CMPSCI 677 Distributed Operating Systems **Spring 2025**

Lecture 22: April 23

Lecturer: Prashant Shenoy *Scribe: Srikrushna Pinaki Budi (2025), Shrutiya Mohan (2024)*

22.1 File System Basics

22.1.1 File

A file is a container of data in text format, binary format etc. which is stored on a disk so that the user can re-visit it at a later point in time. In UNIX, a file is an uninterpreted sequence of bytes which implies that the file system is unaware of the contents/type of the file. Other operating systems like Windows and Mac knows the file types (This information can be useful to open a file in the right application).

22.1.2 File System

- File system abstracts and provides a logical view of data (a hierarchy of files and folders) and storage functions.
- It helps us to create, modify, organize and delete files and takes care of how to map them to the underlying storage device.
- It provides a user-friendly interface so that the user need not deal with the low-level interfaces exported by the disk.
- It allows us to share the files among other users by giving permissions and also allows us to protect the files.

22.1.3 UNIX File System Review

- In UNIX, the files structure can be viewed as a directed acyclic graph. Note that this looks like a tree structure but can contain soft links pointing from one directory to other which makes it a DAG. Each directory entry for each file contains the file name, inode number (metadata for the file), major device number and minor device number. All inodes are stored at a particular location on the disk called super block. To access the file, the file system needs to first get this metadata to know where the file is located in the actual disk (aka block locations of the file).
- An inode structure consists of the fields like mode, Owner ID, group id, Dir file, protection bits, last access time, size, reference count, address[0]...address[14] etc. The addresses stores the pointers to the data blocks. The first 12 are the direct blocks which stores the pointers to the data blocks (see figure 22.1), the 13th address stores the pointers to the location which in turn stores the pointers to the direct data blocks (one level of indirection). The 14th address follows two levels of indirection which stores the pointers to one level indirection blocks. So the hierarchy grows as the size of the file grows but we have an upper limit of the size of the file that can be stored on this file system because we only have a certain number of pointers in addresses (In this case from 0 till 14).

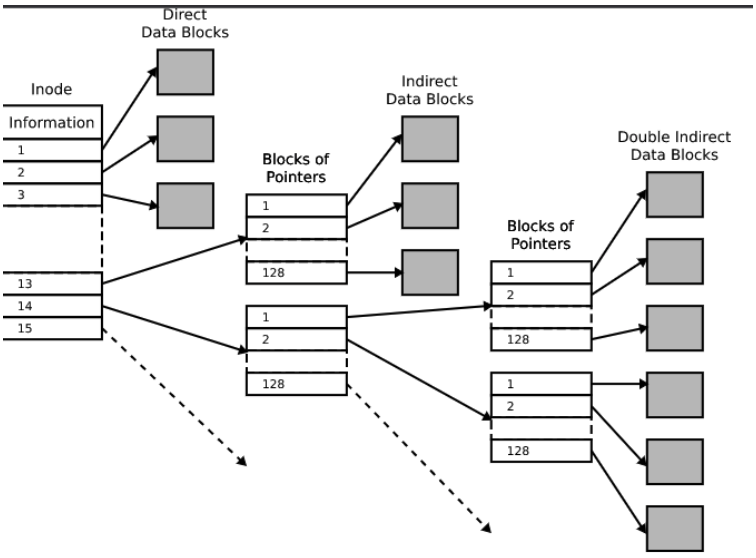

Figure 22.1: Inode structure

22.2 Distributed File Systems (DFS)

If files on a different machine can be accessed, it is a distributed file system. Another way to think about DFS is that the servers store different files on different servers, and all the servers collectively form your file system.

22.2.1 File server

A machine that stores all the files.

22.2.2 File service

The interface that the machine exposes for other machines to access the files on this machine. For example NFS uses RPCs to send read/write requests to a remote file system. There are two types of file services as shown in figure 22.2.

- Remote access model: The client requests are sent to the server and the server sends back the results after doing the work requested by the client. This model is typically stateful since we need to keep track of which clients are accessing which file and so on. This might eventually cause the server to become a bottleneck if there are many incoming requests from multiple clients (I/O bottleneck at the server).
- Upload/download model: When the client performs a request to the server, entire file is sent as a copy to the client, and subsequent access are made to the local copy. To maintain consistency, the client eventually sends back the changed file to the server. This model works only if there is one client one file at a time, hence maintaining consistency.

Note: As the files are directly updated on the server, there is consistency in the remote access model, but each operation is an RPC call which makes it slow. In upload/download model, there is a period of time in which the file on the server is out of date. Having said that, upload/download model gives better performance as the operations are taking place on the local machine and very less calls are made to the remote machine.

