CMPSCI 677 Distributed & Operating Systems **Spring 2017**

Lecture 21: April 17

Lecturer: Prashant Shenoy *Scribe:* ***Anirudha Desai***

21.1 Distributed File Systems

The class was primarily about stand-alone(UNIX) file systems.

21.1.1 File System (FS) Basics

There are many file systems. Linux can read many file systems. But windows on the other hand has its own proprietary file systems and cannot read linux file system. A FS manifests as files and directories.

A file is a named collection of logically related data. OS has no way to interpret the files. It is the application's job to interpret the file. A file system :

1. Provides a logical view of data and storage functions
2. User-friendly interface
3. Provides facility to create , modify , organize, and delete files.
4. Provides sharing among users in a controlled manner - Permissions can be granted on files to specific users.
5. Provides protection

21.1.2 UNIX File System review

A user file is a linear array of bytes. A file structure is a directed acyclic graph as shown in fig 21.1.

But the operating system does not interpret English. It only understands numbers. Every directory/file has an associated number in its metadata. This number is the *inode* number. The OS can understand this inode number. All inodes are stored at a special location on disk [super block]. Within a file structure, directories may not be shared, but files may be. Shared in this context means links. 2 files can be linked to the same inode number. There can be a *hard link* or a *soft link* (shortcuts to a file). But directories cannot have links because we can end up with a cycle when a child directory has a hard link back to its parent directory.

Inodes store file attributes and a multi-level index that has a list of disk block locations for the file. As shown in figure 21.2, there are multiple levels of indirection used to point to different blocks of a file. In this multi-level index format, we can technically store $11 * 1024 * 1024 \approx 4gb$
file. It should be noted that this is one way of implementing a FS. There are many other ways.

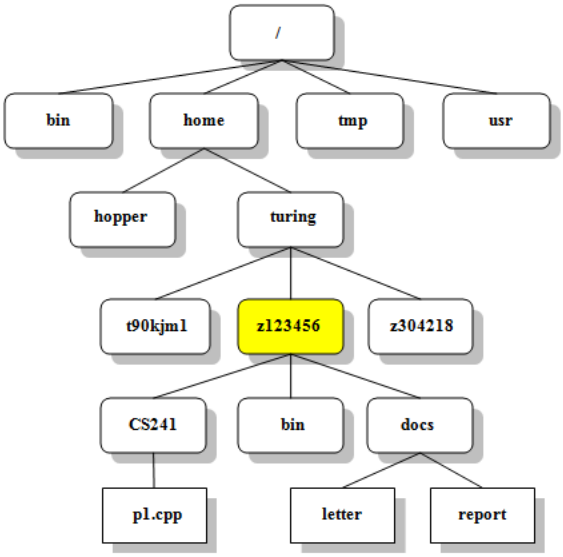

Figure 21.1: Typical file structure

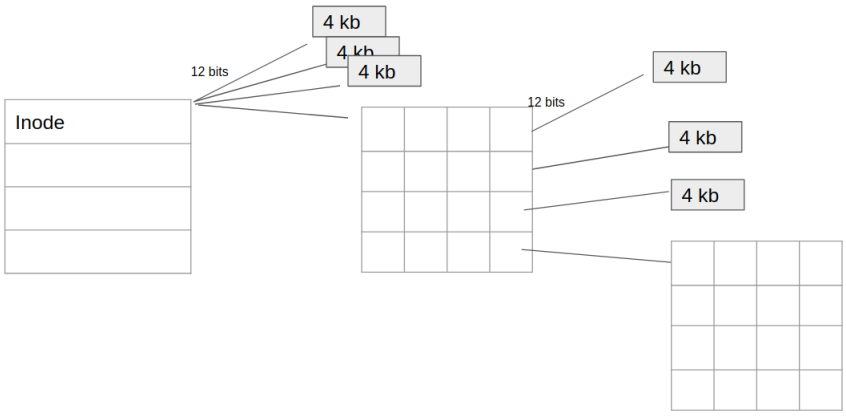

Figure 21.2: inode multi level index

21.1.3 Inode structure

The following fields comprise the inode number : mode, owne_id, dir_file, protection bits, last access time, last write time, last inode time, size, ref_cnt.

