Distributed Systems
Fall 2025

Yuvraj Patel

Disclaimer: Slides prepared using multiple sources (UW-Madison – Remzi, Andrea, Mike; Cambridge – Martin Kleepman; Distributed Systems book by MVS/ AST; University of Edinburgh – Yuvraj Patel)

Agenda

Distributed File systems

- Network File System (NFS) (Today's Class)
- Andrew File System (AFS) (Next Class)
- Google File System (GFS) (Next Class)

Types of File Systems

Local file systems (FFS, ext3, ext4, LFS, etc.)

- Processes on same machine access shared files on the machine

Network file systems (NFS, AFS, etc.)

- Processes on different machines access shared files on a different machine
- Many client connect with a nearby single server

Virtual File System (VFS)

VFS is a virtual abstraction like local file system

- Provides virtual superblocks, inodes, files, and dentry
- Compatible with a variety of local and remote file systems

VFS helps in allowing the same system call interface to be used across different file systems

- Implementation related to how things work for each file system is different

Mounting

Directory Hierarchy

- Abstraction layer over multiple actual filesystems

Mounting

- Makes a filesystem accessible by mapping a filesystem at a hierarchical level
- Provide a mount point (a directory) to access the filesystem

Goals for Distributed File Systems

Fast + Simple crash recovery

- Both clients and file server may crash

Transparent access

- Can’t tell accesses are over the network
- Normal UNIX semantics

Reasonable performance

- Scale with number of clients

Building a Distributing File System

Network File System (NFS)

