Design and Implementation of the Sun Network File System

Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon

Appears in USENIX Annual Technical Conference 1985

Presented by Hakim Weatherspoon

Goals

- Designed in early/mid 80s.
  - Before: each computer had its own private file disk + file system.
    - Fine for expensive central time-sharing.
    - Awkward for individual workstation.
- New model: One server, LAN full of client workstations. Not WAN.
  - Allow users to share files easily.
  - Allow a user to sit at any workstation.
  - Diskless workstations could save money

Implementation Goals

- Had to work with existing applications.
- Had to be easy to retro-fit into UNIX O/S.
- Had to implement same semantics as local UNIX FFS.
- Had to be not too UNIX specific
  - work w/ DOS, for example.
- Had to be fast enough to be tolerable
  - (but willing to sacrifice some).

Kernel FS Structure *before* NFS

- Specialized to local file system called Fast File System (FFS).
- Disk inodes.
- O/S keeps in-core copies of inodes that are in use.
  - File descriptors, current directories, executing programs.
- File system system calls used inodes directly.
  - To find e.g. disk addresses for read().
- Disk block cache.
  - Indexed by disk block #.

Why not a Network Disk (ND)?

- What was ND?
  - Server supplied a block store, with JUST read/write block RPCs.
  - This makes read/write sharing awkward.
  - Clients would have to carefully lock disk data structures.
- How is NFS different from ND?
  - Moves complex operations to server.
- Why might NFS be slower than ND?
  - Have to worry about consistency between multiple clients