21.1.4 Distributed File Systems (DFS)

So far, the class was about single FS. A DFS is more complicated.

21.1.4.1 File Service

2 types : The slides could be referred for diagrammatic representation.

1. Remote access model : Work done at server. It is a stateful server. Server may be a bottleneck.

   The client needs to send RPC to server for every action. The file is stored on the server. There is excessive communication happening in this model. The server needs to maintain the state to know the status of communication with a client.

2. Upload/download model : Work done at client. It is a stateless server. Simple functionality. Moves files/blocks, need storage.

   The server moves the files from local storage to client storage. The client performs the processes and pushes the file back to server. This makes the server stateless. This is very slow in the beginning in order to load the file.

21.1.5 NFS Architecture

The NFS architecture is shown in fig 21.3.

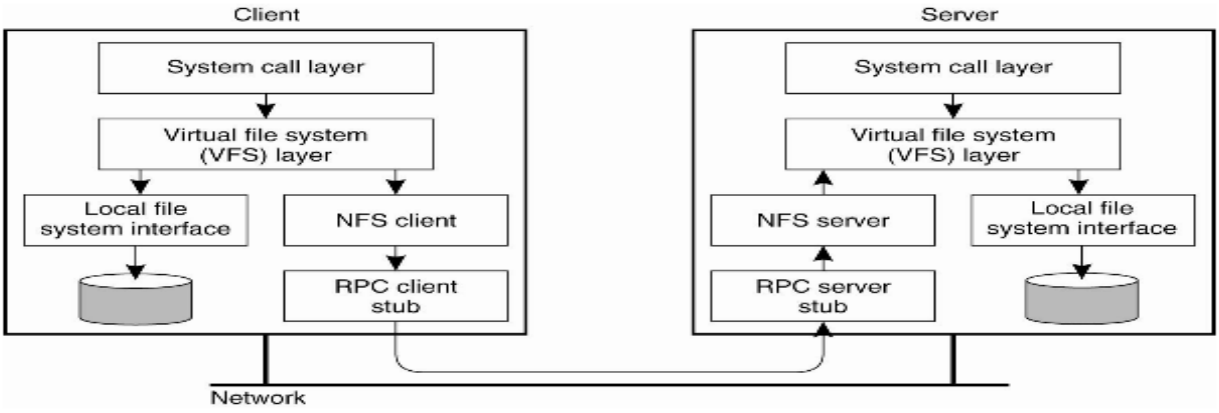

Figure 21.3: NFS Architecture

Either the data is on local FS or the data is on another machine. Then we go through the NFS. The NFS, as can be seen from the figure, is essentially an add-on to the OS.

21.1.5.1 NFS Operations

The basic NFS operations can be referred on the slides. The highlights here are that the v3 NFS was stateless where v4 is stateful. *Open* and *Close* were not present in v3 because it was stateless. These operations are present in v4.

21.1.5.2 Communication

The communication between client and server between v3 and v4 of NFS is represented in fig 21.4