Think of NFS as more of a protocol than a particular file system

Many companies have implemented NFS since 1980s

- Oracle/Sun, NetApp, EMC, IBM

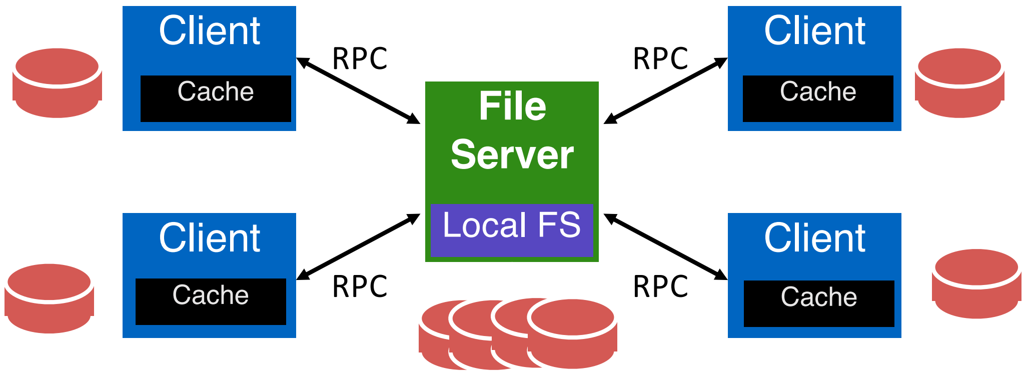
We are looking at NFSv2

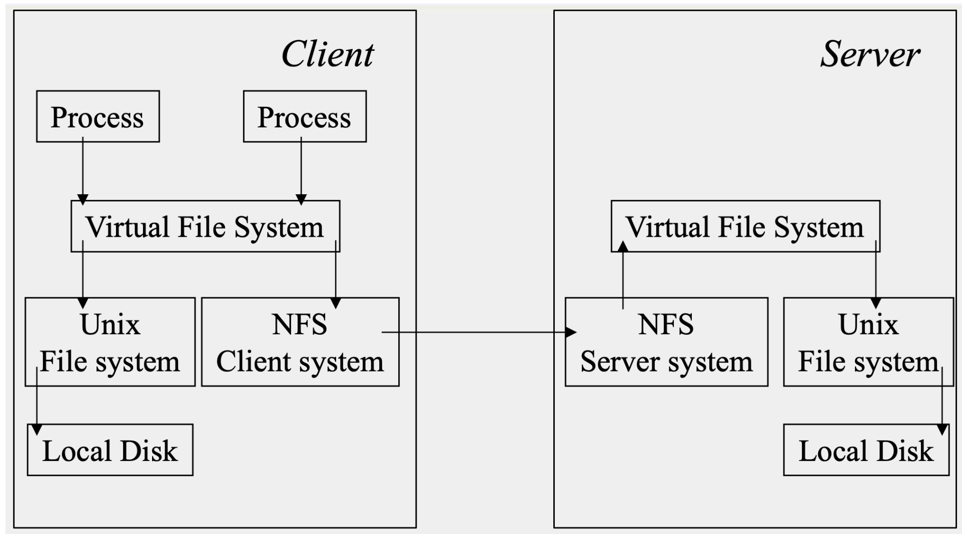
- Nfsv4 has many changes

Why look at an older protocol?

- Simpler, focused goals (simple crash recovery, stateless)

NFS Architecture

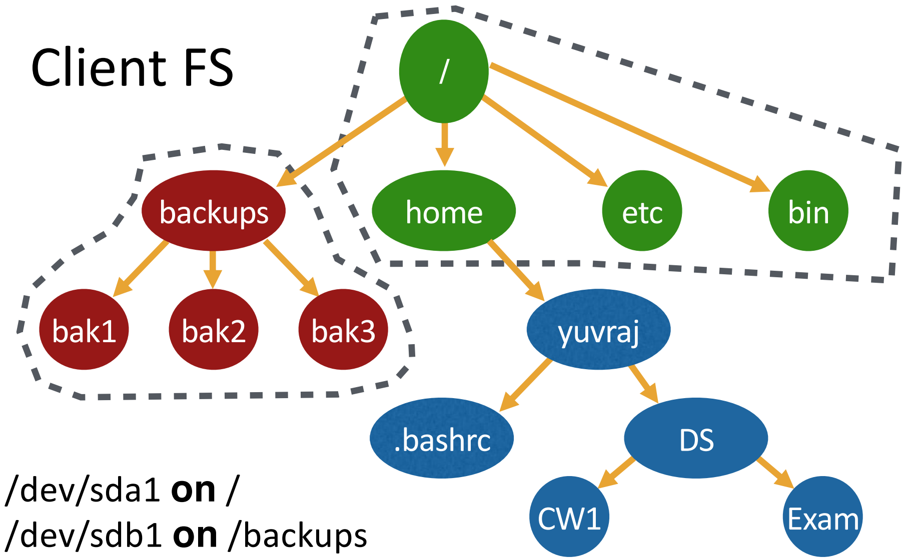




RPC: Remote Procedure Call
Cache individual blocks of NFS files

Exporting NFS

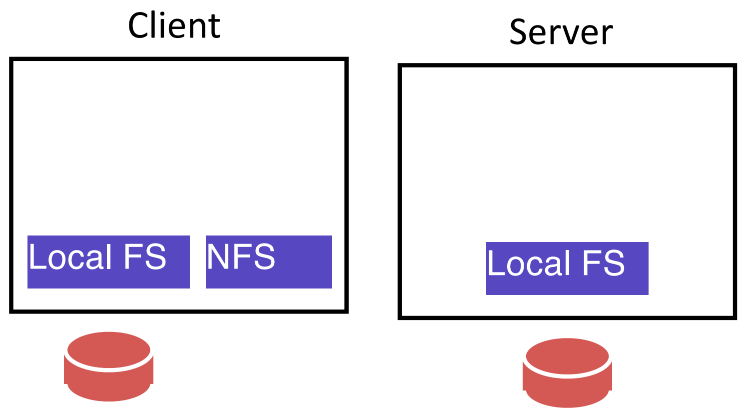
Client FS



/dev/sda1 **on** /
/dev/sdb1 **on** /backups
NFS **on** /home/yuvraj

Where will read to /backups/bak1 go?

Where will read to /home/yuvraj/.bashrc go?