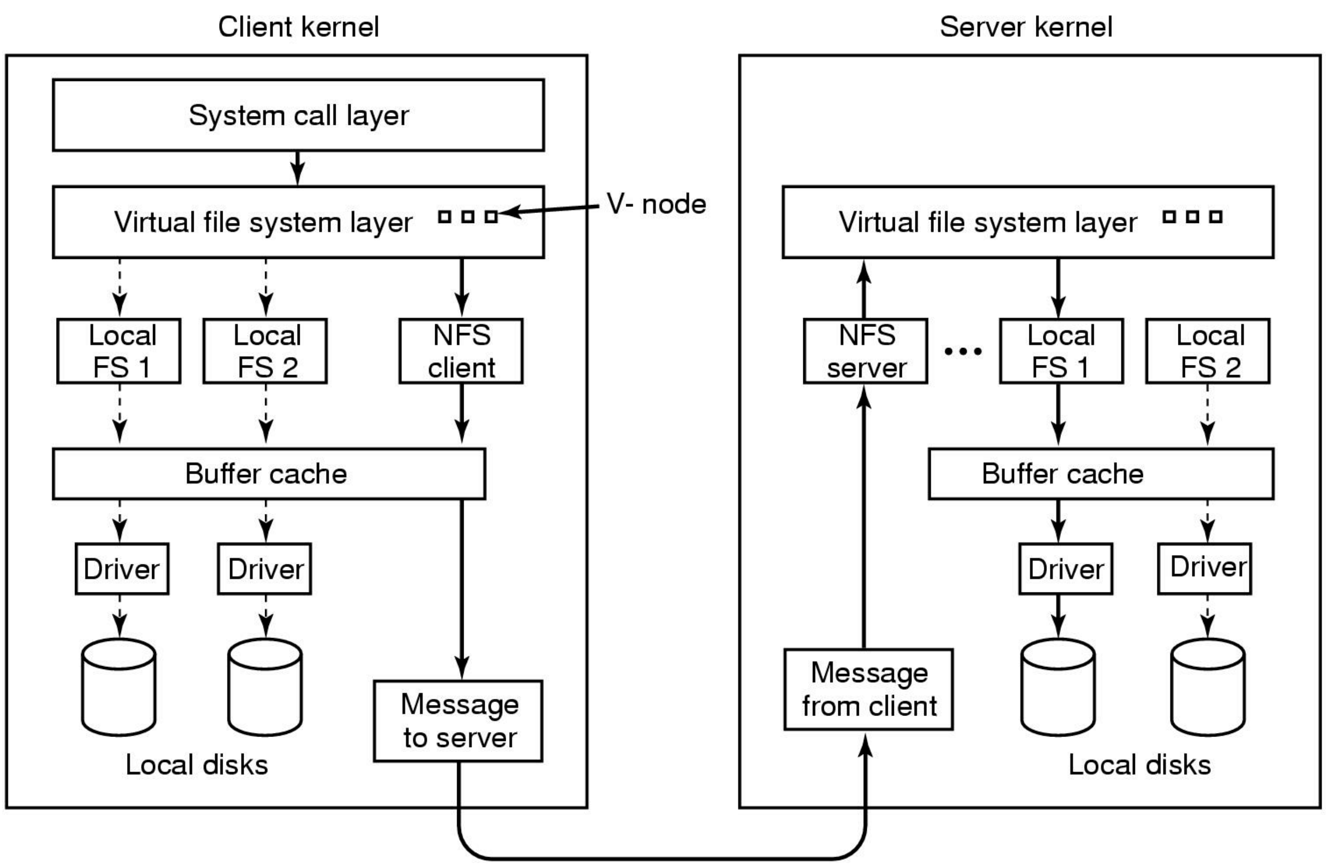
Virtual File System (VFS) Interface

- New "vnode" plan, invented to support NFS.
- Need a layer of indirection, to hide implementation.
  - A file might be FFS, NFS, or something else.
  - Replace inode with vnode object.
  - vnode has lots of methods:
    - open, close, read, remove.
  - Each file system type has its own implementation methods.
- What about disk cache?
  - Replaced with per-vnode list of cached blocks.

NFS Implementation

- Three main layers:
- System call layer:
  - Handles calls like open, read and close
- Virtual File System Layer:
  - Maintains table with one entry (v-node) for each open file
  - v-nodes indicate if file is local or remote
    - If remote it has enough info to access them
    - For local files, FS and i-node are recorded
- NFS Service Layer:
  - This lowest layer implements the NFS protocol

NFS Layer Structure



NFS Client/Server Structure

- Client programs have file descriptors, current directory, &c.
- Inside kernel, these refer to vnodes of type NFS.
- When client programs make system calls:
  - NFS vnode implementation sends RPC to server.
  - Kernel half of that program waits for reply.
  - So we can have one outstanding RPC per program.

NFS Client/Server Structure

- Server kernel has NFS threads, waiting for incoming RPCs.
  - NFS thread acts a lot like user program making system call.
  - Find *vnode* in server corresponding to client's vnode.
  - Call that vnode's relevant method.
  - Server vnodes are typically of type FFS.
  - This saves a lot of code in the server.
  - This means NFS will work with different local file systems.
  - This means files are available on server in the ordinary way.
  - NFS server thread blocks when needed.

How does NFS RPC designate file to read?

- E.g. a read RPC
- Could use file name.
  - Client NFS vnode would contain name, send in rpc.
  - Easy to implement in the server.
- Why doesn't this work?
  - Doesn't preserve UNIX file semantics.
    - Client 1: chdir("dir1");
      fd = open("file");
    - Client 2: rename("dir1", "dir2");
      rename("dir3", "dir1");
    - Client 1: read(fd, buf, n);
  - Does client read current dir1/file, or dir2/file?
  - UNIX says dir2/file.

The NFS File Handle

- What else goes in file handle, besides i-number?
  - File system ID - because server may server multiple file systems
  - Generation number - what is this?
    - Suppose an inode gets deleted & recycled while file/dir still open
    - Don't want old file descriptor referring to new file--very dangerous
    - Solution: Store generation number in inode on disk, change when recycled
    - What happens to read/write of old handle when generation number changes?
    - NFS stale file handle error
    - Sometimes: Inode of export point
  - So server can disallow lookup ("..") past export point
    - (not secure)

The NFS File Handle

- File systems already need a way to name files/inodes
  - E.g., How do directory entries refer to files?
    - Map name -> i-node number ("i-number”)
- Don't want to expose i-node number details to client
  - I.e. client should never have to make up a file reference.
- So file handles are opaque.
  - Client sees them as 32-byte blob.
  - Client gets all file handles from the server.
  - Every client NFS vnode contains the file's handle.
  - Client sends back same handle to server.