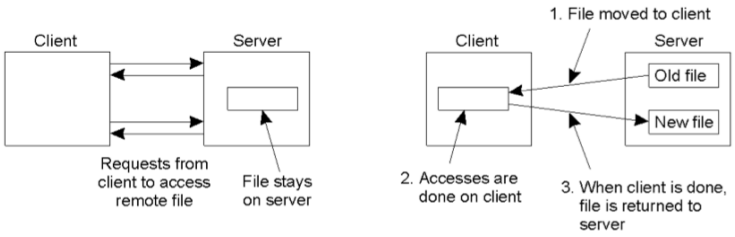

Figure 22.2: Remote access model(left), Upload/download model(right).

Question: Which model does Google Docs use?
Answer: Google docs is a cloud service and not a distributed file system in a technical sense. Today, Google, One drive, Dropbox provide a form of remote storage that looks like distributed file system, but is not necessarily the same. Answering the question, google docs model depends on the mode of the browser. In the online version, every change is saved on the cloud server instantly, while in the offline mode, there is a copy at the client and a master copy at the server.

22.2.3 Server Type

There are two types of server and one would need to make the choice from one of them when building any distributed file system.

Stateless server: No information about clients is kept at the server.

Stateful server: Server maintains information about the client accesses. It is less tolerant to failures because the state is lost when a server crashes. There is slight performance benefit here due to the compact request messages (Clients do not need to send the information like permissions every time the request is being made). Consistency and idempotency are easier to achieve.

Note: An Idempotent server executes as if it has performed the request only once regardless of how many times same request was received.

22.2.4 Network File System (NFS)

NFS is a layer on top of an existing file system that allows to share the file system over a network. NFS is implemented using virtual file system layer supported by the underlying operating system. Virtual file system layer can be seen as a forwarding layer that looks at where is the file stored and invoke that file system(local file system for local files and possibly NFS for remote files). Here, the client and server communicate with each otehr using RPC calls. The VFS layer in the client and server provides a system independent abstraction to the layer above it. Thus, no need to worry about the type of the filesystem the file is stored on.

Note: Till version 3, NFS used stateless server protocol but from version 4, it uses stateful server protocol. So it now supports open call to a remote file.

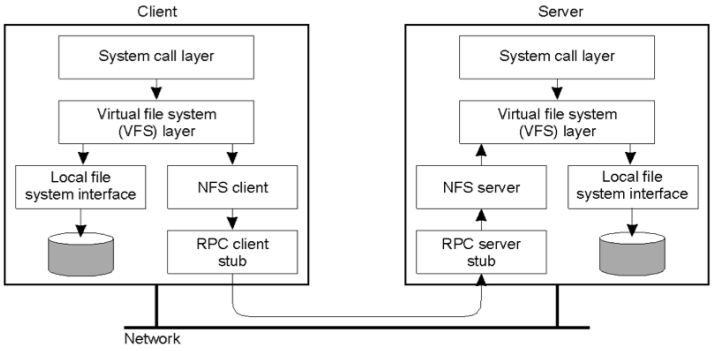

Figure 22.3: Network File System (NFS)

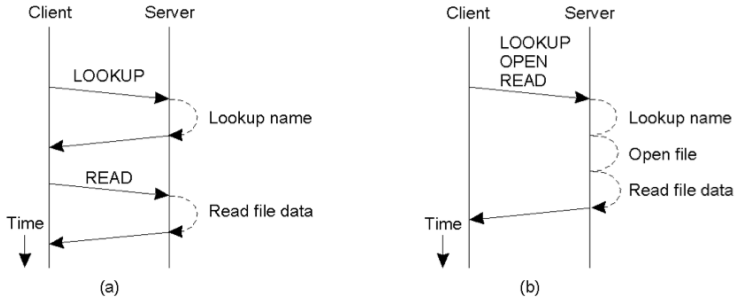

Figure 22.4: Difference in communication in NFS version 3 and version4.

In figure 22.4, we can observe that in version 3 of NFS, individual LOOKUP and READ RPC calls were needed whereas in version 4 of NFS, we can perform a batch RPC request. Version 3 executes one RPC per operation, whereas version 4 supports multiple RPC calls per operation(batched).

Question: Is the NFS Client and Server implemented as a user space process or is it implemented inside the kernel?

Answer: It is an in kernel implementation. It is not a user space process. In most operating systems, NFS code is going to reside inside the kernel like any other file system code but this is just distributed in nature.

Question: Why is Lookup triggered?

Answer: We usually access a file by opening the file and reading it. This eventually will go to the OS as a system call that's called an open system call at the OS level. This will go to a virtual file system layer and then the NFS Client and client has to service the open but version3 does not have a concept of an open so instead it is going to send a lookup operation to the server saying client wants to access this file. Then checks if file name is valid and if client has privileges to access it. It sends a response in a yes or no (yes if open call succeeds) then we get another read system call and another RPC is triggered.