Accessing Files using API

Multiple sys-calls to access the files

- open() – Open a file for reading, writing, or both
  - fd = open (const char* *Path*, int *flags*);
- read() – Reads the specified number of bytes cnt of input into the memory area indicated by buf
  - size_t read (int *fd*, void* *buf*, size_t *cnt*);
- close() – Close a file; Cannot access file after close
  - int close(int fd);

...

File Descriptor

While opening file, do expensive path traversal

Store the inode in descriptor object (kept in memory)

Do reads/writes via descriptor, which tracks offset

Each process has a file-descriptor table that contains pointers to open file descriptors

Integers used for file I/O are indexes into the per-process table

- stdin:0, stdout:1, stderr:2

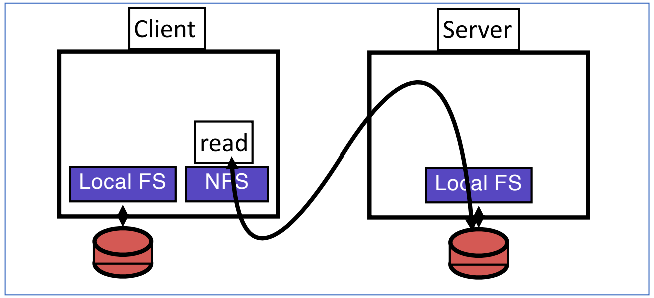
On close(), the descriptor object is removed

What do Clients Send to Server?

Strategy 1: Wrap regular UNIX system calls using RPC

- open() on client calls open() on server
- open() on server returns fd back to client

- read(fd) on client calls read(fd) on server
- read(fd) on server returns data back to client

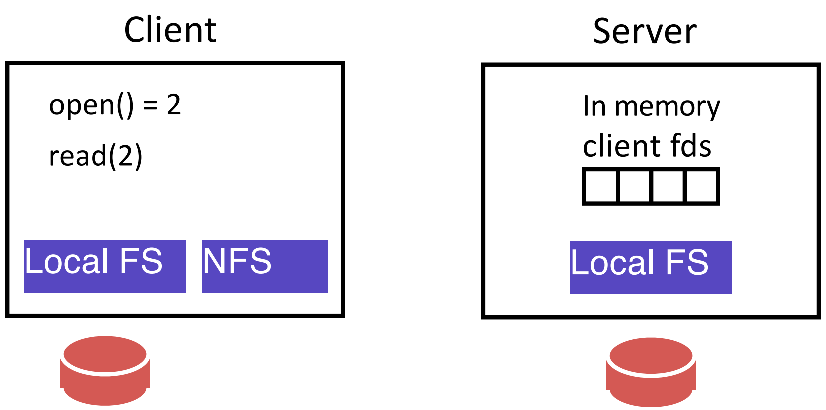


What do Clients Send to Server? (contd…)

Strategy 1: Wrap regular UNIX system calls using RPC

Problem: What about server crashes (and reboots)

```
int fd = open(“foo”, O_RDONLY);
read(fd, buf, MAX);
read(fd, buf, MAX);
… Server crash
read(fd, buf, MAX);
```



Recall: What is fd tracking?

What do Clients Send to Server? (contd…)

Strategy 1: Wrap regular UNIX system calls using RPC

Problem: What about server crashes (and reboots)

Potential Solutions

- Run some crash recovery protocol when server reboots
  - Complex
- Persist fds on server disk
  - Slow for disks
  - How long to keep fds? What if client crashes or misbehaves?

