires huge communication between the machines during merge phase. To overcome this problem, let's say we know the range of numbers. In such a case, we could follow the bucket sort paradigm where each machine sorts a specific range of numbers. This way, inter-machine communication can be reduced.

The datasets being processed here are actually stored on HDFS. Each node can read its local data from HDFS or it can also read remote data. Reading locally has lesser overheads and is cheaper.

Question: What is the reason to shuffle? Why cant you serialize it?
Answer: That is what we are doing. In the second phase, we have to decide who is responsible for which set of data and send it to that node. Because all the data incoming to the first phase (Node 1) in the case of sorting a data is random. So from second phase we decide what set of data will be handled by which set of nodes and forward it.

24.13.3 Hadoop Big Data Platform

Hadoop is an implementation of Map-Reduce framework. It is an open-source software framework for storing data and running applications on clusters of commodity hardware. It provides massive storage for any kind of data, enormous processing power and the ability to handle virtually limitless concurrent tasks or jobs. It has :

- **store managers** : where the datasets are stored. Eg: HDFS, HBASE, Kafka, etc (replication for fault tolerence.
- **processing framework** : Map-reduce, Spark, etc
- **resource managers** : allocates nodes and resources to jobs. Some of the concepts of distributed scheduling are also adopted since they need to serve multiple users. Example : Yarm, Mesos, etc

Question : How is HBASE using MapReduce?
Answer : HBASE is just a storage layer. MapReduce reads from the storage layer, in this case - HBASE

24.13.3.1 Ecosystem

Based on the requirements different types of frameworks can be used. For example, if user wants to process the data that have lot of graphs then Graph processing framework Giraph can be used. There are machine learning frameworks like MLLib, Oyyx, Tensorflow that are also designed to run to on Hadoop. If a user wants to input data to these distributed processing framework, he could use applications like hive to easily write map-reduce codes. For real time data processing where data is generated continuously by some external source, framework like Spark Storm etc could be used.
We can see that it is not a single distributed application, it is a set of applications that work together.

24.13.4 Spark Platform

Apache Spark is a powerful open source processing engine built around speed, ease of use, and sophisticated analytics. It was originally developed at UC Berkeley in 2009. Spark was an important innovation over MapReduce. Although MapReduce uses parallelism, it is very heavy on I/O that can slow down the application. In Spark, we store intermediate data in memory of some server. What we decide to store and how to store is something we have to think about when writing a Spark application.

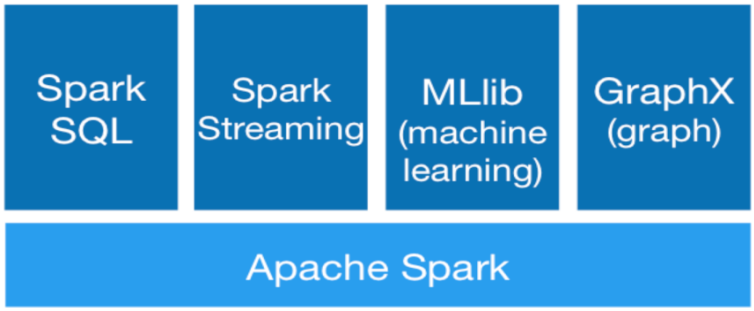

Figure 24.16: Spark Platform

Spark SQL : Spark SQL is a Spark module for structured data processing. It provides a programming abstraction called DataFrames and can also act as distributed SQL query engine. It enables unmodified Hadoop Hive queries to run up to 100x faster on existing deployments and data. It also provides powerful integration with the rest of the Spark ecosystem (e.g., integrating SQL query processing with machine learning).

Spark Streaming Many applications need the ability to process and analyze not only batch data, but also streams of new data in real-time. Running on top of Spark, Spark Streaming enables powerful interactive and analytical applications across both streaming and historical data, while inheriting Spark's ease of use and fault tolerance characteristics. It readily integrates with a wide variety of popular data sources, including HDFS, Flume, Kafka, and Twitter.

MLlib Built on top of Spark, MLlib is a scalable machine learning library that delivers both high-quality algorithms (e.g., multiple iterations to increase accuracy) and blazing speed (up to 100x faster than MapReduce). The library is usable in Java, Scala, and Python as part of Spark applications, so that you can include it in complete workflows.

GraphX GraphX is a graph computation engine built on top of Spark that enables users to interactively build, transform and reason about graph structured data at scale. It comes complete with a library of common algorithms.

Adavatages of Spark

- **Speed :** Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.Apache Spark has an advanced DAG execution engine that supports acyclic data flow and in-memory computing.
- **Ease of Use:** Spark offers over 80 high-level operators that make it easy to build parallel apps. And you can use it interactively from the Scala, Python and R shells.
- **Generality :** Combine SQL, streaming, and complex analytics.Spark powers a stack of libraries including SQL and DataFrames, MLlib for machine learning, GraphX, and Spark Streaming. You can combine these libraries seamlessly in the same application.
- **Runs Everywhere:** Spark runs on Hadoop, Mesos, standalone, or in the cloud. It can access diverse data sources including HDFS, Cassandra, HBase, and S3.

24.13.4.1 Spark Architecture

Distributed memory is just like memory but we access data across various machines. All of the memories of the servers can be accessed if we store data in the form of an RDD (Resilient Distributed Dataset). The idea is that we will first read data from disk, say HDFS. Then we will do some partial processing, then transformed dataset will be stored in RDD and so on. Since data is in memory, processing will be much faster. If the data is larger than the memory available, data can spill over to disk.

RDD is designed to support in-memory data storage, distributed across a cluster in a manner that is demonstrably both fault-tolerant and efficient. Fault-tolerance is achieved, in part, by tracking the lineage of transformations applied to coarse-grained sets of data. Efficiency is achieved through parallelization of processing across multiple nodes in the cluster, and minimization of data replication between those nodes. Server failures can be handled by recomputation.

Question : Is Spark a type of distributed middleware?
Answer : Both Hadoop and Spark are type of distributed middleware designed for large data processing.

Question : In Spark where are the computations stored?
Answer : The computations are code that a user has written. Spark will keep track of graph as its generated. For each node it will keep track of what code was run to generate this output. We will only redo that part to do the minimum computation to generate the data.