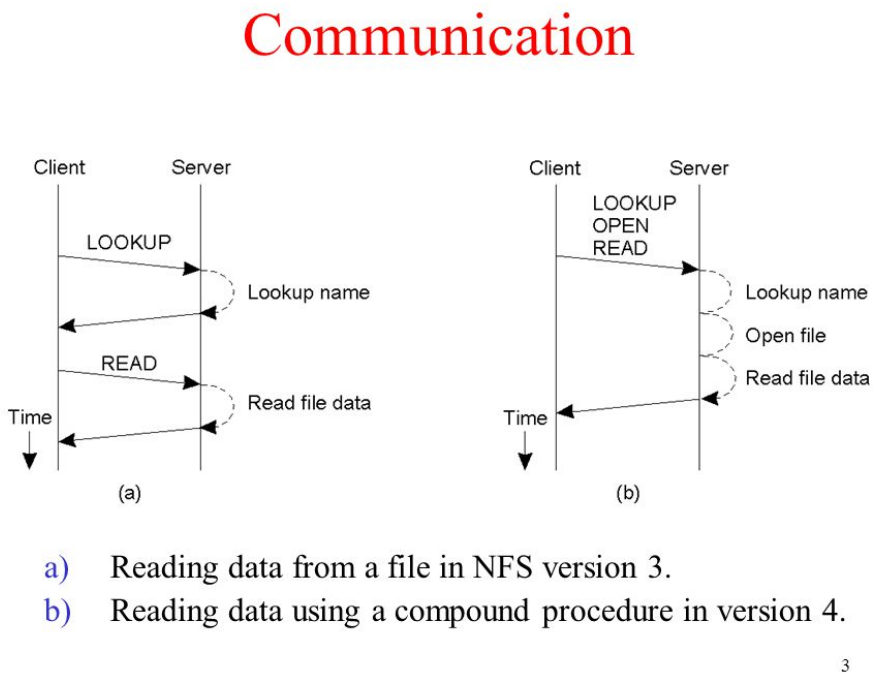

Figure 21.4: communication in NFS

In v3, the client takes the open call and does a *lookup*. If the file exists, server returns the lookup name. The client then sends a *read* call.
In v4, a bunch of calls can be sent like lookup, open and read. The server performs all the actions and returns to client. The v4 is faster because of lesser *round trip time*.

21.1.5.3 Naming: Mount Protocol

Mounting tells the OS about where to put the external storage like USB etc. The OS takes care of the mapping. Users can access remote files using local names.

21.1.5.4 Automounting

In a multiuser setting, 2 different users can share data together. In v3, mounting mounts everything the NFS has access to. But, with automounting, the user can mount exactly what is needed. It is basically, mount on demand.

21.1.5.5 File Attributes

Attribute	Description
TYPE	The type of the file (regular, directory, symbolic link)
SIZE	The length of the file in bytes
CHANGE	Indicator for a client to see if and/or when the file has changed
FSID	Server-unique identifier of the file's file system

Figure 21.5: Mandatory attributes for `NFS`

The attributes in fig 21.5 is mandatory for every FS. With v4, there were additional attributes that are recommended which are listed in fig 21.6

Attribute	Description
ACL	an access control list associated with the file
FILEHANDLE	The server-provided file handle of this file
FILEID	A file-system unique identifier for this file
FS_LOCATIONS	Locations in the network where this file system may be found
OWNER	The character-string name of the file's owner
TIME_ACCESS	Time when the file data were last accessed
TIME_MODIFY	Time when the file data were last modified
TIME_CREATE	Time when the file was created

Figure 21.6: Recommended attributes for `NFS`

21.1.5.6 Semantics of File Sharing

If a file is shared among users concurrently, it is necessary to define the semantics of reading and writing. The semantics which enforces an absolute ordering on file reads and writes is known as **UNIX semantics**. This semantics is generally desirable as it avoids inconsistency issues across different users. However, for a distributed system, Unix semantics can only be achieved if there is one file server and clients do not cache

Method	Comment
UNIX semantics	Every operation on a file is instantly visible to all processes
Session semantics	No changes are visible to other processes until the file is closed
Immutable files	No updates are possible; simplifies sharing and replication
Transaction	All changes occur atomically

Figure 21.7: Different File sharing semantics

files. A no cache policy, for a distributed file system, can cause serious performance issues. Hence weaker semantics are adopted. By allowing clients to update in local caches might create inconsistent versions of files across users. For example, if a client locally modifies a cached file and shortly thereafter another client reads the file from the server, the second client will get an obsolete file. To avoid these issues, a weaker semantics known as **Session semantics** is adopted. In this semantics, only when the file is closed the changes are made visible to other clients. A description of different file sharing semantics is shown in Figure 21.7.

21.1.5.7 File Locking in NFS

Operation	Description
Lock	Create a lock for a range of bytes
Lockt	Test whether a conflicting lock has been granted
Locku	Remove a lock from a range of bytes
Renew	Renew the lease on a specified lock

Figure 21.8: `NFS`v4 operations related to file locking

`NFS` allows clients to use locking at a file system level. Locking can also be enabled for a a particular section of a file. Typically applications for databases and emails use file system level locking. Locking was not part of `NFS` until version 3. `NFS` v4 supports locking as part of the protocol. A description of several file locking operations of `NFS` v4 is shown in Figure 21.8.