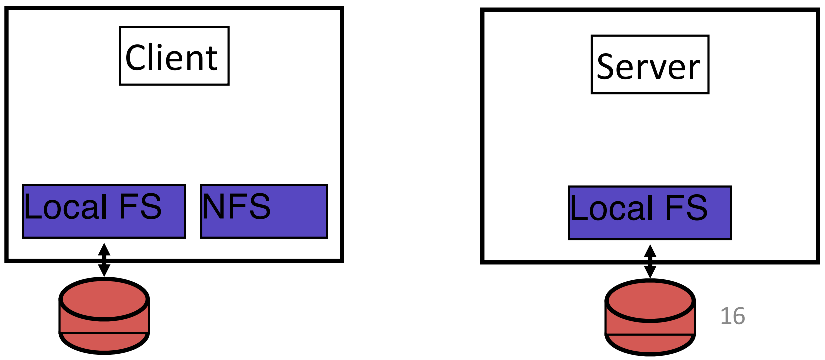
What do Clients Send to Server? (contd…)

Strategy 2: Every request from client completely describes desired operation

Use stateless protocol

- Server maintains no state about clients (that is necessary for correctness)
- Server can keep state only for performance (hints or cached copies)
- Can crash and reboot with no correctness problems (just slower performance)

Main idea of NFSv2



What do Clients Send to Server? (contd...)

Strategy 2: Stateless protocol

Need API change; Get rid of fds; One possibility:

```
pread(char *path, buf, size, offset);
pwrite(char *path, buf, size, offset);
```

Specify path and offset in each message

Server need not remember anything from clients

- Server can crash and reboot transparently to clients

Too many path lookups

What do Clients Send to Server? (contd...)

Strategy 3: Stateless protocol + Inode requests

```
inode =  open(char *path);
pread(inode, buf, size, offset);
pwrite(inode, buf, size, offset);
```

With some new interfaces on server for accessing by inode number

Correctness problem

- Inode not guaranteed to be unique over time
- If file is deleted, the inode could be reused

What do Clients Send to Server? (contd…)

Strategy 4: Stateless Protocol + File Handle

```
fh = open(char *path);
pread (fh, buf, size, offset);
pwrite(fh, buf, size, offset);
```

File Handle = <volume ID, inode #, generation #>

Opaque to client

- Client should not interpret internals

Generation count is incremented each time inode is allocated to new file/directory

What do Clients Send to Server? (contd…)

Final Strategy: Stateless Protocol + File Handle + Client Logic

Build normal UNIX API on client side on top of RPC-based APIs

Clients maintain their own file descriptors

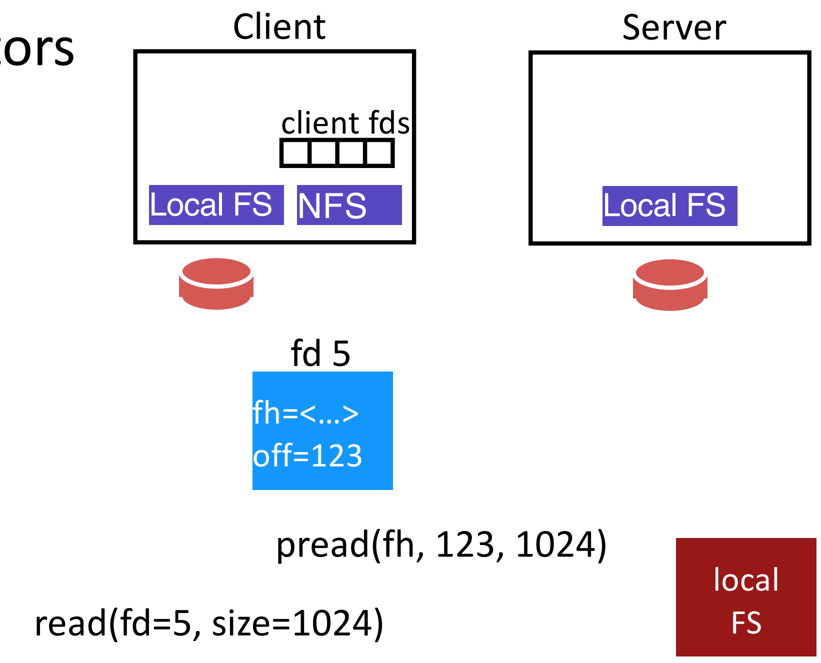
Client open() creates a local fd object

Local fd object contains

- File handle (returned by server)
- Current offset (maintained by client)

Client sends fh, offset, size to server

Server extracts inode from fh



Idempotent vs. Non-Idempotent Operations

Append operation adds content at the end of the file

append(fh, buf, size);

RPC often has “at-least-once” semantics

- May call procedure on server multiple times
- Implementing “exactly once” requires state on server, which we are trying to avoid

If RPC library replays messages, what happens when append() is retried on server?

