Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanarayanan, M., Sidebotham, R.N., and West, M.J.
Scale and Performance in a Distributed File System
ACM Transactions on Computer Systems, Vol. 6, No. 1, February 1988, pp. 51-81.

1. **Initial Prototype.** What were the primary goals of the Andrew File System? Why did the authors decide to implement a usable prototype first? What were the primary problems they found with their prototype and what are the general implications?

- Scalability! (performance) + manageability
- Need experience to see issues, need existing system to evaluate ("plan to throw one away!")

Problems: Limited scalability + hard to admin

1) Too many overhead messages (TestAuth + GetFileStat)
   → Change protocol, reduce server interactions

2) CPU Load too high on server
   a) Pathname traversal all on server
      → Change protocol, move work to client
      → Change implementation; allow server to access w/ i-node
   b) Too many context switches
      → Change imp; use LWPs

3) Load-imbalance across servers
   - some files more popular
   ⇝ Implement Volumes

2. **Whole File Caching.** Why does AFS use whole file caching? Where are files cached? What are the pros and cons of this approach? For what workloads is this a good idea? When is it a bad idea?

+ No network traffic for indiv. reads/writes (just open/close)

+ Studies show most access whole file anyways

+ Usually small amount of sharing

+ Simplifies cache management

- On disk

- Need disks, bad if access only small portion of file

- Can't give exact same semantics as local since server doesn't see indiv. reads/writes

What happens when a file is opened?

3. **Client Caching.** AFS clients perform caching to improve performance. For read requests, how does a client know that its cached copy is up to date? When are writes sent from the client to the server? What happens when the server receives a write? What happens when a client crashes and reboots? What are the pros and cons of the AFS approach versus the NFS approach?

Read: up to date ~~[illegible]~~ by definition
(established when file was opened + file read in its entirety)
No checks @ this point!!

Write: Send changed data to server on close

Server breaks callbacks from other clients

Open: If still have callback, do not need to refetch (if in cache)
- If don't, then refetch

Client reboot: Throw away callbacks (may have missed them being revoked)

Pro: Good consistency model (clear)
Helps w/ scalability (fewer interactions)

Con: Requires state on server

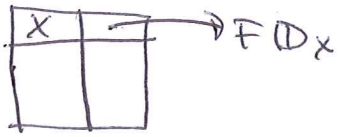
4. **Pathname Lookup.** AFS clients perform pathname lookups. What does an AFS fid look like? How does a client find the server that is responsible for a given volume? What steps take place when doing the pathname lookup for "/x/y.doc" (assume the client already has the root directory)? What portions of the needed information for a pathname lookup can be cached?

FID: <vol #, vnode #, unique id> → map to i-node w/ table lookup on server

- lookup in "vol loc.db" or server map
- every server has a copy of this (contact any)
- cache on clients too

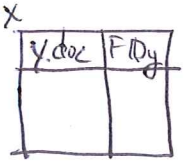
Important: No server info in FID so can change

Read / dir (assume already have this)



Read FIDx (get volume→server from map)

read FIDx



Read FIDy (get volume → server from map)

get data!

Cache all direntries! (/, x) using callbacks in same way as data

5. **Example Protocol.** Describe the operations that take place on the two separate client machines and the server for the following operations (specifically, when messages must be sent). Focus on the state of callbacks. Can you describe the consistency semantics of AFS? If two clients write to a file, which one will win (i.e., be store on the server)?

Time	Client A	Client B	Server Action?
0	fd = open("file A");		setup callback for A
10	read(fd, block1);		send all of file A
20	read(fd, block2);		
30	read(fd, block1);		
31	read(fd, block2);		
40		fd = open("file A");	setup callback
50		write(fd, block1);	send all of A
60	read(fd, block1);		
70		close(fd);	send back changes of A
80	read(fd, block1);		break callbacks
81	read(fd, block2);		
90	close(fd);		
100	fd = open("fileA");		
110	read(fd, block1);		
120	close(fd);		

local!!

local

local

nothing changed

no callback!! need to fetch A again

send A

further opens w/o other processes writing would not need to refetch file (callback not revoked)

Consistency:

Open-to-close semantics

guaranteed when open file to get contents from previous close

see no changes from other clients in that open session - always 1 version of a file (no intermixing)

Last-writer-wins: (closer) last client to close file will have its version sent to disk