21.1.5.8 Client Caching

`NFS` does not enforce any caching policy to clients. Figure 21.9 depicts the client side implementation of caching in `NFS`. The client talks to memory cache and then to disk cache to access a file. If the file is not present in the caches, then the request is forwarded to the `NFS` server over RPC call. `NFS` by default implements remote access file service model. `NFS` v4 extends this architecture to accommodate upload/download file service model as well. If a single client is connected to the file server, the server delegates the master file to

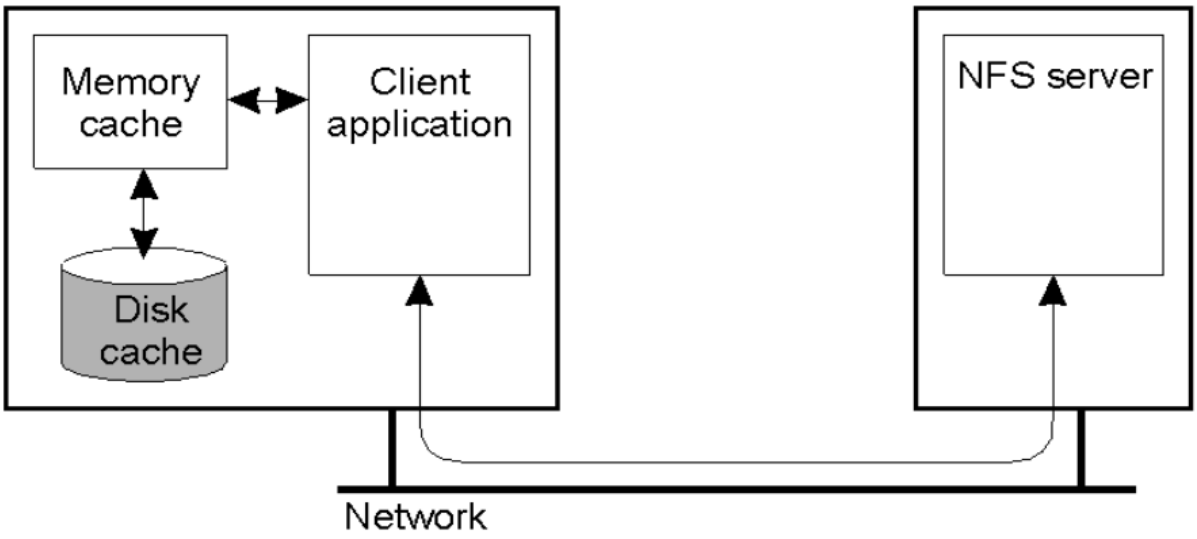

Figure 21.9: Client side caching in `NFS`

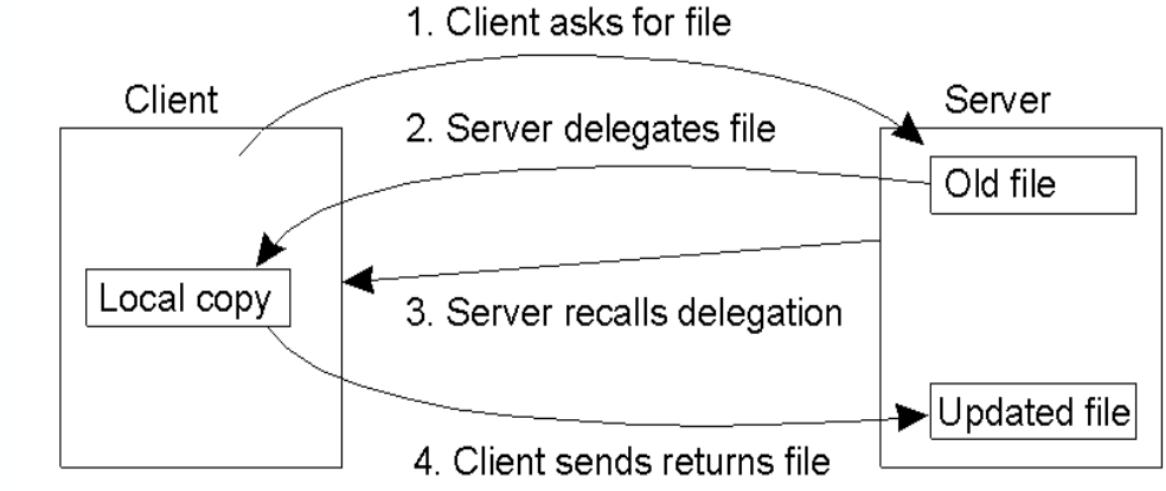

Figure 21.10: File delegation in `NFS` v4

the client. All read/write operation occurs at the delegated copy. This is known as **File Delegation**. When the file is closed or when the server recalls the delegation, the delegated file is sent back to the server. The overall file delegation procedure is depicted in Figure 21.10.

21.1.5.9 RPC Failures