22.2.5 Mount protocol

Mount protocol is a way how a NFS client gets access to a remote file system. Certain directories can be mapped from remote file system to the local file system in order to get access.

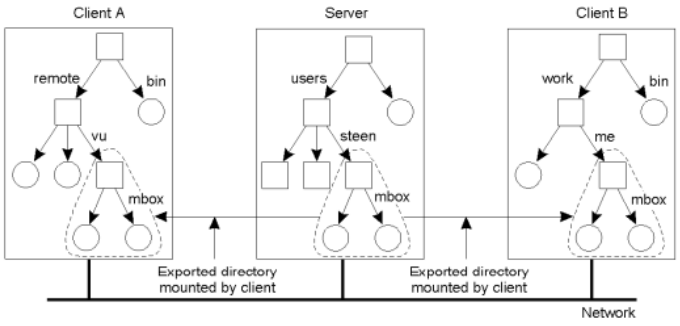

Figure 22.5: Mount Protocol

Question: Client A can access few files in the mounted directory and Client B can access few files in the mounted directory, can you see all of those files?

Answer: The visibility of these files is controlled by file permissions. Similar to Unix we set the file permissions which mention read write and execution permissions to other users. OS will control this and file sharing can be done in distributed systems in the same way.

Question: Does mounting create a list of files stored on the server or are you always going to lookup files on the server?

Answer: Mounting is simply a mapping. It is not going to create any list. Mounting creates a mapping from the directory searched to the directory's location on the server. Client OS is not going to know what is stored in the directory. So when the user tries to access anything inside the directory, all the operations are sent to the server and whatever responses are returned are checked.

Question: Is concurrent access possible in this model?

Answer: There are two types of concurrent accesses: accessing same file, accessing same volume of files. It is of course possible, example Ed Lab with multiple users accessing the file volume on the system and working on it concurrently. To access the same file, it is possible but we need locking mechanisms to avoid overwriting each other's data.

22.2.6 Crossing mount points

Crossing mount points is mounting nested directories from multiple servers. NFS v3 does not support transitive exports NFS v4 allows clients to detect crossing of mount points and supports recursive lookups.

Question: What happens if the Client tries to access inner nested file which is local to Server B?
Answer: In NFS version 3 it does not allow transitive exports for security reason. It will return an error. But in version 4 first RPC request will go to Server A which will then forward it to Server B and perform the operation and send back the response.

22.2.7 Automounting

Automounting is also known as mounting on demand. The mappings get established but the mounting only happens when the user tries to access those directories. And if there is an idle time, it unmounts. This way we can reduce the amount of kernel resources used.

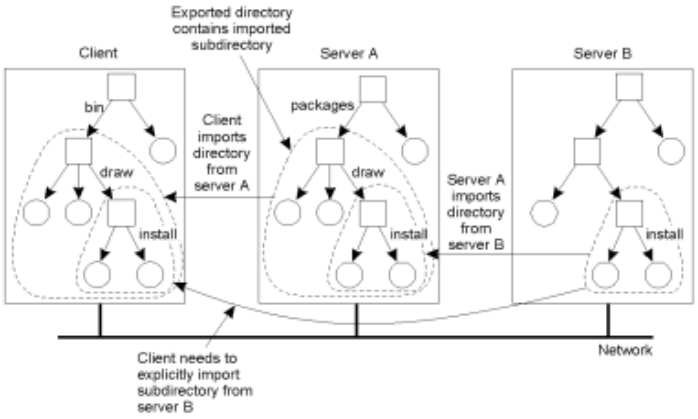

Figure 22.6: Crossing Mount Points

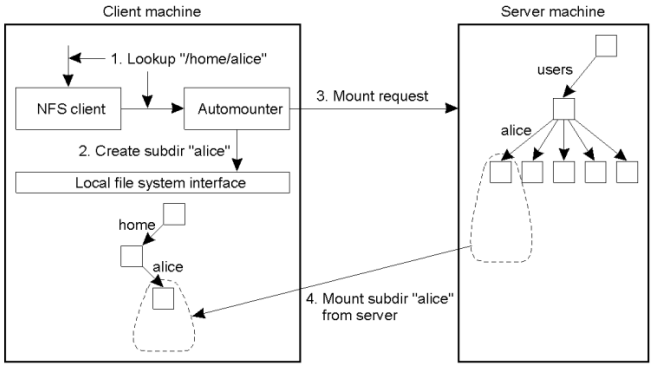

Figure 22.7: Automounting

22.2.8 File attributes

There are specific attributes (like TYPE, SIZE, CHANGE, FSID) that a file system must to support to be compatible with NFS. There are other attributes (like OWNER of a file) which are not mandatory to be compatible with NFS but are recommended.