- Could wrongly append() multiple times if server crashes and reboots

Idempotent vs. Non-Idempotent Operations

Idempotent Operations

- If f() is idempotent, f() has the same effect as f(); f(); …. f(); f();
- pwrite(), any read operation

Non-Idempotent Operations

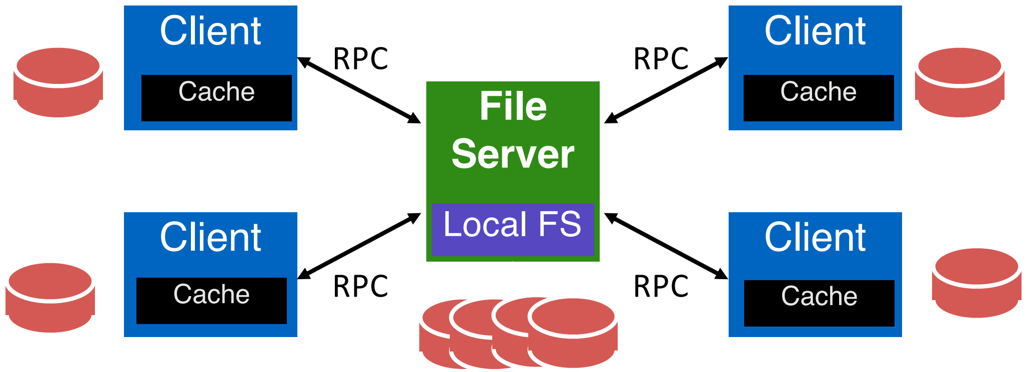
- Cannot be retried multiple times
- Append, mkdir, rmdir, creat

NFS Caching

With NFS, data can be cached in three places

- Server memory
- Client disk
- Client memory

How to make sure server and all client versions are in sync?

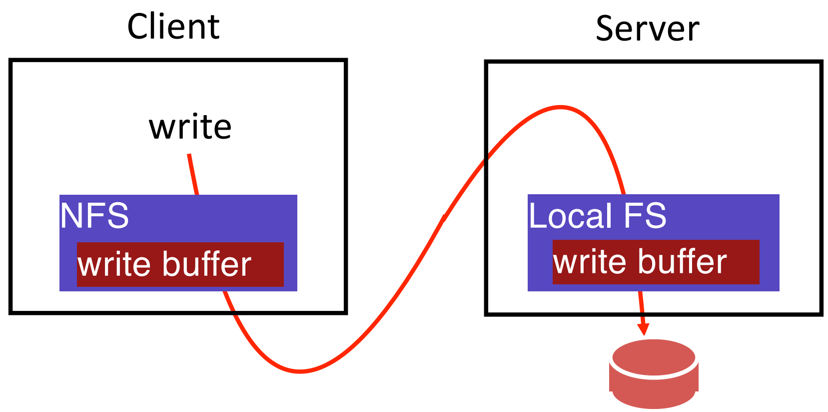


NFS Caching: Problem 1

NFS server often buffers writes to improve performance

Server might acknowledge write before pushed to disk

What happens if server crashes?



NFS Caching: Problem 1 (contd…)

NFS server often buffers writes to improve performance

Server might acknowledge write before pushed to disk

What happens if server crashes?