NFS RPCs

- lookup
- read
- write
- getattr
- create
- remove, setattr, rename, readlink, link, symlink, mkdir, rmdir, readdir
- ***(no open, close, chdir)*** – why not?
  - Requires state to be maintained at server

Example

- `fd = open("./notes", O_RDONLY);`
- `read(fd, buf, n);`
- Client process has a reference to current directory's vnode.
- Sends `LOOKUP(dir-vnode, "notes")` to server.
- Server extracts i-number from file handle.
- Asks local file system to turn that into a local vnode.
- Every local file system must support file handles...
- Calls the local vnode's lookup method.
- dir->lookup("notes") returns "notes" vnode.
- NFS server code extracts i-number from vnode, creates new file handle.
- Server returns new file handle to client.
- Client creates new vnode, sets its file handle.
- Client creates new file descriptor pointing to new vnode.
- Client app issues read(fd, ...).
- Results in READ(file-handle, ...) being sent to server.

Where does first File Handle come from?

- Every NFS RPC has to be contain a file handle
  - A valid file handle
- Server's mount daemon maps file system name to root file handle.
- Client kernel marks mount point on local file system as special.
- Remembers vnode (and thus file handle) of remote file system.

Crash Recovery

- Suppose server crashes and reboots.
- Clients might not even know.
- File handles held by clients must still work!
- That's why file handle holds i-number, which is basically a disk address.
  - Rather than, say, server NFS code creating an arbitrary map.
  - That is, server is stateless!

What if open file gets deleted by different client?

- UNIX semantics
  - file still exists until I stop using it.
- Would require server to keep reference count per file.
  - Would require open() and close() RPCs to help mantain that count.
- Which would have to persist across server reboots.
- So NFS just does the wrong thing!
- RPCs will fail if some other client deletes a file I have open.
  - this is part of the reason why there is no open() or close() RPC.

What about performance?

- Does *every* program system call go over the wire to the server?
- No: client cache for better performance.
- Per-vnode block cache, name->file handle cache, attribute cache.
- Can satisfy read()s, for example, from block cache.

What about consistency?

- Is it enough to make the data cache write-through?
- No: I read a file, another client writes it, I read it again.
- How do I realize my cache is stale?

Consistency

- What are the semantics of read and write system calls?
- One possibility:
  - read() sees data from most recent write().
  - This is what local UNIX file systems implement.
- How to implement these strong semantics?
  - Turn off client caching altogether.
  - Or have clients check w/ server before every read.
  - Or have server notify clients when other clients write.

NFS chooses poor consistency

- In V2 implementation described, very poor consistency
- In newer implementations, close-to-open consistency
- If I write() and then close(), then you open() and read(), you see my data.
- Otherwise you may see stale data.
- How to implement these strong semantics?
  - Writing client must force dirty blocks during close().
  - Reading client must check w/ server during open().
  - Ask if file has been modified since data were cached.
  - This is much less expensive than strong consistency.
  - Though maybe not very scalable; every open() produces an rpc.

Optimizations

- NFS server and block I/O daemons
- Caching!
  - Client-side buffer cache
    - (write-behind w. flush-on-close)
  - Client-side attribute cache
  - Name cache
- XDR directly to/from mbufs
- Fill-on-demand clustering, swap in small programs

What about Security?

- Server has list of IP addresses.
- Fully trusts any client with that address.
- Client O/S expected to enforce user IDs, send to server.

Other issues

- soft vs hard mounts
- replay cache.
- I can execute files I can't read.
- what if I open(), then chmod u=?
  - owner always allowed to read/write…
- dump+restore may wreck client file handles.

Next Time

- Read *NFS* and write review. *Turn into CMS before class*:
  - *A Toolkit for User-Level File Systems*. David Mazieres. Appears in *Proceedings of the USENIX Annual Technical Conference*, June 2001
  - *Implementing Remote Procedure Calls*. Andrew D. Birrell and Bruce Jay Nelson. Appears in *ACM Transaction on Computer Systems (TOCS)*, 1984
- Do Lab 0---**Due this Thursday, January 29th**
- Be prepared to select paper to present
  - Check schedule on website
  - Sign up to present on Thursday, January 29th
- Check website for updated schedule