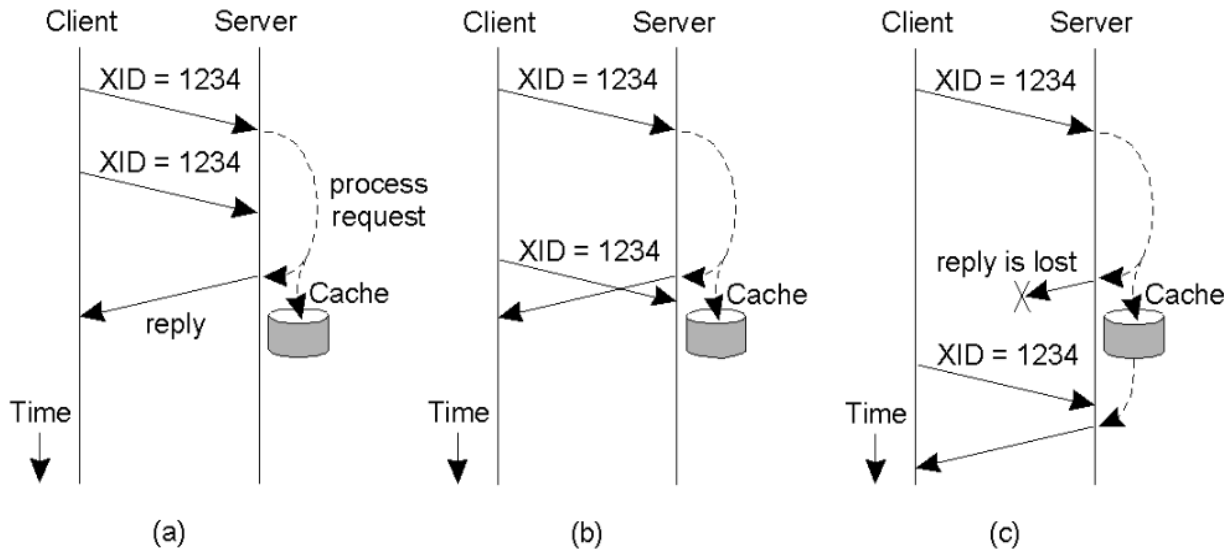

Figure 21.11: RPC retransmissions using a duplicate request cache

For `NFS` request over UDP, both the client and the server have to implement timeout and retransmission mechanism due to unreliability of UDP protocol. Generally RPC file service model is adopted. Retransmitting an RPC request requires the RPC operation to be **idempotent** i.e. any number of execution of the same RPC request should produce same result. Idem-potency is achieved through a duplicate-request cache. Each RPC is assigned a transaction Id. The cache stores the results for a transaction. When ever a RPC request comes, it is first checked with the cache. If a cache hit occurs, the result is returned. If not, the transaction is executed, the cache is updated and the result is returned back. Figure 21.11 depicts retransmissions of RPC requests using a duplicate request cache.

21.1.5.10 Security

Secure RPC is used in NFS v4. NFS V4 , all client server communications are encrypted.

21.1.5.11 Replica servers

NFS v4 supports replication of servers. This is implementation