Solutions:

- Don’t use server write buffer (persist data to disk before acknowledging write) → Slow
- Use persistent memory → More expensive

NFS Caching: Problem 2

Clients must cache some data

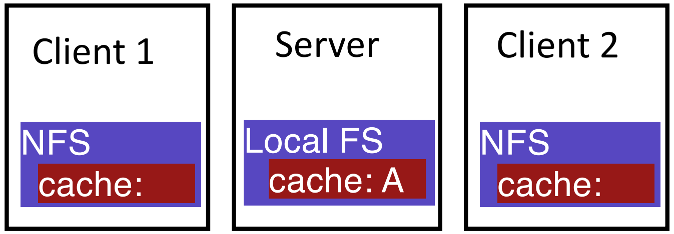
- Too slow to always contact server;
- Server would become severe bottleneck

Update visibility problem: Server doesn't have latest version

Some clients may see old version (different semantics than local file system)

When client buffers a write, how can server see update?

- Client flushes cache entry to server
- When should client perform flush?



NFS Caching: Problem 2 (contd…)

When should client perform flush?

Possibilities

- After every write (too slow)
- Periodically after some interval (odd semantics)

NFS Solution

- Flush on close()
- Other times optionally too – e.g., when low on memory

Problems not solved by NFS

- File flushes not atomic (one block of file at a time)
- Two clients flush at once can lead to mixed data

NFS Caching: Problem 3

"Stale Cache" Problem

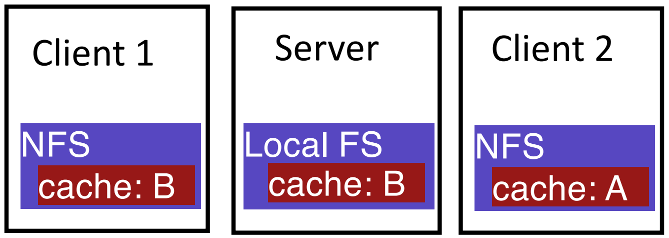
- Clients doesn't have latest version from server
- Clients may see old version (different semantics than local FS)

How can it get latest update?

- Maintaining state – push update to relevant clients

Stateless solution

- Clients recheck if cached copy is current before using data
- Recheck faster than getting data



NFS Caching: Problem 3 (contd…)

Client cache records time when data block is fetched (t1)

Before using data block, clients sends file STAT request to server

- STAT gets last modified timestamp (t2) for this file

If t2 > t1, then refetch data block

NFS developers found server overloaded

- Found stat accounted for 90% of server requests

Fix

- Client caches result of stat (attribute cache)
- Make stat cache entries expire after a given time (3 seconds)
- Clients could read data that is up to 3 seconds old

NFS Protocol Flow

Time	Client A	Client B	Server Action?
0	fd = open("file A");		
10	read(fd, block1);		
20	read(fd, block2);		
30	read(fd, block1);		
31	read(fd, block2);		
40		fd = open("file A");	
50		write(fd, block1);	
60	read(fd, block1);		
70		close(fd);	
80	read(fd, block1);		
81	read(fd, block2);		
90	close(fd);		
100	fd = open("fileA");		
110	read(fd, block1);		
120	close(fd);		

NFS Protocol

Time	Client A	Client B	Server Action?
0	fd = open(“file A”); Filehandle		Lookup for file A
10	read(fd, block1);		Read
20	read(fd, block2);		Read
30	read(fd, block1); Check cache; attr expired; call get_attr(); else use local copy	Latest attributes	Get_attr()
31	read(fd, block2);		Get_attr()
40		fd = open(“file A”); Filehandle	Lookup for file A
50		write(fd, block1); Keep local	
60	read(fd, block1); Check cache; attr expired; call get_attr(); else use local copy	Latest attributes	Get_attr()
70		close(fd); Send data to server	Write to disk
80	read(fd, block1); Check cache; attr expired; call get_attr(); expired; flush cache; fresh read again	Latest attributes	Get_attr()
81	read(fd, block2);		Get_attr()
90	close(fd);		
100	fd = open(“fileA”); Filehandle		Lookup for file A
110	read(fd, block1); Check cache; attr expired; call get_attr(); else use local copy	Latest attributes	Get_attr()
120	close(fd);		

Andrew File System (AFS)

Andrew File System: Developed at CMU in 1980s

Used in many universities (UoE home directories are AFS backed)