22.2.9 Semantics of file sharing

- In UNIX semantics, every operation on a file is instantly visible to all the other processes using the same file.
- In session semantics, no changes are visible to other processes until the file is closed.
- Immutable files cannot be mutated. A new version of the file needs to be created if we need any changes.
- In transaction semantics, all changes occur atomically.

Note: NFS follows semantics in between UNIX and session. It caches the file and periodically flushes the

changes to the server. If one process writes to a file, the other process might have a different or outdated version of the file for a period of time. NFS uses local caches for performance reasons which leads to this weak consistency.

22.2.10 File locking in NFS

- Version 3 of NFS used stateless server protocol. One of the uses of having a state is file locking. Version 4 of NFS uses stateful server protocol, so applications can now use locks to ensure consistency. File locking can be done in different ways like locking the entire file, locking a specific range of bytes in a file etc.
- In share reservations file locking, we have the notion of a denial state where if an application has a write denial state to a file, it cannot write to the file but it can read the file.

22.2.11 Client Caching: Delegation

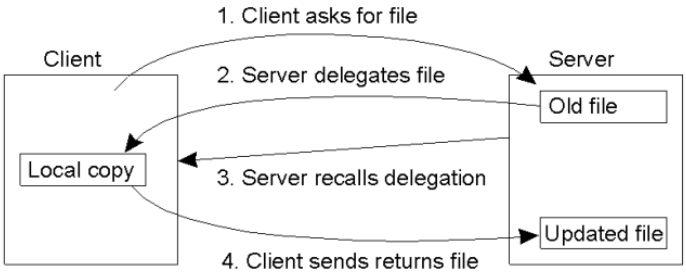

Figure 22.8: Delegation

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.

Question: The client is updating the file and the server recalls the delegation, what happens to the update?
Answer: It is fine for the client to perform all outstanding operations, have them finish and then send the file back. There is no need to cancel any write operations, it can wait and then send the latest version and

all subsequent accesses of the file back to the server, as we cannot have any local copy on the client. The open call will just take longer because the file has to come back to the server.

Question: Does it have any security concern? Can the file have make any unsanctioned updates?
Answer: The unsanctioned updates are not an issue because if the process has read and write access to the file, it can write whatever it wants. Access Control is the responsibility of the OS.

22.2.12 RPC Failures

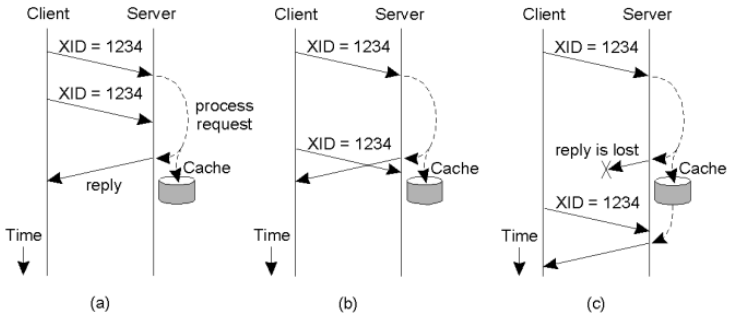

Figure 22.9: RPC Failures

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.

Question: Who is responsible for keeping the IDs unique, client or the server?
Answer: The IDs have to be generated at the client. This can be made unique by using the client's IP address followed by a number that is incremented sequentially for each RPC call.

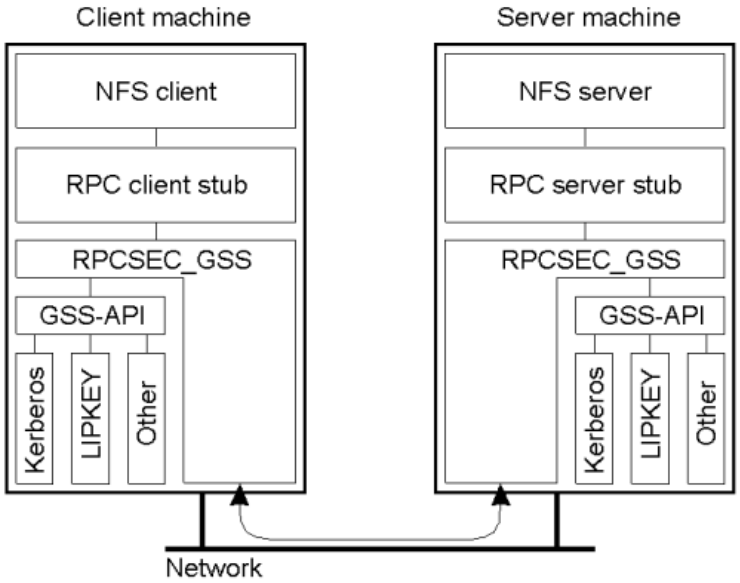

Figure 22.10: Secure RPCs

22.2.13 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

22.2.14 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.