Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B.
Design and Implementation of the Sun Network Filesystem
Proceedings of the Summer 1985 USENIX Conference, Portland OR, June 1985, pp. 119-130.

1. **Motivation.** What were the goals of NFSv2?

- Machine/OS Indepedence
  → RPC/XDR

- Crash Recovery

- Transparent Access
  → mount protocol, VFS layer, YP

- UNIX semantics

- Reasonable performance

2. **Stateless Servers.** In NFSv2, servers are stateless; what does it mean to be stateless? What are the advantages of having a stateless server? What does a client need to do when a server crashes? What does a server need to do when a client crashes? Is it okay to keep anything in server memory? What must be included in requests to a stateless server?

- server has no essential state in non-persistent storage (can be on disk)

- Adv: on crash, server can just restart + won't lose any needed info (just appears slower)

- Client may need to retry outstanding requests if see no reply

- Server does nothing (doesn't track which clients are active/connected)

- Memory okay as long as also on disk - just a cache for performance

- Every request must contain all info to complete operation (e.g. no open file descr.)

3. **Idempotent Operations.** Most NFSv2 operations are idempotent; what does it mean for an operation to be idempotent? Why are idempotent operations a good match for stateless servers? Give examples of idempotent operations in NFSv2. Give an example of a non-idempotent operation. How can non-idempotent operations be handled?

- Idempotent operations can be repeated any # of times w/ same results + no side effects

- Can resend idempotent op to stateless server + doesn't matter if request arrived (or was completed or partially completed) before

Idempotent:

- lookup(dirfh, name) → (fh, attr)
- getattr(fh) → attr
- read(fh, offset, count) → attr, data → specified!
- write(fh, offset, count, data) → attr
  → all the way to disk

Not Idempotent:

mkdir(dirfh, name, attr) → fh, attr

→ Different results if filename exists or not!

$\boxed{\text{Replay Cache}}$ on server; log of successful ops on disk

- crash, reboot, see new request → check reply cache so give same answer

4. **Performance Implications.** NFSv2 clients perform caching to improve performance. When are writes sent from the client to the server? Why does the stateless NFSv2 server cause performance problems for client write requests? How could this problem be fixed using something on the server? For read requests, how does the client know if the data in its cache is up to date? What problems did that cause? How was this overhead reduced?

- Writes are sent when file is closed locally, not immediately
- Stateless → writes must be persisted to disk
  → synchronous write protocol very slow
- Fix: Add NVRAM to server
- Reads are cached at individual block level
  - Send getattr() to check if any part of file has changed since last cached
  - If no change, use cached copy
    else get new copy from server
- Problem: 90% of calls to server are getattr()!
- Soln: attribute cache
  - Keep attributes for file local for 3 seconds
    for directory entries for 30s

5. **Implementation.** What does an NFSv2 file handle look like? Why? What were some of the auxiliary services that are needed to support NFSv2? Why are they needed? Does NFSv2 always provide the same semantics as a UNIX local file system? Are there any examples of differences? What were some operations that were difficult to handle?

filehandle: < inode #, gen #, FS id>
gen # ← needed because inode # can be reused

- VFS/vnode layers → generic layer so OS can handle multiple different fs
- mount protocol
- YP - credentials, uid same everywhere
- clock synchronization - apps use time (mod)

Not same: Process will see changes to files/dirs immediately when on same machine

Difficult: open, delete: local can still r/w file but others can't; NFS won't work since following stateless reads don't see file

soln: In client, change delete → rename if open (others can't see old file name)
delete when client closes

problem: if client crashes, garbage remains

Assume no prefetching

6. **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 what is currently contained in each client's cache and attribute cache. Can you summarize the consistency semantics provided by NFSv2?

Time	Client A	Client B	Server Action?
0	fd = open("file A"); $\longrightarrow$		lookup()
10	read(fd, block1); read $\longrightarrow$		read
20	read(fd, block2); read $\longrightarrow$		read
30	read(fd, block1); check cache; attr expired getattr();	okay; use local $\longrightarrow$	get attr
31	read(fd, block2); attr not	expired, use local	
40		fd = open("file A"); $\longrightarrow$	lookup
50		write(fd, block1); keep local	
60	read(fd, block1); attr. expired use local	data	getattr()
70		close(fd); write bl to server!	write to disk
80	read(fd, block1); attr expired; get attr.	CHANGED FILE - kickout	read()
81	read(fd, block2); not in cache	$\rightarrow$ read	read()
90	close(fd);		
100	fd = open("fileA");		lookup
110	read(fd, block1); attr expire; local ok	get new attr	getattr
120	close(fd);		[illegible]

- Ad hoc consistency

- See changes to data 3 seconds after file closed

(Refetch any of file if any block changed)

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 + Get File Stat)
   → 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 <u>disk</u>

- 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 by definition ~~if still have callback~~ (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 ~~[illegible]~~ 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

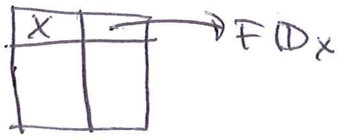
4. **Pathname Lookup.** AFS clients perform pathname lookups. What does an AFS fid look like? How does a client find the server that is responsible for a given volume? What steps take place when doing the pathname lookup for "/x/y.doc" (assume the client already has the root directory)? What portions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

FID : ⟨vol #, vnode #, unique id⟩ → map to i-node w/ table looku on server

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

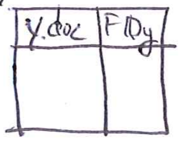
Important: No server info in FID so can change

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); local!!		
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); local		
70		close(fd); send back	changes of A
80	read(fd, block1); local		break callbacks
81	read(fd, block2); local		
90	close(fd); nothing changed		
100	fd = open("fileA"); no callback!! need to	fetch A again	
110	read(fd, block1);		
120	close(fd);	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