Goals

- More reasonable semantics for concurrent file access
- Improved scalability (many clients per server)
- Willing to sacrifice and statelessness

AFS Whole File Caching

Approach

- Measurements show most files are read in entirety
- open(): AFS client fetches whole file, storing in local memory or disk
- close(): Client flushes file to server if file was written

Convenient and intuitive semantics

- Use same version of file entire time between open() and close()

Performance advantages

- AFS needs to do work only for open/close (less load on server)
- Reads/writes are completely local

AFS Caching

AFS faces same problem as we discussed with NFS

Update Visibility

- How are updates sent to the server

Stale Cache

- How are other caches kept in sync with server?

AFS Caching – Update Visibility

AFS, like NFS, also flush on close

Buffer whole files on local disk; update file on server atomically

But what about concurrent writes?

- Last writer wins (i.e., the last file close wins)
- Never get data mixed from multiple versions on server unlike NFS

AFS Caching: Stale Cache

Stateful solution unlike NFS' stateless solution

Server tells clients when data is overwritten

- Server must remember which clients have the file open right now

When clients cache data on open(), ask for "callback" from server if file changes

- Clients can use data during this open() without caching

Clients only verifies callback when open() file (not every read)

- May not refetch file on next open()
- Operate on same version of file from open to close

AFS Callbacks: Dealing with State

Callbacks are good to handle the stale cache issue.

What about client and server crashes?

AFS Callbacks: Dealing with State (contd…)

Client crash

- After reboot, cached data might be on client disk
- Might read stale data from the cached copy
- Solutions
  - Evict everything from cache
  - Recheck specific entries before using

Server crash

- Lose track of all clients who have file open
- Solution – Tell all clients to recheck all data before next open

NFS vs AFS Protocols

Time	Client A	Client B	Server Action?
0	fd = open("file A");		
10	read(fd, block1);		
20	read(fd, block2);		
30	read(fd, block1);		
31	read(fd, block2);		
40		fd = open("file A");	
50		write(fd, block1);	
60	read(fd, block1);		
70		close(fd);	
80	read(fd, block1);		
81	read(fd, block2);		
90	close(fd);		
100	fd = open("fileA");		
110	read(fd, block1);		
120	close(fd);		

NFS Protocol

Time	Client A	Client B	Server Action?
0	fd = open(“file A”); Filehandle		Lookup for file A
10	read(fd, block1);		Read
20	read(fd, block2);		Read
30	read(fd, block1); Check cache; attr expired; call		Get_attr()
31	read(fd, block2); get_attr(); else use local copy	Latest attributes	Get_attr()
40		fd = open(“file A”); Filehandle	Lookup for file A
50		write(fd, block1); Keep local	
60	read(fd, block1); Check cache; attr expired; call get_attr(); else use local copy	Latest attributes	Get_attr()
70		close(fd); Send data to server	Write to disk
80	read(fd, block1); Check cache; attr expired; call		Get_attr()
81	read(fd, block2); get_attr(); expired; flush cache; fresh read again	Latest attributes	Get_attr()
90	close(fd);		
100	fd = open(“fileA”); Filehandle		Lookup for file A
110	read(fd, block1); Check cache; attr expired; call get_attr(); else use local copy	Latest attributes	Get_attr()
120	close(fd);		

AFS Protocol

Time	Client A	Client B	Server Action?
0	fd = open("file A");		Setup callback for A, send all of file A
10	read(fd, block1);		
20	read(fd, block2);		
30	read(fd, block1); Local read		
31	read(fd, block2);		
40		fd = open("file A");	Setup callback for A, send all of file A
50		write(fd, block1);	
60	read(fd, block1); Local read		
70		close(fd); Send back changes of A;	Server break call backs
80	read(fd, block1); Local read		
81	read(fd, block2); Local read		
90	close(fd);		
100	fd = open("fileA"); No callback; fetch file A again		Setup callback for A, send all of file A
110	read(fd, block1); Local read		
120	close(fd);		

When will server be contacted for NFS? For AFS?
What data will be sent? What will each client see?